UNDERSTANDING *and* EXPRESSING SCALABLE CONCURRENCY

Aaron Turon

April 19, 2013

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

to the

Faculty of the College
of Computer and Information Science
Northeastern University
Boston, Massachusetts

COLOPHON

This document was typeset using LaTeX, with a mixture of `classicthesis`[1] developed by André Miede and `tufte-latex`,[2] which is based on Edward Tufte's *Beautiful Evidence*. The bibliography was processed by Biblatex.[3]

Robert Slimbach's Minion Pro acts as both the text and display typeface. Sans-serif text is typeset in Slimbach and Carol Twombly's Myriad Pro; monospaced text uses Jim Lyles's `Bitstream Vera Mono` ("Bera Mono"). Donald Knuth's **Computer Modern** is used throughout, but I'm not saying where.

[1] http://code.google.com/p/classicthesis/

[2] https://code.google.com/p/tufte-latex/

[3] http://www.ctan.org/pkg/biblatex

Understanding and expressing scalable concurrency

NORTHEASTERN UNIVERSITY
GRADUATE SCHOOL OF COMPUTER SCIENCE
Ph.D. THESIS APPROVAL FORM

THESIS TITLE: Understanding and Expressing Scalable Concurrency

AUTHOR: Aaron Turon

Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.

Mitchell Wand — 2/22/13
Thesis Advisor *Date*

Olin Shivers — 2013/2/22
Thesis Reader *Date*

David Van Horn — 2/25/13
Thesis Reader *Date*

[signature] — 2/22/2013
Thesis Reader *Date*

[signature] — 2/22/2013
GRADUATE SCHOOL APPROVAL:

[signature] — 2/26/13
Director, Graduate School *Date*

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

[signature] — 2/26/2013
Recipient's Signature *Date*

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.

Abstract

The Holy Grail of parallel programming is to provide good speedup while hiding or avoiding the pitfalls of concurrency. But some level in the tower of abstraction must face facts: parallel processors execute code concurrently, and the interplay between concurrent code, synchronization, and the memory subsystem is a major determiner of performance. Effective parallel programming must ultimately be supported by *scalable concurrent algorithms*—algorithms that tolerate (or even embrace) concurrency for the sake of scaling with available parallelism. This dissertation makes several contributions to the understanding and expression of such algorithms:

- It shows how to understand scalable algorithms in terms of local protocols governing each part of their hidden state. These protocols are visual artifacts that can be used to informally explain an algorithm at the whiteboard. But they also play a formal role in a new logic for verifying concurrent algorithms, enabling correctness proofs that are local in space, time, and thread execution. Correctness is stated in terms of refinement: clients of an algorithm can reason as if they were using the much simpler specification code it refines.

- It shows how to express synchronization in a declarative but scalable way, based on a new library providing join patterns. By declarative, we mean that the programmer needs only to write down the constraints of a synchronization problem, and the library will automatically derive a correct solution. By scalable, we mean that the derived solutions deliver robust performance with increasing processor count and problem complexity. The library's performance on common synchronization problems is competitive with specialized algorithms from the literature.

- It shows how to express scalable algorithms through *reagents*, a new monadic abstraction. With reagents, concurrent algorithms no longer need to be constructed from "whole cloth," *i.e.,* by using system-level primitives directly. Instead, they are built using a mixture of shared-state and message-passing combinators. Concurrency experts benefit, because they can write libraries at a higher level, with more reuse, without sacrificing scalability. Their clients benefit, because composition empowers them to extend and tailor a library without knowing the details of its underlying algorithms.

Research is what I'm doing when I don't know what I'm doing.

—Wernher von Braun

Acknowledgments

In the 2007 PhD orientation at Northeastern, students were shown a curious graph. The x-axis read *Time* and spanned 0 to 5 years. The y-axis read *Happiness* and, although the units were unlabelled, the trajectory was clear: a fairly steady high for the first couple of years, followed by a precipitous drop and a slow climb that, in the end, never quite recovered. I remember thinking, *Five years to dig deep into the field that I love? There's no way that chart applies to me.* Little did I know.

No one can make it from naive enthusiasm to finished dissertation without a lot of support, both technical and emotional. I was fortunate to be supported by a committee with diverse perspectives, taste, and wisdom—and to have each member enter the scene at just the right time (in the order below):

- Mitchell Wand, my advisor, was originally assigned as my "faculty mentor" and immediately dispensed some advice: read papers until you find one that makes you say *I could do better than that.*[4] As the formative early reading gave way to research, Mitch had just one stipulation: I had to present a clear and compelling case for *why* and *how* we would do better. His combination of high standards and patience meant I got a *lot* of practice trying to make such cases.

- Claudio Russo was officially my internship supervisor at MSR Cambridge—but, with characteristic humility, he treated me as an equal collaborator. The summer we spent hacking together was one of the most enjoyable and rewarding periods of grad school, and I remain inspired by Claudio's restless curiosity and deep integrity, and grateful for his friendship.

- Doug Lea measures research impact in billions of computers,[5] yet he is one of the kindest and most approachable researchers or hackers you'll meet. As they say,[6] programs must be written for people to read, and only incidentally for machines to execute—and Doug's concurrency library is an excellent textbook, one that deeply informs the work in this dissertation.

- Olin Shivers knows how to write, and I returned to his writing over and over again for inspiration while producing my own. I also learned a great deal about the human side of research by being within earshot of Olin.

- Amal Ahmed arrived at Northeastern just before my thesis proposal was due, and enthusiastically agreed to help me "do better than" my previous work on concurrency verification. Her generosity with her time, her passion and raw mental horsepower helped make my final year my best one; her energy and attitude will serve as a North Star for years to come.

The Northeastern PRL has a persistently functional culture—from torture chambers, PL Jr., happy hour, and Elevator Pitch Idol, to a near total lack

[4] Incidentally, this slogan applies to one's own papers as well.

[5] His java.util.concurrent ships with Java, which runs on 1.1 billion desktop computers—and that's not even counting mobile phones or embedded devices.

[6] Abelson and Sussman, *The Structure and Interpretation of Computer Programs*

of ego and unflinching commitment to helping each other improve. While some of this culture is the handiwork of faculty, most of it is passed down through generations of remarkable students and post docs. I am indebted for the friendship, insight, and caring of an incredible number of them:

Dan Brown, *who can explain anything to anyone*;
Harsh Raju Chamarthi, *a true student of the masters*;
Steve Chang, *who knows when to be lazy*;
Jed Davis, *who inhabits all levels of abstraction*;
Christos Dimoulas, *who always remembers the big picture*;
Carl Eastlund, *humor amplifier*;
Tony Garnock-Jones, *metacircular visionary*;
Dave Herman, *who puts up with the Internet*;
Ian Johnson, *man of steel*;
Jamie Perconti, *who sees the Matrix in color*;
Jonathan Schuster, *whose research will someday be divulged*;
Justin Slepak, *master of the precision strike*;
Vincent St-Amour, *the contrarian who cares*;
Paul Stansifer, *purveyor of puns, collector of cleverness, and boxer of bats*;
Stevie Strickland, *who reminds us to have fun*;
Asumu Takikawa, *quietly succeeding from day one*;
Sam Tobin-Hochstadt, *who knows everything*;
Jesse Tov, *who puts us all to shame*;
Dimitris Vardoulakis, *who keeps making it happen*;
David Van Horn, *enigmatic and admired*.

I will miss you all.

There are a number of others who stepped in at one crucial juncture or another. Anne Rogers and John Reppy are the reason I got into research in the first place. Pete Manolios's seminar and our early work together laid a firm foundation for the rest of my time at Northeastern. Derek Dreyer has been a recurring presence, and now a collaborator, mentor, and reason to move across an ocean; his impact in one year makes me wonder if I'll even recognize myself in two more. Lars Birkedal and Jacob Thamsborg graciously offered to collaborate rather than compete, and I am still reaping the benefits.

The 2010 summer in Cambridge, UK was a turning point for me, in no small part due to the encouragement and enthusiasm of Philippa Gardner and Peter O'Hearn. Thanks also to Mike Dodds, Matthew Parkinson, and Viktor Vafeiadis, who warmly welcomed me into their branch of the separation logic family. MSR funded not just the internship that summer, but much of my time in grad school as well, for which I am very grateful.

But none of this could have happened without my family. For as long as I can remember, my parents have encouraged me to pursue my interests and provided me with a safe place in which to do so. Thank you, Mom and Dad.

And Jessica, my partner and my home: I am glad that I have the rest of my life to repay my debt to you, because it will take at least that long.

Aaron Turon
Saarbrücken
April 2013

Contents

List of Figures

Part I

PROLOGUE

1

Overview

▸ Concurrency and parallelism are distinct, and dealing with one without the other is preferable whenever it is possible. But sometimes concurrent programs must take advantage of parallel hardware. Sometimes achieving parallelism requires explicit concurrent programming. And, nearly always, the *implementation* of libraries (or languages) for parallelism demands careful cache-conscious concurrent programming—the very kind of programming these libraries enable application programmers to avoid.

The intersection of concurrency and parallelism is "scalable concurrency," in which concurrent algorithms are designed to scale gracefully with available parallelism—in some cases by increasing throughput, and in others by merely avoiding parallel *slowdown*.

A good example is the ubiquitous hashtable. To use a sequential hashtable in a concurrent setting, it suffices to introduce a single lock, and to surround every operation with an acquisition and release of that lock. Global locking provides exclusive access for the duration of each operation, forcing concurrent operations to take place in some nonoverlapping sequence. It is a tidy way to manage concurrency, but a disaster for parallelism. A *scalable* concurrent hashtable, by contrast, will allow operations from multiple processes to proceed in parallel (and hence concurrently), so long as those operations work on distinct keys of the table, or otherwise do not semantically interfere. There are a variety of strategies for scalability, ranging from fine-grained locking to lock-free algorithms, but they all have one thing in common: they increase concurrency purely for the sake of parallelism. In the end, a scalable hashtable should *externally* behave just like one protected by a lock, but *internally* it should encourage as much concurrency as it can get away with.

In addition to fully embracing concurrency, scalability requires attention to the architectural details of parallelism—especially the memory subsystem. Injecting concurrency sometimes entails additional, fine-grained coordination, which in turn requires cross-core (or worse, cross-socket) memory traffic. Just a few trips on the memory bus can dwarf the gains gotten from parallel processing.

"If we believe in data structures, we must believe in independent (hence simultaneous) processing. For why else would we collect items within a structure? Why do we tolerate languages that give us the one without the other?"

—Alan J. Perlis, "Epigrams on programming," #68

1.1 The Problem

Asking application programmers to grapple with scalability without succumbing to concurrency bugs is a tall order, so the proliferation of spe-

cialized libraries of concurrency primitives is unsurprising. For example, java.util.concurrent[1] (JUC) contains a rich collection of carefully engineered classes, including various kinds of locks, barriers, semaphores, count-down latches, condition variables, exchangers and futures, together with nonblocking collections like queues, skiplists, and hashtables. Several of these classes led to research publications.[2] A Java programmer faced with a concurrency problem covered by the library is therefore in great shape. (Intel's Threading Building Blocks (TBB)[3] provides similar benefits to C++ programmers.)

But to paraphrase Perlis,[4] a library with ten concurrency primitives is probably missing some.

Indeed, because libraries like JUC and TBB are such an enormous undertaking, they are inherently conservative. They implement only those data structures and primitives that are well understood and likely to fulfill common needs. Unfortunately, a client whose needs are not well-matched to the library is back to square one: it is generally not possible to extend or combine the primitives of a library into new primitives with similar scalability and atomicity properties.

For example, while JUC provides queues, sets and maps, it does not provide stacks or bags, and it would be very difficult to build scalable versions of the latter on top of the former. JUC's queues come in both blocking and nonblocking varieties, while its sets and maps are nonblocking only—and there is no way for users to extend or tailor the library to provide additional blocking support. Although the queues provide atomic (thread-safe) dequeuing and sets provide atomic insertion, it is not possible to combine these into a single atomic operation that moves an element from a queue into a set.

[1] Doug Lea, http://gee.cs.oswego.edu/dl/concurrency-interest/

[2] Lea (2000); William N. Scherer, III *et al.* (2006); William N. Scherer, III *et al.* (2005); Lea (2005)

[3] http://threadingbuildingblocks.org/

[4] Epigram #11

1.2 MY THESIS

"Sometimes a scream is better than a thesis."

—Ralph Waldo Emerson

The goal of this dissertation is to improve the above state of affairs along two axes. First, by deepening our understanding of sophisticated scalable algorithms, isolating their essence and thus reducing the barrier to building new ones. Second, by developing new ways of expressing scalable algorithms that are abstract, declarative, and user-extensible.

To that end, the dissertation demonstrates two claims:

A. Scalable algorithms can be *understood* through linked protocols governing each part of their state, which enables verification that is local in space, time, and thread execution.

B. Scalable algorithms can be *expressed* through a mixture of shared-state and message-passing combinators, which enables extension by clients without imposing prohibitive overhead.

We elaborate on each claim in turn.

1.2.1 *Understanding scalable concurrency*

"If the data structure can't be explained on a beer coaster, it's too complex."

—Felix von Leitner, "Source Code Optimization"

Scalable concurrent algorithms abandon global locks in a quest for greater parallelism, a choice that fundamentally alters algorithm design: it is no longer possible to acquire and manipulate a consistent global view of data. Instead, algorithms must work *locally*, observing and modifying small parts of a data structure while maintaining consistency at all times. In other words, concurrent interaction happens at a very fine grain in both time and space.

Our approach is to understand scalable algorithms on their own terms: we think of each *piece* of the data structure (*e.g.,* each node of a linked list) as being subject to a *protocol* that tells its "life story"—how it came to be allocated, how its contents evolve over time, and how it eventually "dies" by being disconnected (or deleted) from the data structure. These protocols work at the same granularity of interaction as the algorithms do.

Just as separate parts of a data structure are linked together through pointers, local protocols can be linked together by placing constraints on their neighboring protocols. In the limit, local protocols and constraints can capture ostensibly global properties like reachability, even in the presence of fine-grained concurrent interaction. It thus becomes possible to explain algorithms that traverse a data structure while it concurrently changes underfoot: as the traversal proceeds concretely from one node to the next, it proceeds abstractly from one local protocol to another, each time using the knowledge it has obtained in one protocol to yield constraints on the next.

Using protocols, the correctness of an algorithm can be shown by considering a single, generic execution of its code, where:

- Concurrent threads are modeled abstractly through the protocol (*thread locality*);
- The execution is understood one step at a time, without the need to refer directly to past or future events (*temporal locality*);
- Each step is understood in terms of the protocol governing the portion of the state it interacts with (*spatial locality*).

Retaining these forms of locality is particularly difficult for algorithms involving cooperation[5] or multiple, semantically-connected races; we develop novel semantic techniques for doing so in Chapter 4.

The dissertation includes both formal and practical validation of protocols.

Formal validation comes in the form of a semantic model,[6] based on protocols, that is sound for *contextual refinement*, showing that no client can tell they are working with the scalable version of an algorithm instead of one based on coarse-grained locks.[7] Thus, clients can safely reason about a scalable library as if all access to it were sequentialized, while at the same time reaping the efficiency benefits of scalability. The model is the first to support direct assertions and proofs of refinement for scalable concurrency.[8] It is also

[5] For example, algorithms wherein one thread can complete an operation on behalf of another; see §2.4.5.

[6] Or, viewed differently, a logic.

[7] Modulo performance gains, of course.

[8] A few prior logics have been shown sound for the related property of *linearizability*; see §3.3 and Chapter 7 for further discussion.

the first model to consider such algorithms in the context of a high-level language, *i.e.,* one with higher-order functions, abstract and recursive types, and general (higher-order) mutable references. Of course, JUC is written in just such a high-level language—Java—and indeed *depends* on the abstraction facilities of Java to ensure that the private state of its data structures is hidden from clients.

PRACTICAL VALIDATION comes in the form of several worked examples exhibiting several dimensions of complexity found in scalable concurrent algorithms.

1.2.2 *Expressing scalable concurrency*

"I regard it as the highest goal of programming language design to enable good ideas to be elegantly expressed."

—C.A.R. Hoare

The most basic unit of abstraction in most languages—the function[9]—does not support the full range of composition needed in a concurrent setting. For example, the functions exported by JUC and TBB execute "atomically" in the sense described above, but clients cannot combine them to build larger atomic functions.

A common workaround with coarse-grained locking is to partially break the abstraction by exposing the internal, global lock—a technique that allows clients to combine critical sections but introduces the danger of deadlock. With scalable concurrency, such a tradeoff is not even possible: fine-grained locks can only be exported by giving up on abstraction altogether (and exposing all implementation details), and with lock-free data structures there is no lock to export!

Another technique for composing atomicity is software transactional memory (STM).[10] But this technique, too, is a non-starter for scalable concurrency: STM works by ensuring that *all* reads and writes in an atomic block are performed atomically, but the whole point of scalable algorithms is to cleverly avoid or minimize such checks.[11]

And anyway, atomicity is just the beginning: some situations call for the choice between two operations, such as receiving a message along either of two channels; others demand layering a blocking interface over a nonblocking abstraction; still others require incorporating timeouts or cancellation.

This dissertation presents two extensible, yet scalable concurrency libraries.

THE FIRST LIBRARY is based on Fournet and Gonthier (1996)'s *join patterns*, which specify that messages should be received along some set of channels simultaneously and atomically. It has long been known that join patterns provide an elegant way to solve synchronization problems: one merely describes the problem declaratively, in terms of channels and joins, and the implementation of join patterns does the rest. But existing implementations of join patterns have all used either locks or STM, and so none are suitable for *scalable* synchronization. We develop a new, lockless implementation that achieves scalability on par with custom-built synchronization primi-

[9] Or equivalently, object with methods.

[10] Shavit and Touitou (1995), "Software transactional memory"

[11] There are some relaxations of STM that would allow avoiding atomicity checks, but they must be explicitly programmed to do so; see Chapter 12.

tives by processing messages in parallel (and thereby avoiding centralized contention). The implementation (1) stores messages in lock-free data structures[12] for parallelism, (2) treats messages as resources that threads can race to take possession of, (3) avoids enqueuing messages when possible, and (4) allows message stealing ("barging"). The result: clients of the scalable joins library can express arbitrary new synchronization problems, declaratively, and the library will automatically derive a correct and scalable solution.

[12] See §2.5.2.

The second library is based on *reagents*, a new monadic abstraction designed for expressing scalable concurrent algorithms in a higher-level, composable style. Reagents blend together synchronous communication through message passing and atomic updates to shared state. Unlike STM, only *explicitly marked* updates are guaranteed to be atomic; other reads and writes are considered "invisible" to the reagent. Through this mechanism, an expert can faithfully express a scalable concurrent algorithm as a reagent, with the guarantee that no overhead will be introduced. But what is the benefit?

Because the reagent implementation explicitly marks the key atomic update(s) for an algorithm, it becomes possible to join two reagents into a larger one that executes both of their algorithms as a single atomic step—much as with join patterns. The invisible reads and writes of the algorithms remain invisible; only their key atomic updates are joined.[13]

[13] Upcoming hardware provides direct support for such "tiny transactions." By permitting invisible operations, reagents are better positioned to take advantage of this hardware support than STM is.

In addition to joins, reagents can be composed through atomic sequencing and choice. Using these operators, clients can extend and tailor concurrency primitives without knowledge of the underlying algorithms. For example, if a reagent provides only a nonblocking version of an operation like `dequeue`, a user can easily tailor it to a version that blocks if the queue is empty; this extension will work regardless of how `dequeue` is defined, and will continue to work even if the `dequeue` implementation is changed. Composability is also useful to algorithm designers, since many sophisticated algorithms can be expressed as compositions of simpler ones.[14]

[14] See §10.2.3, §10.2.4.

In principle, then, reagents offer a strictly better situation than with current libraries: when used to express the algorithms provided by current libraries, reagents provide a higher level of abstraction yet impose negligible overhead; nothing is lost. But unlike with current libraries, the algorithms can then be extended, tailored, and combined by their users. Extra costs are only paid when the new compositions are used.

We demonstrate the *expressiveness* of each library by using it to build a range of example concurrency primitives, and *scalability* by subjecting these examples to a series of benchmarks comparing them to hand-written counterparts, as found in the scalable concurrency literature.

1.3 ORGANIZATION

"TL;DR"

—The Internet

Chapter 2 provides the technical background and philosophical perspective that informs the rest of the dissertation. It defines concurrency and parallelism (§2.1), and then examines both concepts from several angles. It begins with an abstract and high-level account of concurrent programming that completely ignores parallelism (§2.2), teasing out two fundamental concerns that must be faced in any concurrent program: sharing and timing. After that, it jumps to the *bottom* layer of abstraction, and considers the architectural details affecting parallel programming (§2.3). These two perspectives come together in the final two sections, which explore common techniques for both building scalable concurrent algorithms (§2.4) and assessing their correctness (§2.5).

The rest of the dissertation is broken into two largely independent parts:

▸ UNDERSTANDING SCALABLE CONCURRENCY

Chapter 3 formalizes a calculus, F^{μ}_{cas}, which is a variant of the polymorphic lambda calculus extended with mutable references, **cas** and **fork**—the essential features needed to model scalable concurrent algorithms written in a high-level language. The chapter defines and discusses a memory consistency model (§3.2), refinement (§3.3), and atomicity (§3.4), in particular contrasting linearizability and refinement (§3.4.2). Some auxiliary technical details appear in Appendix A.

Chapter 4 introduces local protocols and develops, through examples, the key ideas we use to handle scalable algorithms: role playing via tokens (§4.3), spatial locality via local life stories (§4.2), thread locality via specification resources (§4.4), and temporal locality via speculation (§4.5).

Chapter 5 defines the syntax (§5.2) and semantics (§5.3 and §5.4) of a logic for refinement based on local protocols. The logic ties together a Kripke logical relation (traditionally used for showing refinement of one program by another) with Hoare triples (traditionally used for reasoning about a single program). The chapter sketches some proof theory for the logic (§5.5) and outlines a proof of soundness for refinement (§5.6). The full logic is summarized in Appendix B, and detailed proofs are given in Appendix C.

Chapter 6 exercises the logic of local protocols on a series of realistic examples employing several sophisticated techniques for scalability: elimination backoff (§6.4), lock-free traversal (§6.5), and helping (§6.6).

Chapter 7 discusses work related to local protocols and our semantic model.

▸ EXPRESSING SCALABLE CONCURRENCY

Chapter 8 introduces join patterns and Russo (2007)'s joins API for C♯. It shows how join patterns can solve a wide range of synchronization

problems, including many of the problems solved by JUC's primitives.[15] The full API is given in Appendix D.

Chapter 9 walks through the implementation of scalable join patterns, including excerpts of the core C♯ library code (§9.3 and §9.4) and optimizations (§9.5). It validates our scalability claims experimentally on seven different coordination problems (§9.8). For each coordination problem we evaluate a joins-based implementation running in both Russo's lock-based library and our new scalable library, and compare these results to the performance of direct, custom-built solutions. In all cases, the new library scales significantly better than Russo's, and competitively with—sometimes better than—the custom-built solutions, though it suffers from higher constant-time overheads in some cases.

Chapter 10 presents the design of reagents, both in terms of philosophical rationale (§10.1) and as motivated by a series of examples (§10.2, §10.5). The chapter shows in particular how to write all of the algorithms described in Chapter 2 concisely and at a higher-than-usual level of abstraction. It also demonstrates how the join calculus can be faithfully embedded into the reagent combinators (§10.3). The full API is given in Appendix E.

Chapter 11 walks through the implementation of reagents (in Scala) in significant detail, which reveals the extent to which reagents turn patterns of scalable concurrency into a general algorithmic framework. It includes benchmarking results comparing multiple reagent-based collections to their hand-written counterparts, as well as to lock-based and STM-based implementations. Reagents perform universally better than the lock- and STM-based implementations, and are competitive with hand-written lock-free implementations.

Chapter 12 discusses work related to scalable join patterns and reagents.

Finally, the dissertation concludes with Chapter 13, which summarizes the contributions and raises several additional research questions.

[15] Our versions lack some features of the real library, such as timeouts and cancellation, but these should be straightforward to add (§9.6).

1.4 PREVIOUSLY PUBLISHED MATERIAL

This dissertation draws heavily on the earlier work and writing in the following papers, written jointly with several collaborators:

- Turon and Wand (2011). *A separation logic for refining concurrent objects*. In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)
- Turon, Thamsborg, Ahmed, Birkedal, and Dreyer (2013). *Logical relations for fine-grained concurrency*. In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)
- Turon and Russo (2011). *Scalable Join Patterns*. In proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)
- Turon (2012). *Reagents*. In proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)

2

Concurrency meets parallelism

▸ Synopsis This chapter provides the technical background and philosophical perspective that informs the rest of the dissertation. It defines concurrency and parallelism (§2.1), and then examines both concepts from several angles. It begins with an abstract and high-level account of concurrent programming that completely ignores parallelism (§2.2), teasing out two fundamental concerns that must be faced in any concurrent program: sharing and timing. After that, it jumps to the *bottom* layer of abstraction, and considers the architectural details affecting parallel programming (§2.3). These two perspectives come together in the final two sections, which explore common techniques for both building scalable concurrent algorithms (§2.4) and assessing their correctness (§2.5).

"The moral of the story was that, essentially for the sake of efficiency, concurrency should become somewhat visible. It became so, and then, all hell broke loose."

—Edsger W. Dijkstra, "EWD 1303: My recollections of operating system design"

2.1 Concurrency is not parallelism

Scalable concurrency sits at the intersection of concurrency and parallelism, so it is best understood by first clarifying their relationship. The two concepts are distinct, and neither subsumes the other:[1]

> *Concurrency* is the overlapped execution of processes.
>
> *Parallelism* is the simultaneous execution of computations.

[1] This definition and the following discussion draws inspiration from several sources, old and new: Brinch Hansen (1973); Reppy (1992); Pike (2012); Harper (2011).

"Process"[2] here is meant broadly as a source of activity within a system, encompassing multiple threads of control in a single program, separate programs as managed by an operating system, and even external devices and human users. These processes are independent in that they do not exert direct control over each other, but instead interact through a shared medium, *e.g.,* shared memory or channels. Processes are concurrent when their activities ("executions") overlap in time, and can hence influence each other. Overlapped execution does *not* entail simultaneous execution. Concurrency can, for example, be implemented by preemptive scheduling onto a single processor, which interleaves the executions of the processes into a single sequence of actions.

[2] We will use the word "thread" as a (preferred) synonym throughout.

Interleaving the execution of threads onto a single processor is sometimes called *multiprogramming* or *multitasking*.

Parallelism, by contrast, requires simultaneous execution by definition—but of "computations" rather than processes. Computations are, loosely, what processes *do*. Thus, even in the straightline assembly code

```
r1 <- add(r2, r3)
r4 <- mul(r2, r5)
```

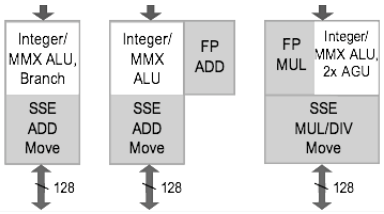
there is an opportunity for *instruction-level* parallelism, one that is aggressively realized by superscalar processors with redundant arithmetic logic units (ALUs). This pervasive source of parallelism *within* a process is wholly independent from concurrency *between* processes—at least from the programmer's perspective. Of course, if one computation depends on the result of another, they cannot be run in parallel, so the potential for parallelism is limited by the longest chain of sequential dependencies, which is called the *depth* (or *critical path length*) of the computation.[3]

Concurrency is a system-structuring mechanism. An interactive system that deals with disparate asynchronous events is naturally structured by division into concurrent threads with disparate responsibilities. Doing so creates a better fit between problem and solution, and can also decrease the average latency of the system by preventing long-running computations from obstructing quicker ones.

Parallelism is a resource. A given machine provides a certain *capacity* for parallelism, *i.e.,* a bound on the number of computations it can perform simultaneously. The goal is to maximize throughput by intelligently using this resource. For interactive systems, parallelism can decrease latency as well.

Parallelism always involves concurrency, but this fact may be profitably hidden behind an abstraction. For example, even the assembly language used in the snippet above is an abstraction of an underlying machine whose execution is both parallel and concurrent. The ALUs, control units, and other components of the machine are fixed, independent processes whose job it is to actually implement the machine's language. This is an example of *implicit parallelism*, in which an apparently sequential program is executed in parallel by communicating concurrent processes.

Conversely, concurrency can involve parallelism: the threads of an explicitly concurrent program may be executed in parallel.[4] The threads may originate naturally from the structure of the problem, as a reflection of independent problems, resources, or external processes. Alternatively, they may arise through *explicit parallelism*, where concurrent processes are used to directly control parallel resources.



Excerpt, Intel Nehalem per-core microarchitecture; note the redundant ALUs. http://en.wikipedia.org/wiki/Nehalem_(microarchitecture)

A machine supporting programmable, parallel execution is called a *multiprocessor*.

[3] Blelloch (1996), "Programming parallel algorithms"

"The point is that concurrency is not relevant to parallelism, even if the engineers who build our parallel computing platforms must deal with concurrency. Another way to say the same thing is that parallelism is a useful abstraction, and abstractions should never be confused with their implementations."

—Robert Harper, "Parallelism is not concurrency"

[4] The validity of a parallel execution strategy is actually quite subtle, since the shared memory model of parallel processors with independent caches is much weaker than the straightforward (but naive) model one imagines for concurrent programming. Recent language standards take this into account by weakening their model of concurrent shared memory (Manson *et al.* 2005; Batty *et al.* 2011; Boehm and Adve 2008).

2.1.1 *Scalable concurrency*

Whichever way concurrency and parallelism are mixed, a fundamental question arises: how can concurrent threads coordinate in a way that takes advantage of—or at least does not needlessly tie up—parallel hardware? In other words, how can we build concurrency constructs that *scale* with the number of parallel processing units?

To see the relevance and importance of this question, consider one of the most widely-used data structures in programming practice: the hashtable. If concurrent threads share access to a single hashtable, its implementation must guarantee that the threads do not interfere by, say, inadvertently overwriting each other's changes to an entry. From the point of view of concurrency alone,

it suffices to protect the hashtable with a single global lock, thereby providing each thread temporary but *exclusive* access for the duration of a read or write to the table. But such a solution fails to scale on even moderately parallel machines. The problem is not just the time spent waiting for the lock to be released, but that (for a "popular" hashtable) it takes a cache miss just to determine *whether* the lock is available.[5]

The Linux kernel uses a global hashtable called the *dcache* (directory entry cache) to drastically speed up pathname lookup. Prior to version 2.6 of the kernel,[6] concurrent access to this hashtable was mediated by a single global lock. Switching to a more scalable concurrent hashtable without global locking provided a 12% increase in *overall system throughput* on a machine with 16 CPUs, as measured by the SPECweb99 benchmark.[7] Thus a single lock in a single data structure turned out to be a massive scalability bottleneck for an entire OS kernel.

Hashtables exhibit "natural parallelism": concurrent threads are likely to access disjoint parts of the table, in which case their operations *commute*—and there is no intrinsic reason for them to coordinate such actions.[8] Nevertheless, because some hashtable operations do *not* commute, expensive synchronization is unavoidable in general.[9] A scalable hashtable must balance between these scenarios, avoiding synchronization where possible, but without sacrificing too much *absolute* performance in comparison to an efficient sequential implementation. Achieving the right balance is often a matter of choosing the right battles; since hashtable lookup is far more common than update, the biggest scalability gains come from enabling parallel reads with *zero* coordination. We examine the relevant algorithmic techniques in §2.4.

A scalable hashtable is useful not just for concurrent systems; it can also be a boon for explicit parallel programming. A simple but vivid example is the problem of *duplicate removal*: given a vector of items, return the items in any order, but without any duplicates.[10] Since the input is unstructured, any way of dividing it amongst parallel threads appears to require global coordination to discover duplicate items. The key to avoiding a multiprocessor game of "Go Fish" is to focus on producing the output rather than dividing the input. If threads share a scalable hashtable that allows parallel insertion of distinct elements, they can construct the correct output with (on average) very little coordination, by simply each inserting a segment of the input into the table, one element at a time.

Scalable concurrent data structures are also crucial for implicit parallel programming. The data structures are not used directly by the application (which is written sequentially), but are instead used by the runtime system to manage the work produced by the application, dynamically balancing it between the available processors. Work balancing can easily become a bottleneck, erasing the gains from parallelism, so implementations generally use scalable concurrent dequeues to implement strategies like "work stealing".[11]

In short, scalable concurrency is indispensable for a wide array of concurrent and parallel programming problems.

[5] We examine this point in much greater detail in §2.3.

[6] According to `www.kernel.org`, Linux 2.6.0 was released in December 2003.

[7] McKenney *et al.* (2004), "Scaling dcache with RCU"

[8] Herlihy and Shavit (2008), "The Art of Multiprocessor Programming"

[9] Attiya *et al.* (2011), "Laws of order"

[10] Blelloch *et al.* (2012), "Internally deterministic parallel algorithms can be fast"

[11] Blumofe *et al.* (1996), "Cilk: An efficient multithreaded runtime system"

2.1.2 *What scalable concurrency is not*

Lest we oversell scalability, however, it is important to keep a basic principle in mind:

The fastest way to communicate is to not communicate at all.

Many concurrent and/or parallel systems can be structured to avoid communication, or to communicate in only highly structured ways. Doing so is almost always a win, both in terms of performance *and* in terms of correctness. So scalable concurrency should only be applied to handle situations where communication or coordination is difficult or impossible to avoid.

Of course, as scalable algorithms evolve, so do the tradeoffs. For example, the duplicate removal algorithm described above is a fast and "obviously correct" implementation, but is made possible only by taking for granted the availability of a sophisticated scalable data structure: the hashtable. Building a parallel implementation of duplicate removal from scratch (while minimizing communication) would take significant ingenuity. As usual, one should use the right tool for the job.

We will not address the question of where and when to use scalable concurrency in application- or systems-level programming. As we have argued above, it is clear that it has *some* role to play in many different kinds of programming, and our goal is to better understand and express scalable concurrent algorithms when they arise.

2.2 TOP DOWN: THE PROBLEMS OF CONCURRENCY

Concurrency has long resisted treatment by a definitive computational or linguistic formalism; there is no agreed-upon analog to Turing machines or the lambda calculus for concurrent programming. One reason for this state of affairs is the typical division of concurrent programming into two warring paradigms, *shared state* and *message passing*, for governing thread interaction:

- IN THE SHARED STATE MODEL, threads interact by inspecting and altering shared, external resources.
- IN THE MESSAGE PASSING MODEL, threads interact by exchanging messages at explicitly-controlled times.

We are ultimately going to argue that *this division is a false dichotomy*—both semantically (in this chapter) and practically (in Chapter 10).

The most knock-down semantic argument is a simple one: almost any incarnation of one of the paradigms can easily encode[12] almost any incarnation of the other. The most sophisticated encodings produce idiomatic programs in either paradigm, and therefore transport common pitfalls from one paradigm into the other.[13] These encodings demonstrate that, at least for particular incarnations, the two paradigms are co-expressive. Thus from a

PARADIGM MYTHS, *a partial list*:

- Mutual exclusion is irrelevant to message passing.
- Proper use of monitors prevents races.
- Message passing avoids races.
- Message passing avoids deadlock.
- Message passing is "shared nothing."

[12] These encodings can be given locally, so that each set of primitives is merely "syntactic sugar" for a particular use of the other's. Hence each paradigm is *macro expressible* in terms of the other (Felleisen 1991).

[13] Lauer and Needham (1979), "On the duality of operating system structures"

semantic standpoint there is little reason to distinguish them, a point which has been borne out by semantic models that seamlessly support both.[14]

There is a deeper semantic reason that the dichotomy is a false one:

Shared state and nondeterminism are unavoidable in an expressive concurrent programming model.

That is, any "expressive" model of concurrency is inherently shared-state. In the next section (§2.2.1), we will give an informal argument for this claim by introducing a notion of *expressive interaction* in concurrent programming. Most shared-state or message-passing constructs are expressive in this sense.

The FUNDAMENTAL PROBLEM OF CONCURRENCY, in our view, follows from the inherently shared-state, nondeterministic nature of expressive interaction:

Concurrent programming is the management of sharing and timing.

There is no silver bullet for managing sharing and timing: as we discuss below, races for access to shared resources are often part of the *specification* for concurrent software, so a programming approach that makes races or sharing impossible is a non-starter for at least some kinds of problems.[15]

Some argue that the advantage of (synchronous) message passing is that its primitives weld together the management of sharing (a shared message queue) with timing (synchronized access to it), or view such a welding as a defining characteristic of message-passing programming.[16] But as we discussed above, shared-state primitives can be encoded even in synchronous message-passing systems, which means that it is possible to build up unsynchronized access to shared state in such systems. Conversely, many shared-state models *do* in fact weld synchronization to access, *e.g.,* software transactional memory and Brinch Hansen-style monitors.

In our view, concurrent programming presents a range of sharing and timing problems, each best solved by different "primitives"—and so to enable effective programming, a language should provide facilities to build *new* concurrency abstractions with varying approaches to sharing and timing. (There is nothing special about a message queue: it is one abstraction among many.) We examine traditional abstraction mechanisms in §2.2.4. Chapter 10 gives a deeper analysis, together with a synthesis of these ideas in the form of a new, more powerful mechanism for creating concurrency abstractions.

In the next section (§2.2.1), we define expressive interaction, introduce the problems of sharing and timing, and argue that they are intrinsic to concurrent programming. We then discuss sharing (§2.2.2) and timing (§2.2.3) more deeply, before explaining how *abstraction* is a key mechanism for mitigating—but not eliminating—sharing and timing problems (§2.2.4).

[14] Brookes (2002), "Traces, Pomsets, Fairness and Full Abstraction for Communicating Processes"

"*The trouble is that* essentially all the interesting applications of concurrency involve the deliberate and controlled mutation of shared state*, such as screen real estate, the file system, or the internal data structures of the program. The right solution, therefore, is to provide mechanisms which allow (though alas they cannot enforce) the safe mutation of shared state.*"

—Peyton Jones *et al.*, "Concurrent Haskell" (Emphasis theirs.)

"*Building software will always be hard. There is inherently no silver bullet.*"

—Brooks, Jr. "No Silver Bullet: Essence and Accidents of Software Engineering"

[15] Van Roy and Haridi (2004), "Concepts, Techniques, and Models of Computer Programming"

[16] Reppy (1992), "Higher-order concurrency"

2.2.1 *Expressive interaction*

Our definition of concurrency stipulates only that thread execution is overlapped, not how (or whether) this overlapping is observed. The style of thread

interaction is a defining characteristic of a model of concurrency—and there are many such models.[17]

[17] Though, as we argued above, most are equivalent from a semantic standpoint.

For concreteness, we will examine expressivity using archetypical, modern primitives in the shared-state and message-passing styles, but the discussion applies to a wide variety of formulations (including, *e.g.,* Erlang-style mailboxes):

Shared state		Message passing	
newRef :	$\forall\alpha.\ \alpha \to \mathsf{ref}(\alpha)$	newChan :	$\forall\alpha.\ \mathbf{unit} \to \mathsf{chan}(\alpha)$
get :	$\forall\alpha.\ \mathsf{ref}(\alpha) \to \alpha$	recv :	$\forall\alpha.\ \mathsf{chan}(\alpha) \to \alpha$
set :	$\forall\alpha.\ \mathsf{ref}(\alpha) \times \alpha \to \mathbf{unit}$	send :	$\forall\alpha.\ \mathsf{chan}(\alpha) \times \alpha \to \mathbf{unit}$

We take dynamically-allocated *references* (in the style of ML) as the basic unit of mutable state.[18] Cells are created with initial contents; calling get yields the last value that was set, or the initial contents if no set has occurred. References are unforgeable first-class values, so two threads can communicate through a reference if and only if both have access to it as a value.

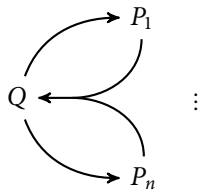
[18] In typical object-oriented languages, the basic unit of state is instead an object, which may posses many fields, each of which behaves like an independent reference cell.

On the message-passing side, we will consider *synchronous* channels in which a thread sending or receiving a value on a channel must wait until another thread performs the opposite action. The channels are *point-to-point*: many threads may attempt to send or receive on the same channel concurrently, but each sender is paired with exactly one receiver.[19] As with references, communication through a channel requires the involved threads to have access to it as a value.

[19] Most message-passing models allow *some* form of nondeterministic merging, either directly through channels or else through an explicit "choice" or "merge" construct.

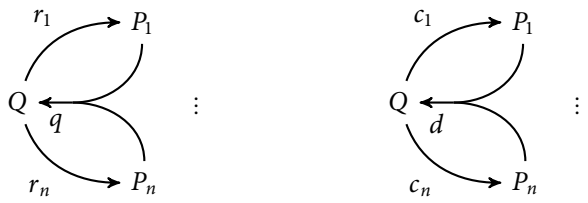
▸ An interaction model is expressive if it allows client-server communication, *i.e.,* if inter-thread influence can be arranged as follows:[20]

[20] This definition captures the distinction Van Roy and Haridi (2004) makes between *declarative* and more expressive, nondeterministic forms of concurrency.

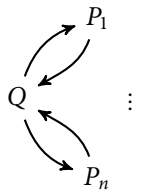


The thread Q plays the role of the server or resource. The other threads can make requests of Q and receive responses (influence is bidirectional). But, crucially, Q will service these requests *as they appear*, without stipulating which of the client threads makes the next request (influence is timing-dependent). Thus, the behavior of the cluster of threads may depend on the relative timing of the clients.

It is quite easy to create such a pattern of interaction with either shared-state or message-passing primitives. We simply introduce references (resp. channels) for each edge of influence, and use reads/writes (resp. sends/receives) to communicate:



Any expressive model of interaction immediately enables *shared access* and *timing dependence* between threads. We can see this fact at an abstract level by decomposing expressive interaction into two simpler patterns:



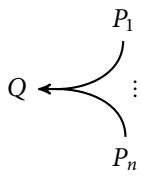
THE FIRST PATTERN EXHIBITS SHARING. What is being shared here is not a passive medium (*e.g.,* reference or channel), but rather bidirectional communication with a single thread. If that shared thread is truly influenced by communication, its behavior changes over time—and it is therefore stateful:

> *An object is said to "have state" if its behavior is influenced by its history.*
>
> —Harold Abelson and Gerald Jay Sussman, "Structure and Interpretation of Computer Programs"

> *"Once the memory is no longer at the behest of a single master, then the master-to-slave (or: function-to-value) view of the program-to-memory relationship becomes a bit of a fiction. . . . It is better to develop a general model of interactive systems in which the program-to-memory interaction is just a special case of interaction among peers. . . . To remove the active/passive distinction, we shall elevate [shared memory] to the status of a process; then we regard program variables x, y, . . . as the names of* channels of interaction *between program and memory. "*
>
> —Robin Milner, "Elements of interaction: Turing award lecture"

In other words, the existence of shared state follows from expressive interaction between threads. This insight dates from the dawn of concurrency theory: Milner showed how a wide range of passive communication media could be understood instead as active threads.[21] The key is that, whether active or passive, access to these objects or threads is shared amongst multiple other threads. The *timing* of this shared access, however, may be determined in advance, *e.g., Q* may wait for messages from the P_i threads in some specified order, or concurrency may be *cooperative* (so that threads yield access to each other only at explicitly-marked times).

[21] R Milner (1982), "A Calculus of Communicating Systems"



THE SECOND PATTERN EXHIBITS TIMING-DEPENDENCE through *arbitration*. That is, the P_i threads may send messages to *Q* at any time, but when multiple threads attempt to send a message concurrently, some arbitrary order of messages is chosen.

Arbitration makes computation unpredictable, but it is necessary precisely because timing is so unpredictable: it allows a thread to respond immediately when events occur, even if the relative timing of their occurrence is not known in advance. For example, an explicitly parallel program that tries to balance work between several threads may not know in advance how long each bit of work will take. To maximize

> *"An Arbiter is like a traffic officer at an intersection who decides which car may pass through next. Given only one request, an Arbiter promptly permits the corresponding action, delaying any second request until the first action is completed. When an Arbiter gets two requests at once, it must decide which request to grant first. . . . The Arbiter guarantees that there are never two actions under way at once, just as the traffic officer prevents accidents by ensuring that there are never two cars passing through the intersection on a collision course."*
>
> —Sutherland and Ebergen, "Computers without Clocks"

throughput, the assignment of work to threads *must be dynamic*; it must emerge from the timing properties of the system. Sometimes systems are also *specified* in a way that forces them to be timing-dependent. For example, operating system services are inherently race-prone: one program should be able to request—and potentially receive—service without any regard for other programs. Thus, the potential for *races* in thread influence is not just inevitable; it is sometimes desirable or even required! Indeed, *every* example of scalable concurrency we gave in §2.1.1 is built on that potential.

The danger, of course, is that such races can lead to systems whose *overall* observable behavior is likewise unpredictable—in particular, systems that sometimes produce wrong answers that are hard to reproduce. One should always use the smallest programming tool for the job,[22] and there is a good argument that timing dependence is the wrong default, that much concurrent programming (and certainly much parallel programming) can be done without it.[23]

"Concurrency is all about managing the unmanageable: events arrive for reasons beyond our control, and we must respond to them. A user clicks a mouse, the window manager must respond, even though the display is demanding attention. Such situations are inherently nondeterministic, but we also employ pro forma *nondeterminism in a deterministic setting by pretending that components signal events in an arbitrary order, and that we must respond to them as they arise. Nondeterministic composition is a powerful program structuring idea."*

—Robert Harper, "Parallelism is not concurrency"

[22] An epigram due to Olin Shivers.

[23] Van Roy and Haridi (2004), "Concepts, Techniques, and Models of Computer Programming"

Shared state alone does not introduce timing problems or nondeterminism; only models that offer some means of arbitration offer the possibilities and pitfalls of timing dependence. For example, Kahn networks provide bidirectional communication between threads, which are connected via channels in a potentially cyclic manner.[24] The channels, however, are strictly *one-to-one*: there is exactly one sending and one receiving thread associated with each. There is, moreover, no way for a thread to selectively receive, *e.g.,* by "peeking" on a channel to see whether a message is ready or by offering to receive on several channels at once. Once a thread has chosen to listen for a message on a particular channel (and hence from a *single* other thread), it is committed. The result is that Kahn networks are deterministic; their behavior is timing-independent. Other models like IVars,[25] and the very general LVars,[26] make similar tradeoffs.

[24] Kahn (1974), "The semantics of a simple language for parallel programming"

[25] Arvind *et al.* (1989), "I-structures: data structures for parallel computing"

[26] Kuper and Newton (2012), "A Lattice-Based Approach to Deterministic Parallelism with Shared State"

Nevertheless, expressive models of concurrency have a role to play, whether in handling the "last mile" of concurrent programming that cannot be addressed easily or efficiently with weaker models, or in implementing those weaker models in the first place. In any case, expressiveness is essential to scalable concurrency as we know it today.

We next consider the problems of sharing and timing more deeply.

2.2.2 *The problem of sharing*

"Sharing is caring."

—Unknown

If state is behavior being influenced by history, *shared* state is history—hence behavior—being influenced by someone else. The division of a program into parties need not fall along thread lines, or even coincide with static regions of code. It is possible to think of separate invocations of the same function, for example, as all sharing (and communicating via) state. Thus shared state is not endemic to concurrency; it can be found wherever control can "leave" and "come back," with a stateful object changing in between.

In a pure, expression-oriented language, data and control flow are tightly coupled: a subexpression is given control during evaluation, and returns control once an answer (some data) has been produced. Sharing, by contrast, allows data to flow in more subtle ways: a subexpression can communicate by altering a shared resource, instantly broadcasting changes to all other sharers, even those that do not receive the return value of the subexpression.

▸ The problem with sharing is its destructive effect on local reasoning. The tradeoffs are best understood through example, so assume that r is an integer reference and f is a unit to unit function, and consider the following expression:[27]

$$r := 0;\ f();\ \textbf{get}\ r$$

What values might this expression return? The answer is *it depends*—and what it depends on is precisely whether f has shared access to r and, if so, in what ways f and r interact. Without knowledge about f, we can only say that the expression returns an integer, a guarantee provided by the type system governing all access to r. When an object is shared arbitrarily, we cannot reason locally about its state: we must know something about other code that accesses it.

A simple way to restrict interaction between f and r is to keep r private. For example, if the expression allocates r locally,

$$\textbf{let}\ r = \textbf{new}\ 37\ \textbf{in}\ r := 0;\ f();\ \textbf{get}\ r$$

then we can conclude that the result will be 0, because f cannot possibly have or gain access to r.[28] When we know an object is not shared, we can reason about it completely locally.

In the absence of concurrency, we can also exercise significant *local* control over the *timing* of interactions with shared objects, and thereby mitigate the effects of sharing. For example, by changing the sequence of events,

$$f();\ r := 0;\ \textbf{get}\ r$$

we can again deduce that the expression always returns 0 with no knowledge about f. Whatever f does to r, the effect is wiped out by the subsequent local assignment—and there is no opportunity between that assignment and the following dereference for f to have any further impact. Sharing is only observable when control is relinquished,[29] and in a sequential language control must be given up explicitly. By controlling the timing of shared access, we control the times at which purely local reasoning applies.

The previous example is unsatisfying, though: our ability to deduce the result value depends on obliterating the contents of r. To permit meaningful sharing, yet retain a degree of local reasoning, we can employ *abstraction*.

[27] All of the examples in this section are written in the calculus $F^{\mu}_{\textbf{cas}}$ that we introduce in Chapter 3; it is a variant of System F with equi-recursive types (μ), general mutable references, and compare-and-set (**cas**, explained in §2.3.2).

[28] Meyer and Sieber (1988), "Towards fully abstract semantics for local variables"

[29] If external parties are not allowed to run, they cannot interact with shared objects.

Rather than sharing a stateful object directly, we instead share a set of operators that can access it:

```
oddCnt = let r    = new 37
             inc  = λ(). r := get r + 2
             read = λ(). (get r − 37) / 2
             test = λ(). isOdd(get r)
         in (inc, read, test)
```

This example inverts the setup of the previous ones: rather than explicitly invoking an unknown function, the exported object is passive until one of its methods is invoked. In other words, f has been replaced by an unknown *context* (client) into which the abstraction is placed.

Abstraction has a profound effect on local reasoning. It is fundamentally a way of restricting resource access to the code that "implements" the abstraction, which is usually[30] known when the abstraction is introduced. This "abstraction barrier" enables a modicum of local reasoning even in the presence of arbitrary sharing. In oddCnt, for example, we can deduce that no matter what the client does, the value of r will always be odd, and so test will always return **true**. The oddness of r is invariant because it is established when the abstraction is created, and preserved by every method of the abstraction—and because r itself never "escapes" to the client. Yet r is still *shared*, in the sense that abstraction-mediated access to it can be freely spread to arbitrary locations in the client code.

[30] More complex forms of access restriction are possible. For example, classes are an abstraction mechanism, but otherwise-hidden class members may be available to subclasses that are introduced afterwards.

Abstraction affects client-side reasoning as well. Because the client cannot access an abstraction's internal representation directly, its behavior remains fixed even if that representation changes.[31] For example, no client can tell the difference between oddCnt and

[31] Reynolds (1983), Mitchell (1986)

```
cnt = let r    = new 0
          inc  = λ(). r := get r + 1
          read = λ(). get r
          test = λ(). true
      in (inc, read, test)
```

because any sequence of method invocations on the two objects will yield the same results. The benefit is that clients can reason about their code using a simple, slow version of an abstraction, but actually link their code against a complex, fast one—a point we will discuss in greater detail in §2.2.4.

Despite the fact that the explicit call to an unknown function has disappeared, the oddCnt and cnt examples still retain significant control over timing in a sequential language. The reason is simple: when a client invokes one of their methods, it hands over control until the method returns. Without concurrency, only one such method can be invoked at a time.

Control over timing makes it possible to temporarily break invariants without harm. For example, replacing inc in oddCnt with a two-step version

```
inc = λ(). r := get r + 1; r := get r + 1
```

would not change the fact that test always returns **true**. The test method can only be invoked after inc has finished, so it cannot observe the intermediate state in which *r* is even.

2.2.3 *The problem of timing*

"The central issue lurking beneath the complexity of state, sameness, and change is that . . . we are forced to admit time into our computational models."

—Abelson and Sussman, "Structure and Interpretation of Computer Programs"

With concurrency, timing is not so easy to control.

Take the following all-too-real scenario:[32]

> You sit down with your favorite text editor[33] to perform a tedious task: inserting a pair of braces at the beginning and end of each of a few dozen lines. To speed up the process, you record a keyboard macro that processes one line, and leaves the cursor at the next line. You happily invoke the macro in quick succession, until you notice that braces are piling up, seemingly at random, at the beginning and ends of lines. What happened?

Interactive programs are always part of a concurrent system in which the user is a process. They are often written in a message-passing (or *event-driven*) style. In this case, it appears that keyboard macros are replayed by a process sending messages concurrently with the messages sent by the keyboard itself. The result is a nondeterministic interleaving of macro actions and fresh keyboard commands—including, it would seem, additional macro invocations, launching additional macro-replay threads. Once multiple macro replays begin interleaving with each other, the braces start piling up in odd ways.[34]

Timing is a delicate problem. The obvious way to "fix" the keyboard macros above is to change the message pipeline to allow "compound" messages, whose payloads are a sequence of commands to run without interruption. That change would avert the brace disaster above, only to replace it by another:

> You intend to add braces only to a certain collection of consecutive lines, and so you type the "invoke macro" chord what seems like a comparable number of times—inevitably, too many times. Seeing that braces are now being added to lines you did not intend, you quickly type the "emergency stop" chord, but to no avail: the system is irrevocably committed to some number of uninterruptable macro executions. You watch, glumly, wondering whether you will at least be able to type the "undo" chord the right number of times.

▸ The "easiest" way to solve timing problems is by waiting. We cannot reliably make one thread run faster than another, but we can make a thread run arbitrarily slowly, much to the fury of its clients. Waiting is often called *synchronization*, because it involves multiple threads coordinating their executions. After all, there is little reason to wait unless conditions are going to change, and a waiting thread must rely on concurrent threads to change those conditions. Synchronization is used to provide uninterrupted resource access (*e.g.,* mutual exclusion), to coordinate phases of a joint computation, to enforce an order of events (*e.g.,* producer-consumer coupling), and for many other coordination problems.

[32] Thanks to J. Ian Johnson for this bug.

[33] Emacs.

[34] When the system is quiescent, however, the total number of inserted open braces will match that of the close braces.

For a concrete example, we look again to the cnt abstraction, which provided a well-behaved implementation of a counter in a sequential language. Unfortunately, the abstraction fails when its client employs concurrency. Suppose the construct

$$\textbf{cobegin}\ e_1\ \|\ e_2\ \textbf{coend}$$

executes e_1 and e_2 concurrently, waits until both finish, and then returns unit. Then a client of cnt might execute

$$\textbf{cobegin}\ \mathsf{inc}()\ \|\ \mathsf{inc}()\ \textbf{coend};\ \mathsf{read}()$$

and get the result 1, rather than the expected 2. The introduction of concurrency has *empowered* the client: it can now instigate overlapped executions of the abstraction's code. An unfortunate interleaving of two inc invocations,

$$r\ :=\ \textbf{get}\ r+1 \qquad \| \qquad r\ :=\ \textbf{get}\ r+1$$

can result in both threads reading the current value of r, then both updating it to the same new value—dropping one of the increments. Concurrency has violated the assumption that each inc invocation could reason locally about r until explicitly giving up control, because the interleaving of concurrent threads causes control to be given up arbitrarily.

The lesson is that, with expressive interaction, a single thread never exercises unilateral control over a shared resource. It must instead depend on other threads to follow a *protocol* governing the resource at *all* times, because those threads might execute at *any* time.[35] Abstraction can guarantee protocol conformance by restricting direct access to the resource. For example, we can change the internal representation of cnt to include a lock,[36] and rewrite its methods to follow the protocol that access to r is only permitted when the lock is held:

```
lockCnt = let r    = new 0
              lock = new false
              inc  = λ(). acq(lock); r := get r + 1; rel(lock)
              read = λ(). acq(lock); get r; rel(lock)
          in (inc, read)
```

Abstraction previously enabled local reasoning through invariants on shared, hidden state, but those invariants only needed to hold at explicit control-transfer points. With concurrency, the invariants become continuously-followed protocols, but the benefit of abstraction is the same: we deduce that the protocols are globally followed by ensuring that they are locally followed. Locking protocols like the one above then recover something like explicit control-transfer points—namely, the points at which the lock is not held—so that invariants on the state *protected* by the lock only need to hold at such points. But the locking protocol itself must be followed at all times.

[35] Mechanisms like software transactional memory allow an individual thread to effectively gain unilateral control over memory during an atomic block, but to do so they impose an expensive, global protocol on all access to shared state.

[36] Here we treat the lock as a simple boolean value. The acq function pauses the thread until the lock's value is **false** and the thread wins the subsequent race to set it to **true**. We will discuss more realistic lock implementations in §2.4.6.

▸ Unfortunately, synchronization comes with pitfalls of its own. As illustrated by the "fix" to keyboard macros above,

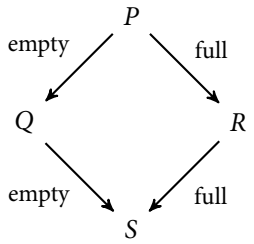
Synchronization is a temporary lapse in concurrency,

and such lapses are not always desirable. Concurrency is often introduced to improve latency and/or throughput (§2.1); synchronization cuts away at these benefits. It is especially damaging in a parallel setting, where waiting threads entail workless processors.

A more pernicious problem with synchronization is what Dijkstra termed[37] the *deadly embrace*: deadlock. The problem lurks whenever a thread is responsible for creating conditions that another thread is waiting for, but is also itself susceptible to waiting. For example, consider a simple message-passing system with four threads connected point-to-point:[38]

[37] Dijkstra (1965), "EWD123: Cooperating Sequential Processes"

[38] Brinch Hansen (1973), "Operating system principles"



The channels connecting the threads are finitely buffered: a thread trying to send a message on a "full" channel will wait until another thread has received one of the messages currently in the buffer. An unfortunate consequence is the potential for situations like the one above, in which

P waits for R to receive, S waits for Q to send,
R waits for S to receive, Q waits for P to send,

and the system grinds to a halt. It is worth stressing that this example uses only *unidirectional* communication. Deadlock does not require explicit cycles; implicit ones will do.

Altogether, then, the problem of timing is to allow the required or desired races (§2.2.1) and prevent the problematic ones, while guaranteeing global progress, maximizing throughput, and minimizing latency.[39] It is a difficult balancing act. There are numerous proposals for easing timing difficulties, ranging from design principles (*e.g.,* waiting on conditions in a fixed, global order) to significant language mechanisms (*e.g.,* software transactional memory), each with their own tradeoffs and pitfalls. But the most fundamental technique is one of the most familiar: managing timing in the same way we manage other software complexity, through layers of abstraction.

[39] Scalable concurrency adds additional goals: enabling parallelism while minimizing memory system traffic. See §2.3.

2.2.4 *The role of abstraction*

Abstraction separates concerns, and thereby isolates complexity. To implement a module, a programmer must choose how to represent and manipulate

data, often with an eye on performance. But when using the module, those choices can and should be irrelevant. They do not affect the "answers" (or, more generally, the observed behavior) that the module produces; they only affect the speed with which those answers are produced. As long as the module's client does not *observably depend* on the module's speed, it can benefit from speed improvements while ignoring how they are achieved.

Take the abstraction of mutable, finite maps. There are many ways to implement the abstraction—hashtables, skiplists, binary search trees, *etc.*—varying in performance and implementation complexity. But a client executing a sequence of operations on a map, *e.g.,*

$$\mathsf{insert}(37, \texttt{"hello"});\ \mathsf{insert}(37, \texttt{"world"});\ \mathsf{lookup}(37)$$

expects to get the same answer, `"world"`, regardless of the implementation used. It should be possible, moreover, for the client to deduce the answer ahead of time, on the basis of an *abstract* specification of a mutable finite map, which then serves as a formal interface between the client and the module implementation.

All well and good in the sequential case—but as we saw in the previous section (§2.2.3), concurrency empowers the client to interact with a module in ways that go beyond one-at-a-time call and response.[40] This new mode of client-module interaction threatens to reveal how the module handles (or fails to handle) timing issues, thereby breaking down the abstraction. It also raises new questions about how to specify such abstractions in the first place.

Concretely: what should the following concurrent client of a finite map expect to happen?

$$\textbf{cobegin}\ \mathsf{insert}(37, \texttt{"hello"})\ \|\ \mathsf{insert}(37, \texttt{"world"})\ \textbf{coend};\ \mathsf{lookup}(37)$$

The reasonable assumption for any "thread-safe" map is that exactly one of the two client subthreads will win the race, so that either of `"hello"` and `"world"` are valid answers, and nothing else is. We need a formal specification that enables such predictions.

For many methods, like insert, the right model of concurrent interaction is *none*—as if, in an amazing stroke of luck, the scheduler always avoids interleaving the executions of concurrent invocations, instead executing them one at a time. The beauty of this "atomic" specification of method execution is that it reduces the explanation of *concurrent* interaction to that of *nondeterministic sequential* interaction. But because it is mediated by an abstraction, this notion of "atomicity" refers only to the *observable* behavior of a module, as opposed to its implementation. The result is that clients can reason about a module as if, *e.g.,* it were implemented using a global lock, when in reality the implementation provides a much greater degree of concurrency (and scalability).

In short, abstraction reduces the degree to which concurrency is observable, while retaining its benefits.

Another way of understanding the role of abstraction is through the *granularity* at which thread execution is interleaved. The implementation

[40] Higher-order languages introduce similar challenges, because a client can pass a module a function (a "callback") which returns control to the client during the module's execution.

> *"In single-threaded programs, an object must assume a meaningful state only between method calls. For concurrent objects, however, overlapping method calls may be in progress at every instant, so the object may never be between method calls. Any method call must be prepared to encounter an object state that reflects the incomplete effects of other concurrent method calls, a problem that simply does not arise in single-threaded programs."*
>
> —Maurice Herlihy and Nir Shavit, "The Art of Multiprocessor Programming"

of an abstraction may be quite fine-grained, eschewing locks and instead allowing threads to interact with the data structure through a series of small, atomic operations on its hidden (but shared) state. If the client could observe the data structure's representation, it could observe the small steps being taken by each method invocation. But the abstraction barrier means that, instead, the client merely observes what the methods return, which can be understood in terms of a coarse-grained interleaving—one in which the operations "take effect" in a single step.[41]

[41] Similar benefits can be had even when methods are not atomic, but instead support meaningful non-sequential interaction. Those interactions can still be understood at a much coarser grain than their implementations; see §2.4.6.

▸ Abstraction does not "solve" the problems of sharing and timing, which are inherent to expressive concurrent programming (§2.2.1). As we saw with finite maps above, an abstraction can be shared by multiple client threads, which can race to access or update it, with nondeterministic results. What *has* been gained is the ability to reason about those races abstractly, at the level of *e.g.,* atomic updates to key-value pairs rather than complicated sequences of updates to a hashtable representation. It remains the client's responsibility to govern its use of the finite map abstraction by a meaningful protocol.

For example, consider the parallel duplicate removal algorithm of §2.1.1. There, a client uses a shared finite set abstraction (represented by a hashtable), creating several threads that insert elements into the set concurrently—and multiple threads might insert the same element at the same time. But the set is inspected only after all of the threads have completed their insertions. Using an atomic specification for set insertion, it is nearly trivial to see that the algorithm is correct. The insertion phase corresponds to inserting all of the elements of the input list into a set, one at a time, in *some* order; we reduce a concurrent execution to a nondeterministic, but sequential one. Since the contents of the set are only read out after the insertion phase, and (abstractly!) insertions into a set commute, the order of insertion clearly makes no difference. Thus the final set represents the input list without duplicates. It would be much harder to see that the client's race conditions were harmless if we had to reason in terms of the set *implementation*, especially when using a scalable concurrent set.

Finally, abstraction determines the rules of the game for scalable concurrency. On the one hand, an abstraction barrier allows a module to govern its internal state via a protocol of arbitrary cleverness, without fear that the client could somehow violate that protocol. On the other hand, the net effect of that protocol must be, *e.g.,* observable atomicity from the client's point of view. Protocols and their guarantees are the subject of Part 1 of this dissertation.

2.3 BOTTOM UP: THE PROBLEMS OF SCALABILITY

"The best computer scientists are thoroughly grounded in basic concepts of how computers actually work, and indeed that the essence of computer science is an ability to understand many levels of abstraction simultaneously. "

—Donald E. Knuth, "Bottom-up education"

The problems of concurrent programming are fundamental and timeless. *Scalable* concurrent programming, on the other hand, is very much contingent on the system architecture *du jour*. Of course, many of the basic techniques for scalability have a long shelf life—some state-of-the-art concurrency libraries

use variants of algorithms designed fifteen years ago for radically different hardware.[42] But in general, achieving high performance concurrency on multiprocessors requires attention to architectural details.

[42] The java.util.concurrent library, for example, uses a variant of Michael and Scott's lock-free queue.

Without a doubt, the most important architectural consideration is the interaction between multiprocessing and the memory subsystem. For a long time, the gap between effective CPU frequency and effective memory bus frequency has been growing:

> *CPUs are today much more sophisticated than they were only 25 years ago. In those days, the frequency of the CPU core was at a level equivalent to that of the memory bus. Memory access was only a bit slower than register access. But this changed dramatically in the early 90s, when CPU designers increased the frequency of the CPU core but the frequency of the memory bus and the performance of RAM chips did not increase proportionally.*
>
> —Ulrich Drepper, "What every programmer should know about memory"

While CPUs got faster, RAM got bigger. These architectural trends were largely hidden, however, through the use of *caches* to mask memory latency. Because most code naturally possesses temporal and spatial locality, caches provided the illusion of memory that is both large *and* fast.

2.3.1 *Cache coherence*

"The presence of multiple cores on a chip shifts the focus from computation to communication as a key bottleneck to achieving performance improvements. . . . High performance on-chip communication is necessary to keep cores fed with work."

—Jerger, "Chip Multiprocessor Coherence and Interconnect System Design"

Unfortunately, multiprocessing and caching conflict.

Caches support the abstraction of a single, global memory space, despite the fact that they permit multiple copies of a given memory cell to exist—the fast, cached version(s) and the slow version in RAM. The scheme works in the single processor case because the lowest level (fastest) cached version is *authoritative*. But multiprocessing brings a new complication: because each processor (or core) has at least one level of private cache, it is no longer possible to determine the authoritative state of a memory cell on a core-local basis, within a single cache hierarchy. Cores must coordinate.

Whether in phones[43] or servers, present-day commodity multiprocessors coordinate through *cache coherence*. Abstractly, cache coherence is a key *mechanism* through which a multiprocessor carries out its memory consistency *policy*:

[43] *e.g.,* the ARM Cortex-A9 MPCore.

> *We view a cache coherence protocol simply as the mechanism that propagates a newly written value to the cached copies of the modified location. . . . With this view of a cache coherence protocol, a memory consistency model can be interpreted as the policy that places an early and late bound on when a new value can be propagated to any given processor.*
>
> —Adve and Gharachorloo, "Shared memory consistency models: a tutorial"

The consistency policy varies by architecture, but we can assume:

> THE FUNDAMENTAL PROPERTY OF MEMORY MODELS[44]
> Memory is sequentially consistent for all well-synchronized programs.

[44] Adve and Gharachorloo (1996), "Shared memory consistency models: a tutorial"

The *sequential consistency* policy dictates that all memory accesses from all processors can be interleaved into a global *sequence* that is *consistent* with the semantics for memory, *i.e.,* a read returns the last value written. All processors observe the same order of events. We will discuss memory consistency in more detail in §3.2, in particular explaining what it means for a program to be well-synchronized.[45] For the moment, the important implication is that by issuing certain synchronization instructions, a program can demand a sequentially-consistent memory.

[45] All of the code in this dissertation is well-synchronized.

To guarantee sequential consistency in the well-synchronized case, architectures allow one core to gain exclusive access to a block of memory while performing a write to it. On the other hand, blocks can be freely shared between cores for read-only access, massively increasing the effective parallelism of the machine. The goal of cache coherence is to efficiently balance between the two kinds of access.

Cache coherence works by storing *ownership metadata* in core-local caches. At a minimum, each local cacheline has a status of *I* ("Invalid"), *S* ("Shared"), or *M* ("Modified"). These statuses reflect increasing access levels to a block: no access, read-only access, and read/write access, respectively. Coherence protocols maintain the system-wide invariant that, for each memory block,[46]

[46] Martin *et al.* (2012), "Why on-chip cache coherence is here to stay"

- either there is exactly one core holding the block with status *M*,
- or there are zero or more cores holding the block with status *S*.

As with a uniprocessor, the lowest-level cache holding a block is authoritative; cache coherence determines which among several local caches *at the same level* is currently authoritative.

The coherence invariant is reminiscent of reader-writer locking, but there is a fundamental difference: any core may obtain read or write access to a block *at essentially any time*. To put it more starkly, exclusive write access to a block can be revoked arbitrarily, without warning. Any changes to the block are flushed (committed), as usual for a cacheline eviction.[47] Allowing access to be revoked is essential; otherwise, one core could prevent another from making progress, indefinitely.[48] Revocation is primarily driven by a *cache miss* on the part of another core, which can be either a *read* or a *write miss* depending on the access required. The only constraint[49] on obtaining access is the coherence invariant, which guarantees mutual exclusion for writing but guarantees nothing about the *duration* of that write access.

[47] In some architectures, the block is sent directly to the core requesting it, rather than being routed through a higher level of the memory hierarchy.

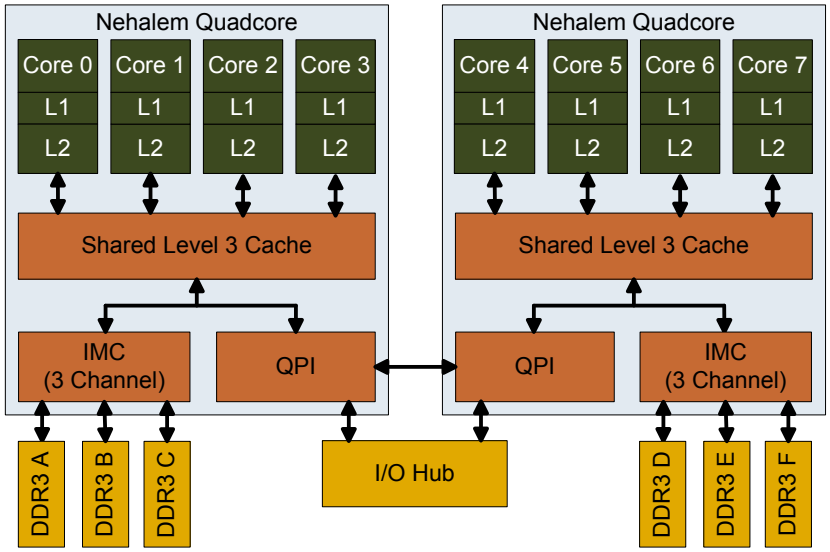
[48] *Cf.* §2.5.2

[49] Some architectures go further and provide a *fairness* guarantee in addition to the basic memory consistency guarantee.

The key remaining question is: how is the status metadata in the core-local caches coordinated? The answer to this question constitutes a *coherence protocol*, which is a major point of variance between architectures, and is coupled with other architectural decisions. It is nevertheless possible to build a rough cost model that applies to a range of current architectures. A key observation is that any coherence protocol involves communication between core-local caches. Such communication is generally routed through a higher

level of the memory hierarchy—one that is shared between the caches being coordinated—and this routing can make coherence very expensive.

For example, take the recent Nehalem architecture from Intel:



A Nehalem system can include multiple processors each with multiple cores, providing a complex memory hierarchy. The lowest two levels of the hierarchy (L1 and L2 caches) are core-private. The next level (L3 cache) is shared between the cores within a processor, but is processor-private. The last level of the hierarchy is RAM, but access to it is *nonuniform*: each processor has dedicated RAM banks (labeled "DDR3") which can be accessed relatively quickly, but processors can access each other's RAM through the *QuickPath Interconnect* (QPI). The read latency for each level of the hierarchy, measured in CPU cycles on a 2.933Ghz machine, is as follows:[50]

	L1	L2	L3	Local RAM	QPI	Nonlocal RAM
Latency (cycles)	4	10	38	191	120	191 + 120 = 311

To get a sense for the cost of coherence, we consider a few examples using the diagrammed Nehalem configuration above.

Suppose that Core 0 and Core 1 each hold a particular block in L1 cache (which must therefore be in state *S*), and that no other cores hold the block in local cache—and that, moreover, the second processor does not hold the block in its L3 cache. If Core 0 tries to write to the block, it will encounter a *write miss*, since it does not have write access. *This cache miss is purely the result of coherence*. After all, the data was already in L1 cache! Such misses are called "coherence misses." Servicing this coherence miss on Nehalem requires communication not just with the shared L3 cache, but also with the L1 and L2 caches of Core 1, resulting in a measured latency of around 65 cycles—more than an order of magnitude longer than the 4 cycles it would have taken to access the L1 cache of Core 0 without coherence.

The situation is much worse for cross-socket coherence. Consider a comparable scenario in which Core 0 and Core 4 hold a block in their L1 caches in

[50] Molka *et al.* (2009), "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System"

state *S*, and no other cores have it locally cached. If Core 0 writes to the block, the resulting coherence miss takes around 320 cycles to service—nearly *two* orders of magnitude slower than accessing the data in L1 directly.

The two scenarios leave Core 1 and Core 4, respectively, with the block in L1 cache in state *I* (invalidated). If the cores immediately try to read the block, they will take coherence misses with similar costs to the misses experienced by Core 0.

▸ There is a counterintuitive principle at work here. In a cache-coherent system, access locality can lead to cache *misses*, rather than cache hits.[51] This kind of "bad" locality is known as *contention*, and is characterized by multiple cores repeatedly accessing data where at least one core is trying to write to the data.[52] The cost of coherence misses tends to rise with the number of cores contending for a block; the resulting memory system traffic makes them more expensive to service than standard cache misses. We can summarize the situation as follows:

Contended access is slower than uncached access.

where "uncached" is relative to the level on the memory hierarchy encountering a coherence miss. A good cache coherence protocol will limit the number and expense of coherence misses, but contention for a memory block is a source of *unavoidable* misses. In the end, lagging memory latency comes back with a vengeance.

The cost of contention has a profound impact on the design of scalable concurrent algorithms. It is the reason that global locks do not scale: a contended lock adds the cost of a cache miss to *every* critical section, no matter how short those critical sections might otherwise be (see §2.4.2). It is the reason that semantically read-only operations should be implemented without writes, whenever possible—which precludes the use of even reader/writer locking. It encourages designs in which each write to shared data does a lot of work, so that fewer writes are needed overall. It can even trump asymptotic performance: some scalable concurrent algorithms asymptotically lag their sequential counterparts in space or time, but their parallel scalability gives better overall system performance. And the likelihood and effect of contention grows with increasing parallelism.

Starting with the next section, we turn to techniques for concurrent programming that are mindful of contention's cost. We begin with hardware-level primitives (§2.3.2) and work our way up to simple nonblocking algorithms implementing abstract data types (§2.4).

[51] This phenomenon is sometimes called *cache ping-ponging*.

[52] Ping-ponging can also result from *false sharing*, where cores are accessing distinct data that happens, unfortunately, to fall into the same cache block.

> "*The processor cores themselves run at frequencies where, at full speed, even in perfect conditions, the connection to the memory cannot fulfill all load and store requests without waiting. Now, further divide the available bandwidth by the number of cores, hyperthreads, and processors sharing a connection to the Northbridge and suddenly parallelism becomes a big problem. Efficient programs may be limited in their performance by the available memory bandwidth.*"
>
> —Ulrich Drepper, "What every programmer should know about memory"

2.3.2 *The foundation of interaction: consensus*

One problem with cache coherence, as we have described it, is that it is impossible for a thread to tightly couple a read and a write. In principle, another thread might gain exclusive access to the block in between the two

operations and perform a write of its own. This is not a problem that can be solved by *using* locks, because it is a problem we face when *implementing* locks in the first place! Suppose, for example, that we represent a lock using a simple boolean flag, where **true** represents that the lock is held by some thread, and **false** represents that it is free. We might arrive at the following naive spinlock implementation, written in a tail-recursive style:

$$\mathsf{acq}(\mathsf{lock}) \;=\; \textbf{if}\ \mathsf{get}(\mathsf{lock})\ \textbf{then}\ \mathsf{acq}(\mathsf{lock})\ \textbf{else}\ \mathsf{lock} := \textbf{true}$$

The implementation is hopelessly broken: after this thread observes that the lock is free, but before claiming it for itself, another thread could acquire the lock. There is not tight coupling between *observation* and *action*.

While there are some famous lock implementations that use only simple reads and writes,[53] in practice is it much more efficient to use a hardware operation that can *couple* reads and writes, sometimes known as *read-modify-write* operations. The most commonly available such operation is *compare-and-set* (CAS), which is usually[54] specified as follows:

$$\begin{aligned}&\textbf{cas} \;:\; \forall \alpha.\ \textbf{ref}(\alpha) \times \alpha \times \alpha \to \textbf{bool}\\ &\textbf{cas}(r, o, n) \;=\; \textbf{atomic}\ \{\ \textbf{if}\ \mathsf{get}(r) = o\ \textbf{then}\ r := n;\ \textbf{true}\ \textbf{else}\ \textbf{false}\ \}\end{aligned}$$

The idea is that **cas**, in one indivisible step, reads a reference and updates it only if it has an expected value, returning a boolean signifying which path was taken. Thus, **cas** *arbitrates* between multiple threads racing to perform an update, ensuring that the outcome of the race is coherent—there is just one winner. In other words, **cas** allows threads to come to a *consensus*.[55]

Using **cas**, we can write a correct spinlock as follows, again using a boolean lock representation:

$$\mathsf{acq}(\mathsf{lock}) \;=\; \textbf{if}\ \textbf{cas}(\mathsf{lock}, \textbf{false}, \textbf{true})\ \textbf{then}\ ()\ \textbf{else}\ \mathsf{acq}(\mathsf{lock})$$

Now if multiple threads attempt to acquire the lock concurrently, **cas** will arbitrate between them, producing exactly one winner, and thereby guaranteeing mutual exclusion.

Cache coherence *already requires* arbitration between processors to resolve races for write access to a block. A **cas** instruction can be implemented by ensuring that such access persists long enough to complete the instruction—which is essentially the same policy as for a primitive write operation. Thus the coherence implications for one "round" of attempts by $n + 1$ threads to **cas** a location are similar to those for n threads trying to read a location that one thread is writing to. Processors that already hold a block exclusively can perform a **cas** within that block relatively quickly, which means that *re*acquisitions of a lock implemented as above are relatively cheap.

All of the scalable concurrent algorithms studied in this dissertation use **cas** in one way or another.

[53] Dijkstra (1965), "EWD123: Cooperating Sequential Processes"

[54] The universal quantification here can be problematic; see Chapter 3 for details.

[55] Formally, the problem of *consensus* is to allow multiple threads to each propose a value, and for all of the threads to subsequently agree on a winning value. The **cas** operation is *universal*, because it can solve consensus for an arbitrary number of threads. This in turn means that **cas** can be used to build a "wait-free" (§2.5.2) concurrent implementation for *any* sequential data structure (Herlihy 1991).

2.4 THE RUDIMENTS OF SCALABLE CONCURRENCY: PERFORMANCE

"An algorithm must be seen to be believed."

—Donald E. Knuth, "The Art of Computer Programming, Volume 1: Fundamental Algorithms"

Taken together, the preceding two sections have isolated the problem of scalable concurrency:

- Section 2.2 argued that the fundamental challenge of concurrency is managing sharing and timing, a challenge that can be mitigated by abstraction.
- Section 2.3 argued that, with current architectures, the challenge of scalability is enabling parallelism while attending to cache-coherence effects like contention.

This section briefly surveys some of the most important techniques for building scalable concurrent abstractions in practice. The rest of the dissertation is focused on understanding these techniques more formally (PART 1) and expressing them more declaratively and composably (PART 2).

As observed in §2.2.4, a policy of mutual exclusion between method invocations is a simple way to *specify* the behavior of an abstract object whose methods should only interact sequentially. That specification leads immediately to an equally simple implementation strategy: take a sequential implementation, add a global lock to its internal representation, and delimit each method with a lock acquisition and release.[56] From the standpoint of scalable concurrency, we take this *coarse-grained locking* strategy as the baseline. It represents, in a sense, the worst we could do: not only does it rule out parallelism within the implementation, it also introduces a single memory location (the lock) which must be written to by every method—a recipe for contention and attendant clogging of the memory subsystem. When used by an increasing number of processors, the result is often parallel *slowdown* rather than speedup.

[56] Monitors embody this implementation strategy (Brinch Hansen 1973).

2.4.1 *Fine-grained locking*

If the state hidden within an abstraction is composed of *independent* pieces, a natural way to increase scalability is to protect those pieces by independent locks. For example, the representation for a hashtable involves some number of buckets of entries, and operations usually interact with a single bucket for lookup or insertion. A *fine-grained* locking strategy for a hashtable might associate a lock with each bucket, resulting in a significant increase in parallelism and decrease in contention. If two threads attempt to lookup items whose keys hash to different buckets, they can proceed in parallel, and in fact do not communicate at all. Although both threads must acquire a lock, the two locks involved can be arranged to sit in separate cache blocks,[57] allowing the threads to gain write access concurrently. Yet the correctness of the data structure is still fairly intuitive: each bucket is the authority on the keys that hash to it, and the per-bucket lock means that threads gain exclusive access to the relevant authority. Thus when two threads try to concurrently insert an

[57] That is, we can deliberately avoid false sharing (§2.3.1) by careful layout of data.

item with the same key, for example, the fact that they must acquire a common lock will force the operations to take place one at a time, in some order.

As long as the number and identity of buckets are fixed, the above strategy works fairly well. But we have ignored a crucial, if rare, operation: splitting the hashtable into a greater number of buckets as it grows. It takes great care to avoid a harmful race between threads trying to acquire a bucket's lock and a thread trying to refine the list of buckets. It is not a simple matter of introducing another lock to protect the buckets themselves; that would put us back to square one.[58] The lesson is that fine-grained locking is easiest to introduce for *fixed size*, flat data structures. Such data structures can be understood as a convenient coupling of smaller data structures, each protected by a (locally) coarse-grained lock. Linked or tree-like data structures, or those requiring dynamic restructuring, require a much more careful introduction of fine-grained locks.

[58] See Herlihy and Shavit (2008, chapter 13) for a survey of concurrent hashtable implementations.

Even when fine-grained locking is done correctly, it can still suffer from scalability problems. The most important problem is that read-only operations require a lock acquisition—and therefore a write to memory. Although these writes are spread out over several disparate locks, they can still result in contention (§2.3.1) between readers if the data is accessed frequently enough. Since read operations tend to dominate workloads, avoiding this unnecessary contention can be a significant win.[59] Doing so requires moving away from a simple protocol in which all access to each piece of hidden state is tied to some set of locks.[60] In particular, the fact that a thread is executing a read-only operation must be *invisible* to concurrent threads, since anything else would entail coherence misses. The techniques outlined in the remainder of this section support invisible reads.

[59] It was the key to improving the Linux *dcache* performance mentioned in §2.1.1, for example.

[60] *Cf.* the discussion of locking protocols in §2.2.3. We will formalize locking protocols in Chapter 4.

2.4.2 *Optimistic concurrency*

Locking is a pessimistic way to deal with timing problems: a thread assumes the worst, *i.e.,* that a concurrent thread will attempt to interfere, and prepares accordingly by *alerting* any other threads of its intent to access shared data, thereby forcing them to delay their work. If the lock is held for a long time, the delay drastically decreases parallel speedup.[61] But if the lock is held for a short time, it becomes a source of memory contention, ping-ponging between the caches of cores trying to access a shared resource.

[61] Amdahl (1967), "Validity of the single processor approach to achieving large scale computing capabilities"

In many cases, the pessimism of locking is unfounded: it may turn out that the threads are merely trying to read some shared data,[62] or are trying to perform writes that can safely proceed concurrently. *Optimistic* concurrency is an alternative approach, in which threads do not inform each other of their presence up front, but instead attempt to work independently for as long as possible, only checking at the very last moment that their work was not invalidated by another thread. When the optimism is well-founded, *i.e.,* no thread meaningfully interfered, the cost of coordination is thereby avoided.

[62] Reader/writer locking only helps when critical sections are lengthy. For short-lived lock acquisitions, both readers and writers still introduce coherence traffic, since they must still write to the lock itself.

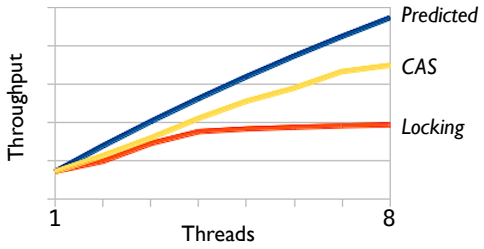
For a concrete example, we return once again to counters:

```
casCnt = let r    = new 0
             inc  = λ(). let n = get r
                         in if cas(r, n, n + 1) then () else inc()
             read = λ(). get r
         in (inc, read)
```

While very simple, this code illustrates the essence of the optimistic approach to dealing with timing: the retry loop. In particular, the inc method reads the current value of the counter without any explicit coordination with other threads, which means that it gains no *lasting* knowledge about the counter's state. The very instant after reading, for example, another thread could concurrently increment the counter. Nevertheless, inc carries on, optimistically computing the new value of the counter based on the earlier snapshot—and at the very last moment, atomically performs the update while checking whether its optimism was misplaced. If in fact another thread interfered, inc simply throws away its work and tries again.[63]

The read method stands to benefit the most from optimistic concurrency: because the data structure is so simple, it can be read in its entirety without any coordination or validation. If the workload of the counter is read-dominated, then most of the time its contents can be present in the caches of *multiple* cores in the "shared" state, recovering cache locality.

As it turns out, casCnt scales better than lockCnt even for workloads consisting *solely* of writes! Consider the following simple-minded benchmark: each threads executes a loop where for 98% of the iterations it increments a *private* counter, but in 2% of the iterations it increments a *shared* counter. We ran the benchmark on a machine[64] with two physical processors, each with four cores, and got the following results:



The "predicted" series is based on a simple application of Amdahl's law.[65] Neither counter implementation matches the prediction, but this is easy to explain: the cost of incrementing the shared counter rises with contention, while our simple prediction assumes a fixed cost (*i.e.*, a 2% sequential bottleneck).

It is difficult to say with certainty why the cas-based counter scales so much better in this case, but the likely cause is the fact that it requires a thread to gain exclusive access to only one cache line, one time, per successful increment. The spinlock (which included backoff; see §2.4.4), on the other hand, lives

[63] We are assuming here that the language properly implements tail recursion, so that inc will be compiled to a simple loop.

[64] In more detail: the machine is a 3.46Ghz Intel Xeon X5677 (Westmere) with 32GB RAM and 12MB of shared L3 cache. It has two physical processors with four hyper-threaded cores each, for a total of 16 hardware threads. L1 and L2 caches are per-core.

[65] Amdahl (1967), "Validity of the single processor approach to achieving large scale computing capabilities"

in a different cacheline than the counter's data,[66] which means two lines are needed in exclusive mode. Another explanation is that the cacheline holding the lock ping-pongs under high contention *between* lock acquisitions: the thread acquiring the lock obtains exclusive access and writes the lock; other threads immediately read the lock, so exclusivity is lost; then the original thread releases the lock, which requires it to *again* obtain exclusive access. In contrast, the **cas**-based algorithm does *all* of its work in one acquisition of exclusive cacheline access. In any case, coherence misses become a real problem once they must go across sockets, hence the rightward-turn of the lock-based scalability after four threads.

[66] Most lock representations are padded to avoid false sharing, but in this case sharing a cache line with the data would have been helpful.

While this particular benchmark should not be taken too seriously, it lends credence to our earlier claim: scalable concurrency demands attention to architectural details.

2.4.3 *Linked data structures*

Unlike counters, most data structures do not fit in a single machine word. So in general, a single **cas** operation cannot validate the contents of an entire data structure while performing an update; **cas** must be used in a more localized way. A significant amount of the work that goes into creating a scalable concurrent algorithm is arranging the representation of its data so that a single-word **cas** is meaningful.

Probably the most common representation strategy in practice is to use *linked* structures, where updates replace one node with another and **cas** compares nodes for identity.[67] The classic example is Treiber's lock-free stack implementation:[68]

```
Treiber = let h      = new (null)
          push   = λx. let t = new (null)
                       let n = cons(x, t)
                       let loop = λ(). let c = get h
                                       in t := c;
                                          if cas(h, c, n)
                                          then () else loop()
                       in loop()
          tryPop = λ(). let c = get h
                        in case c
                           of null ⇒ none
                            | cons(x, t) ⇒ if cas(h, c, get(t))
                                           then some(x) else tryPop()
in (push, tryPop)
```

[67] In particular, **cas** enables *pointer* comparison on the structures of the language, exposing the allocation and representation strategy of the compiler. We return to this point in Chapter 3.

[68] Treiber (1986), "Systems programming: Coping with parallelism"

Here the stack is represented by a linked list which is *internally* immutable; only the pointer h to the current head changes. The push method begins by allocating a new node, and then entering a retry loop. The loop works by (1) taking a snapshot c of the identity of the current head node, (2) linking the

new node's tail to c, and (3) updating the head assuming its identity has not changed in the meantime. The implementation of tryPop follows a similar optimistic strategy.

Treiber's stack illustrates that optimistic algorithms need not validate their snapshot against an entire data structure. For the case of push, this fact is quite easy to see: the new head node has a tail pointer that is directly validated against the current head pointer, and since that step takes place atomically (thanks to **cas**) the rest of the data structure is irrelevant. For tryPop, however, things are a bit more subtle. In particular, while tryPop does confirm that the current head pointer is the same as the preceding snapshot c, it does *not* confirm that the *tail* of that node is the same as the preceding snapshot t. Yet its correctness clearly depends on that fact.

The fact that a node's tail pointer never changes follows from two other facts:

- First, the stack employs abstraction to *hide* its internal state; its nodes never escape as values that the context could manipulate. Consequently, we can see by inspection that after a node is successfully added to the stack via push, its tail pointer is never changed. A tail snapshot lasts forever.

- Second, and much more subtly, the snapshot node c cannot be garbage-collected prior to the **cas**. This fact is relevant because **cas** exposes the *identity* of nodes (as pointers), so even if a node is not mutated *explicitly*, it can be mutated *effectively* by being reallocated with new contents. Luckily such a situation is never observable, because as long as a pointer to the node is maintained in order to perform such a comparison, the node will not be garbage-collected.[69]

In general, algorithms that depend on the identity of nodes to validate snapshots are susceptible to the so-called "ABA problem," in which a snapshot A is taken, after which the data structure is changed to state B and then back to A—which allows some **cas** to succeed even though some aspect of the data structure vitally changed in the interim. If, for example, the tail pointers of nodes in the stack could be mutated—say, to allow elements to occasionally be removed from the middle of the stack—then the tryPop method would suffer from an ABA problem by failing to validate its snapshot of the tail.

Treiber's stack is much more scalable than a stack protected by a global lock, even if the latter uses a more efficient sequential representation. It is nevertheless a quite simple example because, like the counter, there is a single mutable reference through which all concurrent activity takes place. More complex linked data structures permit mutation at a dynamically-growing set locations, and often require *traversing* the data structure even as the links between nodes are mutated. We will study such examples in Chapter 4.

[69] We assume garbage collection throughout. Adding explicit memory management to scalable concurrent algorithms is difficult, but there are now several established techniques for doing so (Michael 2004; McKenney and Slingwine 1998).

2.4.4 *Backoff*

Optimism is beneficial when it avoids unnecessary worry, but sometimes worry is warranted. For example, if several threads are attempting to quickly push large numbers of elements onto a Treiber stack, most of them will lose the race to **cas** most of the time. The result is a cascade of memory coherence traffic due to incessant retrying—where, again, most races will be lost, creating further retries, and so on. At some point, it would be better to give up on optimistic concurrency, and simply sequentialize access to the stack.

Fortunately, it is easy for a thread to determine when it should start worrying: every lost **cas** race is a sign of contention over the data structure. A simple strategy for avoiding cascading coherence traffic is *exponential backoff*, where a thread busywaits for (on average) exponentially longer periods of time after each lost **cas**.[70] Randomized backoff tends to evenly spread out access by a set of threads, effectively sequentializing access without introducing lock-like coordination. It is a pragmatic, heuristic way of improving the management of timing in a parallel system, but it has no semantic effect; correctness still depends on how the underlying algorithm uses **cas**.

[70] Agarwal and Cherian (1989), "Adaptive backoff synchronization techniques"

Backoff can go beyond busywaiting. Sometimes there is useful work that can be done before reattempting a **cas**. For example, in some concurrent data structures items are removed in two phases, a "logical" phase in which the items are marked as deleted, and a "physical" phase in which they are actually removed.[71] During backoff, a thread might traverse the data structure and remove nodes awaiting physical deletion.[72] Another example is elimination, which we discuss below.

[71] Heller *et al.* (2006), "A lazy concurrent list-based set algorithm"

[72] The java.util.concurrent skiplist does this.

2.4.5 *Helping and elimination*

We have argued that a key benefit of optimistic concurrency is that threads do not have to announce their intent to interact with a data structure, a benefit that is especially important for (morally) read-only operations. For updates, however, scalability can sometimes be improved by advertising intent in a *cooperative*, rather than *competitive*, fashion:

- In the competitive approach, a thread obtains exclusive access by acquiring a lock, which forces all other threads to wait until the end of its critical section.
- In the cooperative approach, a thread advertises *what it is trying to do*, which allows other threads to proceed by first *helping* it finish its work.

Helping is not so selfless: in reality, one thread is merely "helping" another thread get out of its way, so that access to some resource is no longer obstructed. In theory, helping provides a stronger guarantee of system progress—one that does not depend on fairness—and we explore this point

in §2.5.2. In practice, the progress guarantees must be weighed against the potential for extra memory coherence traffic. Sometimes it is better to wait.

One particular form of helping that works very nicely in practice is *elimination*. The idea is that certain operations on a data structure cancel each other out, and so if threads can somehow discover that they are trying to perform such operations concurrently, they can eliminate their work. For example, if one thread is trying to push a value onto a stack and another is trying to pop from the stack, the stack does not need to change at all! After all, if the operations are concurrent, they can be sequenced in either order, and a push followed by a pop is a no-op. This observation leads to "elimination backoff stacks," which consist of a Treiber stack together with a "side channel" array[73] that pushers and poppers use to find each other.[74] Operations are first attempted on the stack itself, but if the necessary **cas** fails, the thread flips a coin and either advertises its operation on the side channel, or else looks for an advertised operation that it can eliminate. In either case, the amount of time spent trying to find an elimination partner increases exponentially with each failed **cas**, just as in busywait-based backoff. Once the time is up, the thread cancels any advertised offers and retries the operation on the stack.

[73] The side channel is an array so that multiple eliminations can proceed in parallel. The indices at which offers are made or looked for follows the same exponential growth pattern as the backoff timing.

[74] Hendler *et al.* (2004), "A scalable lock-free stack algorithm"

Elimination backoff is effective because it spreads out contention: instead of many cores trying to gain exclusive access to a single cache block containing the stack head, cores compete for access to a set of blocks (including each element of the side channel array). Just as randomized exponential busywaiting tends to spread concurrent accesses out uniformly in *time*, randomized elimination tends to spread out contention uniformly in *space* (over the side channel array). The result is that each individual cache block is contended for by fewer threads on average, which greatly reduces the cost of coherence. Remarkably, the potential for parallelism *increases* as a result of highly-concurrent access, because of the increasing likelihood that parallel sets of pushers and poppers will discover each other.

2.4.6 *Synchronization and dual data structures*

With scalable concurrency, even the speed of waiting matters.

Consider implementing a lock intended to protect a "popular" resource with short-lived critical sections. On a parallel[75] machine, a good strategy is to use a *spin*lock, in which threads waiting to acquire the lock busywait ("spin") instead of actually blocking. Spinlocks make sense when the average critical section is much shorter than the average cost of a context switch.[76] We saw a very simple spinlock implementation in §2.3.2, in which the lock is just a boolean reference and acquisition works by repeated **cas** attempts:

[75] Without parallelism, busywaiting of *any* kind is useless: no concurrent thread could possibly change conditions while another "actively" waits.

[76] A context switch occurs when a core stops executing one thread and begins executing another, in this case because the execution of the first thread was "blocked," waiting for some condition to change.

```
acq(lock) = if cas(lock, false, true) then () else acq(lock)
```

While the implementation is semantically sound, it suffers from unfortunate interactions with cache coherence. To see why, imagine a situation in which

many threads are concurrently attempting to acquire the lock. As these waiting threads spin around the acquisition loop, they generate a massive amount of coherence traffic: each **cas** attempt requires gaining exclusive access to the cache block containing the lock, so the block continuously ping-pongs between the waiting cores. The traffic deluge eats into memory bandwidth that could otherwise be used by the lock-holding thread to get *actual* work done. By waiting so aggressively, threads delay the very thing they are waiting for.

The situation can be mildly improved by guarding the **cas** with a snapshot of the lock, adopting an approach more in the spirit of optimistic concurrency:

```
acq(lock) = if get(lock) then acq(lock)              // lock is not free; retry
            else if cas(lock, false, true) then ()   // lock appears free; race to claim it
            else acq(lock)                           // lost race to claim; retry
```

It is enlightening to think carefully through the cache implications of this new strategy. After the lock is acquired by a thread, its value will remain **true** until it is released. Consequently, the cache block holding it can be held concurrently by *all* the waiting threads in "shared" mode. When these threads spinwait on the lock, they will do so by accessing their own local cache—the ping-ponging has disappeared.

But what happens when the lock is released? First, the lock-holder must gain exclusive access to the lock's cache block. Afterwards, *all* of the waiting threads will read that block, requiring coherence traffic proportional to the number of waiters. All of the waiters will observe that the lock is now free, and so they will all attempt to **cas** it at once. Only one **cas** will win, of course, but nevertheless all of the waiting threads will get the cache block in exclusive mode for their **cas** attempt. Thus, while the revised locking strategy eliminates coherence traffic during the critical section, there is still a cascade of traffic once the lock is released. For popular resources and short critical sections, this traffic spells disaster.

The most important strategy for cutting down this traffic was introduced by Mellor-Crummey and Scott (1991), who suggested that waiting threads should place themselves in a (lock-free) queue. Instead of spinwaiting on the lock itself, each thread spinwaits on a *distinct* memory location associated with its entry in the queue. A thread releasing the lock can therefore signal a *single* waiting thread that the lock is available; there is no cascade of **cas** attempts (most of which will fail).

▸ WAITING ALSO PLAYS A ROLE IN OPTIMISTIC CONCURRENCY. Take Treiber's stack. Both push and tryPop are *total* operations: they can in principle succeed no matter what state the stack is in, and fail (and retry) only due to active interference from concurrent threads. A true pop operation, on the other hand, is *partial*: it is undefined when the stack is empty. Often this is taken to mean that the operation should wait until another thread changes

conditions such that the operation is defined, *e.g.,* by pushing an item onto the stack. Partial operations introduce considerable complexity, because *all* of the operations on the data structure must potentially signal waiting threads, depending on the changes being performed.

William N. Scherer, III and Scott (2004) introduced the concept of *dual data structures*, which contain both traditional data as well as its "dual," *reservations* for consuming a bit of data once it has arrived. The beauty of the approach is that both data and reservations can be added to a data structure following the usual methods of optimistic concurrency, which makes it possible to build scalable concurrent abstractions with both total and partial operations. In fact, as we will see in Chapter 8, even the scalable locks of Mellor-Crummey and Scott (1991) can be viewed as a kind of dual data structure in which acq is a partial operation.

2.5 THE RUDIMENTS OF SCALABLE CONCURRENCY: CORRECTNESS

When reasoning about "reactive systems" which participate in ongoing interaction with their environment, it is helpful to distinguish between two basic kinds of correctness properties:

- Safety properties, which say that "nothing bad happens." Semi-formally, a pure safety property is one for which any violation can be witnessed by a finite amount of interaction, *no* continuation of which will satisfy the property. A failure of safety requires only finite observation.
- Liveness properties, which say that "something good keeps happening." Semi-formally, a pure liveness property is one for which *every* finite amount of interaction has *some* continuation that will satisfy the property. A failure of liveness requires infinite observation.

Each step of internal computation a thread takes is considered to be an (uninformative) interaction. So, for example, the fact that a thread will send *some* message is a liveness property: if all we have observed so far is that the thread has computed internally for some finite number of clock ticks, we cannot say yet whether the thread will eventually return a result. We can only observe divergence by waiting for a message for an "infinite amount of time." On the other hand, if the thread sends the *wrong* value, it will do so after some finite amount of interaction, and no subsequent amount of interaction will erase that mistake.

In common temporal logics, every property can be expressed as a conjunction of a pure safety property with a pure liveness property.[77] Such a decomposition is useful because the two kinds of properties are best tackled with different tools (invariance and well-founded measures, respectively), and the proof of a liveness property often builds on already-proved safety properties.

Although we will be exclusively concerned with proving safety properties, it is helpful to understand both the safety and the liveness properties that are

[77] Alpern and Schneider (1985); Manolios and Trefler (2003)

commonly sought for scalable concurrent algorithms, since both influence algorithm design. We briefly discuss them next.

2.5.1 *Safety: linearizability*

From the outset, the safety of scalable concurrent algorithms has been characterized by a property called *linearizability*.[78] The property is intended to formalize the "atomic specification" of abstractions we discussed informally in §2.2.4. The idea is to view the behavior of a data structure abstractly, as a sequence of calls by and responses to some number of concurrent clients. In such a "history" of interaction, clients cannot issue a new request until the last request has returned, but multiple clients may have outstanding requests, which abstractly models the unpredictable timing of concurrent interaction. The goal of linearizability is to formalize the following principle:

[78] Herlihy and Wing (1990), "Linearizability: a correctness condition for concurrent objects"

> *Each method call should appear to take effect instantaneously at some moment between its invocation and response.*
>
> —Herlihy and Shavit (2008, chapter 3)

An implementation is "linearizable" if, for every history it generates, it is possible to produce a "matching atomic history" that obeys its specification. A matching history is a sequence of the same calls and responses in a possibly-different order, subject to a simple constraint: if a given response occurred prior to a given call in the original history, it must do so in the matching history as well. A history is atomic if every call is immediately followed by its response—meaning that the interactions were purely sequential. An atomic history can easily be validated against a traditional, sequential specification for an abstraction.

To illustrate these ideas concretely, consider a concurrent stack. Here are two very similar histories, only one of which linearizable:

Linearizable	Not linearizable
$\mathsf{call}_0(\mathsf{push}, 3)$	$\mathsf{call}_0(\mathsf{push}, 3)$
$\mathsf{call}_1(\mathsf{push}, 7)$	$\mathsf{call}_1(\mathsf{push}, 7)$
$\mathsf{call}_2(\mathsf{tryPop}, ())$	$\mathsf{resp}_0(\mathsf{push}, ())$
$\mathsf{resp}_0(\mathsf{push}, ())$	$\mathsf{call}_2(\mathsf{tryPop}, ())$
$\mathsf{call}_3(\mathsf{tryPop}, ())$	$\mathsf{call}_3(\mathsf{tryPop}, ())$
$\mathsf{resp}_1(\mathsf{push}, ())$	$\mathsf{resp}_1(\mathsf{push}, ())$
$\mathsf{resp}_2(\mathsf{tryPop}, \mathbf{none})$	$\mathsf{resp}_2(\mathsf{tryPop}, \mathbf{none})$
$\mathsf{resp}_3(\mathsf{tryPop}, \mathbf{some}(3))$	$\mathsf{resp}_3(\mathsf{tryPop}, \mathbf{some}(3))$

The atomic history matching the first history is as follows:

Atomic history	
$\mathsf{call}_2(\mathsf{tryPop}, ());$	$\mathsf{resp}_2(\mathsf{tryPop}, \mathbf{none})$
$\mathsf{call}_0(\mathsf{push}, 3);$	$\mathsf{resp}_0(\mathsf{push}, ())$
$\mathsf{call}_3(\mathsf{tryPop}, ());$	$\mathsf{resp}_3(\mathsf{tryPop}, \mathbf{some}(3))$
$\mathsf{call}_1(\mathsf{push}, 7);$	$\mathsf{resp}_1(\mathsf{push}, ())$

The second history, on the other hand, is not linearizable because the push by thread 0 is completed prior to the attempt to pop by thread 2—and so, given the circumstances, the latter call cannot lead to a response of **none**.

The intent of linearizability is to reduce concurrent specifications to sequential ones, and therefore to reduce concurrent reasoning to sequential reasoning. It has long served as the "gold standard" safety specification for concurrent data structures with method-atomic specifications. Nevertheless, we will argue in §3.3 and §3.4 that *contextual refinement* is the semantic property that clients want, rather than linearizability. Contextual refinement directly formalizes the idea that a client can link their program with a scalable concurrent algorithm for performance, but reason about it as if they were using a much simpler, coarse-grained algorithm.

2.5.2 *Liveness: nonblocking progress properties*

One downside of locking is that a delay of the thread holding a lock results in a further delay of all threads waiting for the lock. For example, a very unlucky thread might encounter a page fault within its critical section, with disastrous performance ramifications. Whether or not such a situation is likely to arise in practice, it is a problem that many scalable concurrent algorithms simply do not have: they are formally *non-blocking*.[79]

[79] Herlihy and Shavit (2008), "The Art of Multiprocessor Programming"

Non-blocking liveness properties are applied to scalable concurrent abstractions that provide some set of methods. They characterize progress guarantees for completing in-progress, concurrent method invocations.

The weakest property is *obstruction-freedom*, which says that at any time, if a single thread executing a method is allowed to proceed in isolation, it will eventually complete the method execution. With an obstruction-free abstraction, method invocations can only fail to make progress if another thread is *actively interferes*. In contrast, with a lock-based abstraction, an isolated thread executing a method may fail to make progress for the simple reason that it does not hold the lock, and will be unable to acquire the lock until some *other* thread makes progress. The idea behind obstruction-freedom is to model an *unbounded delay* of one thread as a *permanent failure*; obstruction-freedom then asks that other threads can continue making progress in such a case. It thereby precludes the use of locks.

A yet-stronger condition is the (confusingly named) *lock-freedom* property, which says that if *some* method is being executed, *some* method will complete execution—but not necessarily the same one! In other words, a method invocation can only fail to make progress if some other invocation is succeeding.[80] Lock-freedom neither assumes fairness[81] from the scheduler, nor guarantees fair method execution. It implies obstruction-freedom because if only one method is executing, that must be *the* method making progress.

[80] Consider the cases in which **cas** can fail in Treiber's stack, for example.

[81] *e.g.*, that a thread that is continuously ready to execute will eventually be executed.

Finally, *wait-freedom* simply guarantees that *every* method invocation will eventually finish in a finite number of steps, *even if concurrent method invocations arrive continuously*.

Altogether, we have:

$$\text{wait-free} \Rightarrow \text{lock-free} \Rightarrow \text{obstruction-free} = \text{nonblocking}$$

Most scalable concurrent data structures are designed to be lock-free, and many rely on helping (§2.4.5) to achieve this goal. Wait-freedom is considered prohibitively expensive. Obstruction-freedom was introduced to characterize progress for some software transactional memory implementations.[82] Just as complexity analysis does not tell the whole story of real-world performance for sequential algorithms, non-blocking liveness properties are only part of the story for scalable concurrency—they can sometimes profitably be traded in exchange for better cache behavior.[83]

[82] Herlihy, Luchangco, and Moir (2003), "Obstruction-free synchronization: double-ended queues as an example"

[83] Hendler *et al.* (2010), "Flat combining and the synchronization-parallelism tradeoff"

Part II

UNDERSTANDING SCALABLE CONCURRENCY

3

A calculus for scalable concurrency

▸ Synopsis This chapter formalizes a calculus, $F^{\mu}_{\mathbf{cas}}$, which is a variant of the polymorphic lambda calculus extended with mutable references, **cas** and **fork**—the essential features needed to model scalable concurrent algorithms written in a high-level language. The chapter defines and discusses a memory consistency model (§3.2), refinement (§3.3), and atomicity (§3.4), in particular contrasting linearizability and refinement (§3.4.2). Some auxiliary technical details appear in Appendix A.

"Language is the armory of the human mind, and at once contains the trophies of its past and the weapons of its future conquests."

—Samuel Taylor Coleridge

3.1 THE CALCULUS

Any formal study of concurrent algorithms must begin by formalizing a language in which to write them. In the past, scalable algorithms have been formalized in low-level, C-like settings, usually untyped and without abstraction mechanisms or a way to define procedures. But in practice, even algorithms written in low-level languages use hiding to enforce local protocols on their data. Libraries like JUC and TBB are written in typed, object-oriented languages and use polymorphism to provide generic data structures. JUC in particular provides support for futures and fork/join computations that is intrinsically higher-order, relying on the ability to pass objects with unknown methods as arguments and on inheritance, respectively. There are also generic constructions for producing concurrent data structures from sequential ones—*e.g.,* Herlihy's universal construction[1] and the more practical "flat combining" construction[2]—which are best-expressed as higher-order functions (or even as SML-style functors).

We therefore study scalable concurrency within a calculus we call $F^{\mu}_{\mathbf{cas}}$, which is a variant of the polymorphic lambda calculus (System F), extended with tagged sums, general mutable references (higher-order state), equi-recursive types, **cas**, and **fork**. The result is a very compact calculus with that can faithfully model JUC-style algorithms, including those that use polymorphic, recursively-linked data structures, hiding, and higher-order features. As we will see in Chapters 4 and 5, these features also suffice to formalize the interaction between linguistic hiding mechanisms and concurrent data structures, or, more succinctly, to study *concurrent data abstraction*.

[1] Herlihy (1991), "Wait-free synchronization"

[2] Hendler *et al.* (2010), "Flat combining and the synchronization-parallelism tradeoff"

Comparable types	σ	::=	**unit**	Unit (*i.e.*, nullary tuple)
		|	**bool**	Boolean
		|	**nat**	Natural number
		|	$\tau + \tau$	Tagged union
		|	$\mathbf{ref}(\overline{\tau})$	Mutable tuple reference
		|	$\mathbf{ref}_?(\overline{\tau})$	Optional reference
		|	$\mu\alpha.\sigma$	Recursive comparable type
Types	τ	::=	σ	Comparable type
		|	α	Type variable
		|	$\tau \times \tau$	Immutable pair type
		|	$\mu\alpha.\tau$	Recursive type
		|	$\forall\alpha.\tau$	Polymorphic type
		|	$\tau \to \tau$	Function type
Values	v	::=	$()$	Unit value
		|	**true**	Boolean value
		|	**false**	Boolean value
		|	n	Number value
		|	(v, v)	Pair value
		|	**rec** $f(x).e$	Recursive function
		|	$\Lambda.e$	Type abstraction
		|	ℓ	Heap location
		|	**null**	Null optional reference
		|	x	Variable
Expressions	e	::=	v	Value
		|	**if** e **then** e **else** e	Conditional
		|	$e + e$	Addition
		|	(e, e)	Pair introduction
		|	**let** $(x, y) = e$ **in** e	Pair elimination
		|	$e\ e$	Function application
		|	$e\ \_$	Type application
		|	$\mathbf{case}(e, \mathbf{null} \Rightarrow e, x \Rightarrow e)$	Optional reference elimination
		|	$\mathbf{inj}_i\ e$	Tagged union injection
		|	$\mathbf{case}(e, \mathbf{inj}_1\ x \Rightarrow e, \mathbf{inj}_2\ y \Rightarrow e)$	Tagged union elimination
		|	**new** $\overline{e}$	Mutable tuple allocation
		|	$\mathbf{get}(e[i])$	Mutable tuple dereference
		|	$e[i] := e$	Mutable tuple assignment
		|	$\mathbf{cas}(e[i], e, e)$	Mutable tuple atomic update
		|	**fork** e	Process forking

Figure 3.1: $F^{\mu}_{\mathbf{cas}}$ syntax

3.1.1 *Syntax*

Figure 3.1 gives the syntax of $F^{\mu}_{\textbf{cas}}$. The language is essentially standard, but there are subtleties related to **cas**, and a few unusual aspects that help keep our treatment of algorithms concise. We discuss both below.

▸ MODELING COMPARE-AND-SET As we discussed in §2.3.2, **cas** is a hardware-level operation: it operates on a memory cell containing a single word-sized value, comparing that value for physical equality and updating it as appropriate. In practice, this word value is usually a pointer to some other object.[3] In a low-level language, none of this is surprising or problematic. In a high-level language, however, pointer comparison is not always appropriate to expose, since it reveals the allocation strategy of the compiler (and otherwise may break programmer abstractions). On the other hand, languages like Java (and, of course, C++) already expose pointer comparisons, and these are the languages used to build JUC and TBB.

In order to faithfully model JUC-style algorithms without strongly committing the linguistic model to pointer comparison, we introduce a distinct category of *comparable types* σ, as opposed to *general types* τ.[4] The **cas** operator can only be applied at comparable type, so the distinction allows a language designer to choose which physical representations to reveal. For simplicity, $F^{\mu}_{\textbf{cas}}$ treats only base types (*i.e.,* unit values, natural numbers, and booleans), locations (*i.e.,* reference types) and tagged unions (*i.e.,* sum types) as comparable. The inclusion of tagged unions is justified by the fact that the operational semantics explicitly heap-allocates them; see §3.1.3.[5] Comparable types also include recursive type definitions over other comparable types, which is a bit subtle: the type variable introduced by the recursion can only appear within a general type τ, which in turn is protected by a layer of indirection through the heap (either in a tagged union or reference). The idea is just that there needs to be some physical source of identity to actually compare, and in the end we use erased *equi*-recursive types, which have no physical, run-time existence.

▸ KEEPING THINGS CONCISE We make several concessions in the calculus that will pay dividends later on, when formally reasoning about algorithms:

- Following Ahmed (2006), terms are not annotated with types, but polymorphism is nevertheless introduced and eliminated by explicit type abstraction ($\Lambda.e$) and application ($e\ \_$).
- References are to *mutable tuples* $\textbf{ref}(\overline{\tau})$ (as opposed to *immutable pairs* $\tau_1 \times \tau_2$), useful for constructing objects with many mutable fields.[6] The term $\mathsf{get}(e[i])$ reads and projects the i-th component from a reference e, while $e[i] := e'$ assigns a new value to that component. When e is a single-cell reference, we will usually write $\mathsf{get}(e)$ instead of $\mathsf{get}(e[1])$ and $e := e'$ instead of $e[1] := e'$.

[3] See §2.4.3.

[4] The distinction could also be phrased in terms of a *kinding* system: one would have a single syntactic category of types, but assign types distinct kinds of either comparable $\mathbf{C}$ or general $\star$, with a subkinding relation $\mathbf{C} \preceq \star$. We have opted for the lighter-weight syntactic distinction, and model subkinding by embedding comparable types into the syntax of general types.

[5] An alternative design would be to introduce an explicit type constructor of immutable "boxes," which simply wrap a value of an arbitrary type with an explicitly-comparable identity. Boxed values would then be the only comparable type. A smart compiler would be able to treat the box as a no-op most of the time.

[6] The overbar notation represents a vector.

- The type $\mathsf{ref}_?(\overline{\tau})$ of "option references" provides an *untagged union* of null values and reference types.[7] Because reading and writing operations work on references, and not option references (which must be separately eliminated by cases), there are no null-pointer errors.

[7] Contrast this with a tagged option type, which would require an extra indirection through the heap.

The net effect is flatter, less verbose type constructions with fewer layers of indirection.

▸ DERIVED FORMS We will freely use standard derived forms, *e.g.*,

$$\begin{array}{rcll} \lambda x.e & \triangleq & \mathsf{rec}\ z(x).e & z \text{ fresh} \\ \mathsf{let}\ x = e\ \mathsf{in}\ e' & \triangleq & (\lambda z.e')\ e & z \text{ fresh} \\ e; e' & \triangleq & \mathsf{let}\ z = e\ \mathsf{in}\ e' & z \text{ fresh} \end{array}$$

as well as nested pattern matching and recursive function definitions in let-bindings.

3.1.2 *Typing*

The type system for F^{μ}_{cas} is quite straightforward to formalize. First, there are typing contexts:

$$\begin{array}{lrcl} \text{TYPE VARIABLE CONTEXTS} & \Delta & ::= & \cdot \mid \Delta, \alpha \\ \text{TERM VARIABLE CONTEXTS} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau \\ \text{COMBINED CONTEXTS} & \Omega & ::= & \Delta; \Gamma \end{array}$$

The typing judgment in Figure 3.2 uses combined typing contexts Ω to reduce clutter; comma-adjoined type or term variables are simply joined to the appropriate context within. The calculus additionally requires that all recursive types $\mu\alpha.\tau$ be *productive*, meaning that all free occurrences of α in τ must appear under a non-μ type constructor.

3.1.3 *Operational semantics*

The operational semantics of F^{μ}_{cas} is formulated in evaluation-context style. To define the primitive reductions, we first formalize heaps:

$$\begin{array}{lrcll} \text{HEAP-STORED VALUES} & u & \in & \text{HeapVal} & ::= (\overline{v}) \mid \mathsf{inj}_i\ v \\ \text{HEAPS} & h & \in & \text{Heap} & \triangleq \text{Loc} \rightharpoonup \text{HeapVal} \end{array}$$

Here we clearly see that tagged sums and mutable tuples are heap-allocated: their canonical forms are *heap-stored values* u rather than *values* v. Heaps are just partial functions from locations ℓ to heap-stored values. Primitive reductions work over pairs of heaps and expressions, and their definition (Figure 3.3) is entirely straightforward.

▸ WELL-TYPED TERMS $\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\Omega \vdash () : \mathsf{unit} \qquad \Omega \vdash \mathsf{true} : \mathsf{bool} \qquad \Omega \vdash \mathsf{false} : \mathsf{bool} \qquad \Omega \vdash n : \mathsf{nat} \qquad \Omega, x : \tau \vdash x : \tau$$

$$\frac{\Omega \vdash e : \mathsf{bool} \qquad \Omega \vdash e_i : \tau}{\Omega \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau} \qquad \frac{\Omega \vdash e_1 : \tau_1 \qquad \Omega \vdash e_2 : \tau_2}{\Omega \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Omega \vdash e : \tau_1 \times \tau_2 \qquad \Omega, x : \tau_1, y : \tau_2 \vdash e' : \tau}{\Omega \vdash \mathsf{let}\ (x, y) = e\ \mathsf{in}\ e' : \tau}$$

$$\frac{\Omega, f : \tau' \to \tau, x : \tau' \vdash e : \tau}{\Omega \vdash \mathsf{rec}\ f(x).e : \tau' \to \tau} \qquad \frac{\Omega \vdash e : \tau' \to \tau \qquad \Omega \vdash e' : \tau'}{\Omega \vdash e\ e' : \tau} \qquad \Omega \vdash \mathsf{null} : \mathsf{ref}_?(\overline{\tau}) \qquad \frac{\Omega \vdash e : \mathsf{ref}(\overline{\tau})}{\Omega \vdash e : \mathsf{ref}_?(\overline{\tau})}$$

$$\frac{\Omega \vdash e : \mathsf{ref}_?(\overline{\tau}) \qquad \Omega \vdash e_1 : \tau \qquad \Omega, x : \mathsf{ref}(\overline{\tau}) \vdash e_2 : \tau}{\Omega \vdash \mathsf{case}(e, \mathsf{null} \Rightarrow e_1, x \Rightarrow e_2) : \tau} \qquad \frac{\Omega \vdash e : \tau_i}{\Omega \vdash \mathsf{inj}_i\ e : \tau_1 + \tau_2}$$

$$\frac{\Omega \vdash e : \tau_1 + \tau_2 \qquad \Omega, x : \tau_i \vdash e_i : \tau}{\Omega \vdash \mathsf{case}(e, \mathsf{inj}_1\ x \Rightarrow e_1, \mathsf{inj}_2\ x \Rightarrow e_2) : \tau} \qquad \frac{\Omega, \alpha \vdash e : \tau}{\Omega \vdash \Lambda.e : \forall \alpha.\tau} \qquad \frac{\Omega \vdash e : \forall \alpha.\tau}{\Omega \vdash e\ \_ : \tau[\tau'/\alpha]} \qquad \frac{\Omega \vdash e_i : \tau_i}{\Omega \vdash \mathsf{new}\ (\overline{e}) : \mathsf{ref}(\overline{\tau})}$$

$$\frac{\Omega \vdash e : \mathsf{ref}(\overline{\tau})}{\Omega \vdash \mathsf{get}(e[i]) : \tau_i} \qquad \frac{\Omega \vdash e : \mathsf{ref}(\overline{\tau}) \qquad \Omega \vdash e' : \tau_i}{\Omega \vdash e[i] := e' : \mathsf{unit}} \qquad \frac{\Omega \vdash e : \mathsf{ref}(\overline{\tau}) \qquad \tau_i = \sigma \qquad \Omega \vdash e_o : \sigma \qquad \Omega \vdash e_n : \sigma}{\Omega \vdash \mathsf{cas}(e[i], e_o, e_n) : \mathsf{bool}}$$

$$\frac{\Omega \vdash e : \mathsf{unit}}{\Omega \vdash \mathsf{fork}\ e : \mathsf{unit}} \qquad \frac{\Omega \vdash e : \mu\alpha.\tau}{\Omega \vdash e : \tau[\mu\alpha.\tau/\alpha]} \qquad \frac{\Omega \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Omega \vdash e : \mu\alpha.\tau}$$

Figure 3.2: F^{μ}_{cas} typing

▸ PRIMITIVE REDUCTIONS $\boxed{h; e \hookrightarrow h'; e'}$

$$\begin{array}{rcll}
h; n + m & \hookrightarrow & h; k & \text{when } k = n + m \\
h; \mathsf{get}(\ell[i]) & \hookrightarrow & h; v_i & \text{when } h(\ell) = (\overline{v}) \\
h; \ell[i] := v & \hookrightarrow & h[\ell[i] = v]; () & \text{when } \ell \in \mathrm{dom}(h) \\
h; \mathsf{cas}(\ell[i], v_o, v_n) & \hookrightarrow & h[\ell[i] = v_n]; \mathsf{true} & \text{when } h(\ell)[i] = v_o \\
h; \mathsf{cas}(\ell[i], v_o, v_n) & \hookrightarrow & h; \mathsf{false} & \text{when } h(\ell)[i] \neq v_o \\
h; \mathsf{case}(\ell, \mathsf{inj}_1\ x \Rightarrow e_1, \mathsf{inj}_2\ x \Rightarrow e_2) & \hookrightarrow & h; e_i[v/x] & \text{when } h(\ell) = \mathsf{inj}_i\ v
\end{array}$$

$$\begin{array}{rcl}
h; \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 & \hookrightarrow & h; e_1 \\
h; \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 & \hookrightarrow & h; e_1 \\
h; \mathsf{case}(\mathsf{null}, \mathsf{null} \Rightarrow e_1, x \Rightarrow e_2) & \hookrightarrow & h; e_1 \\
h; \mathsf{case}(\ell, \mathsf{null} \Rightarrow e_1, x \Rightarrow e_2) & \hookrightarrow & h; e_2[\ell/x] \\
h; \mathsf{let}\ (x, y) = (v_1, v_2)\ \mathsf{in}\ e & \hookrightarrow & h; e[v_1/x, v_2/y] \\
h; (\mathsf{rec}\ f(x).e)\ v & \hookrightarrow & h; e[\mathsf{rec}\ f(x).e/f, v/x] \\
h; \mathsf{inj}_i\ v & \hookrightarrow & h \uplus [\ell \mapsto \mathsf{inj}_i\ v]; \ell \\
h; (\Lambda.e)\ _ & \hookrightarrow & h; e \\
h; \mathsf{new}\ (\overline{v}) & \hookrightarrow & h \uplus [\ell \mapsto (\overline{v})]; \ell
\end{array}$$

Figure 3.3: F^{μ}_{cas} primitive reductions

To formalize general reduction, we first define the *evaluation contexts* K, which specify a left-to-right, call-by-value evaluation strategy:

$$\begin{array}{rcl} K & ::= & [\,] \mid \textbf{if } K \textbf{ then } e \textbf{ else } e \mid K + e \mid v + K \mid (K, e) \mid (v, K) \\ & \mid & \textbf{let } (x, y) = K \textbf{ in } e \mid K\ e \mid v\ K \mid \textbf{inj}_i\ K \mid K\ \_ \\ & \mid & \textbf{case}(K, \textbf{inj}_1\ x \Rightarrow e, \textbf{inj}_2\ x \Rightarrow e) \mid \textbf{case}(K, \textbf{null} \Rightarrow e, x \Rightarrow e) \\ & \mid & \textbf{new } (\overline{v}, K, \overline{e}) \mid \textbf{get}(K[i]) \mid K[i] := e \mid v[i] := K \\ & \mid & \textbf{cas}(K[i], e, e) \mid \textbf{cas}(v[i], K, e) \mid \textbf{cas}(v[i], v, K) \end{array}$$

A *program configuration* (or just "configuration") ς is a pair of a heap and a *thread pool* T:

$$\begin{array}{rcl} T & \in & \text{ThreadPool} \triangleq \mathbb{N} \xrightarrow{\text{fin}} \text{Expression} \\ \varsigma & ::= & h; T \end{array}$$

The thread pool assigns each thread a unique identifier in $\mathbb{N}$. For simplicity, the language does not allow a thread to observe its identifier; while adding this feature (along with the ability to wait for termination of a given thread) to the language is easy, doing so would complicate the model in Chapter 5 and is not necessary for the algorithms we study.

Finally, reduction on configurations simply allows a given thread to execute a step (possibly within an evaluation context) and defines the semantics of **fork**:

▸ General reduction $\boxed{h; T \to h'; T'}$

$$\frac{h; e \hookrightarrow h'; e'}{h; T \uplus [i \mapsto K[e]] \to h'; T \uplus [i \mapsto K[e']]}$$

$$h; T \uplus [i \mapsto K[\textbf{fork } e]] \to h; T \uplus [i \mapsto K[()]] \uplus [j \mapsto e]$$

3.2 THE MEMORY CONSISTENCY MODEL

The semantics we have just defined provides a model of shared-memory concurrency that, until recently, was uncontroversial: the heap h in $F^{\mu}_{\textbf{cas}}$ provides a global, sequentially consistent view of memory to all threads. That is, the semantics interleaves thread execution, and each step a thread takes operates on that single, global heap. Each update a thread performs is immediately visible to all other threads through the global heap.

Unfortunately, such an account of shared memory is wildly unrealistic in general, for reasons that span the entire tower of abstractions that make up modern computer systems:

- Optimizing compilers liberally reorder instructions in ways that are unobservable for sequential code, but very much observable for concurrent code.

- Writes performed by a CPU are not sent directly to main memory, which would be extremely expensive; instead, they update the most local cache.[8] Even with cache-coherent architectures, it is possible for activity on different memory locations to appear to happen in different orders to different CPUs.

- CPUs also reorder instructions, for the same reasons and with the same caveats as compilers.

[8] See §2.3.

These problems all stem from the implementation of "memory" as an abstraction for sequential code—an abstraction that begins leaking its implementation details in the concurrent case.

Because sequential performance is paramount, the solution is to weaken the abstraction rather than to change the implementation. The result is the (ongoing) study and formulation of *memory consistency models* at both the hardware and language levels.

Here we are concerned only with language-level memory models, and the key question is: how does the sequentially-consistent model of $F^{\mu}_{\mathbf{cas}}$ limit the applicability of our results in practice? More concretely, if we prove the correctness of an algorithm in $F^{\mu}_{\mathbf{cas}}$, what does that tell us about a transliteration of the algorithm into, say, Java or Scala?

While no modern memory model guarantees sequential consistency in general, the primary goal of most memory models is to delineate a class of "well-synchronized" programs[9] for which memory *is* guaranteed to be sequentially-consistent (while still leaving plenty of room for optimization). The idea is that certain language primitives are considered to be "synchronized,"[10] meaning amongst other things that they act as explicit barriers to instruction reordering and force cache flushes. In other words, synchronized operations effectively "turn off" the problematic optimizations to the memory hierarchy, and provide walls over which the optimizations cannot cross. Lock acquisition/release and **cas** are examples of such operations. Moreover, many languages provide a way to mark a particular reference as sequentially-consistent,[11] meaning that every read and write to the reference acts as a synchronized operation.

[9] Proper synchronization is often called "data race freedom".

[10] For simplicity, this discussion glosses over the various distinctions that some memory models make between different flavors of synchronization or barrier operations.

[11] In Java, this is provided by the **volatile** keyword.

We have glossed over many details, but the upshot is clear enough: the memory model of $F^{\mu}_{\mathbf{cas}}$ *is* realistic if we consider *all* references as being implicitly marked as sequentially-consistent. In particular, if a transliteration of an algorithm into a real language explicitly marks its references as volatile, the algorithm will behave as it would in $F^{\mu}_{\mathbf{cas}}$. This strategy is reasonable for a majority of scalable concurrent algorithms, which in practice use such marked references anyway. But it does mean that $F^{\mu}_{\mathbf{cas}}$ cannot be used to study more subtle algorithms, such as RCU in Linux,[12] that use references with weaker consistency. It also means that $F^{\mu}_{\mathbf{cas}}$ cannot specify the "happens-before"[13] implications of concurrency abstractions that make up part of the API for libraries like JUC. Finally, it means that $F^{\mu}_{\mathbf{cas}}$ is inappropriate for studying reference-heavy sequential code, which would run incredibly

[12] McKenney and Slingwine (1998), "Read-copy update: Using execution history to solve concurrency problems"

[13] The "happens-before" relation is a key aspect of many memory models.

slowly—though it should be noted that in a functional language such code is relatively uncommon.

3.3 CONTEXTUAL REFINEMENT

"Type structure is a syntactic discipline for maintaining levels of abstraction."

—John C. Reynolds, "Types, abstraction and parametric polymorphism"

The essence of abstraction is hiding: abstraction is only possible when some details are not meaningful and need not be visible. Or, put differently, an abstraction barrier delineates which details are visible at which "level of abstraction." When we speak of *specification* and *implementation*, we usually think of these as characterizing, respectively, *what* a client can observe from outside an abstraction barrier and *how* those observations are generated by code within the barrier. Clients thus only reason about what an abstraction does (its "spec"), not how (its implementation).

Specifications come in many forms, some of which are tied to a particular logical formalism. But one particularly simple, logic-independent method of specification is a so-called *reference implementation*, which avoids saying "what" an abstraction should do by instead saying "how" to implement a very simple version of the abstraction that exhibits all the permissible observable behavior. This form of specification has many downsides—it is often more concrete than necessary, and it is very easy to *over*specify—but it is well-suited for libraries where it is difficult to formulate a sufficiently general "principal specification," and where there is a large gap in complexity between the real and the reference implementations. Clients are then free to introduce further abstraction, *e.g.,* by showing that the reference implementation in turn satisfies some other, more abstract specification, and then reasoning on the basis of *that* specification.

"Don't ask what it means, but rather how it is used."

—Ludwig Wittgenstein

This section defines the standard notion of *contextual refinement* (hereafter: refinement), which we use to formalize specification via reference implementation. We discuss our particular use of reference implementations in §3.4.

▸ REFINEMENT CAPTURES OBSERVABLE BEHAVIOR VIA CLIENT INTERACTION. Suppose e_I is a library, and e_S is a reference implementation for it. If no client can tell that it is linked against e_I rather than e_S—that is, if every "observable behavior" of e_I can be reproduced by e_S—then indeed e_I meets its spec e_S, *i.e.,* e_I refines e_S. Conversely, if e_I behaves in a way that is *meaningfully* different from its spec, it should be possible to find a client that can tell the difference.[14]

We formalize the notion of an unknown (but well-typed!) client as a context[15] C. Contexts are classified by a standard typing judgment

$$C : (\Omega, \tau) \rightsquigarrow (\Omega', \tau')$$

such that whenever $\Omega \vdash e : \tau$, we have $\Omega' \vdash C[e] : \tau'$. The notation $C[e]$ indicates plugging the expression e into the hole of context C, yielding a new expression.

[14] This is, in fact, a tautology: a difference is "meaningful" precisely when it is observable by a client.

[15] A context is an expression with a "hole" into which another expression can be placed.

Then, if $\Omega \vdash e_I : \tau$ and $\Omega \vdash e_S : \tau$, we say e_I *contextually refines* e_S, written $\Omega \vDash e_I \preceq e_S : \tau$, if:

$$\begin{array}{l} \text{for every } i, j \text{ and } C : (\Omega, \tau) \rightsquigarrow (\varnothing, \mathbf{nat}) \text{ we have} \\ \quad \forall n. \forall T_I. \quad \varnothing; [i \mapsto C[e_I]] \to^* h_I; [i \mapsto n] \uplus T_I \\ \implies \exists T_S. \quad \varnothing; [j \mapsto C[e_S]] \to^* h_S; [j \mapsto n] \uplus T_S \end{array}$$

Thus an "observable difference" between two terms is any distinction that *some* client can transform into a difference in the natural number its main thread returns. Refinement is asymmetric, since the spec e_S sets an *upper bound* on the behaviors[16] of the implementation e_I. It is also a *congruence*, since any composition of implementations refines the corresponding composition of specs.

[16] The plural here is intended: $F^{\mu}_{\mathbf{cas}}$ has several sources of nondeterminism, including memory and thread allocation and thread interleaving. The spec should exhibit the maximum observable nondeterminism.

Refinement hinges on the power of the context C: what can it observe, and with what can it interfere? The more powerful the context, the less freedom we have in implementing a specification:

- Concurrency changes the *when*, but not the *what*, of observation. For example, in a first-order language, a sequential context can only interact with the state before or after running the program fragment in its hole. A concurrent context might do so at any time. But the allowed interactions are the same in either case.
- Abstraction mechanisms, on the other hand, determine *what* but not *when* observations are allowed. For example, a context can only observe a function by applying it, concurrently or sequentially. The context cannot tell directly whether the function is in fact a *closure* containing some internal mutable reference; it can only observe which outputs are generated from which inputs over time. The same goes for abstract data types, which deny the context access to their representation.

The scalable concurrent algorithms we will study in the next several chapters rely on abstraction to limit the power of their (possibly concurrent) clients: essentially all of the interesting action is hidden away in their internal state.

3.4 OBSERVABLE ATOMICITY

Most of the scalable algorithms we will study within $F^{\mu}_{\mathbf{cas}}$ are data structures whose operations are intended to "take effect atomically," even though their implementation is in fact highly concurrent.[17] We are now in a position to formalize this intent: a data structure is *observably atomic* if it refines a *canonical atomic spec*, *i.e.,* a spec of the form

[17] See §2.2.4 and §2.4.

$$\mathbf{let}\ \overline{x = e}\ \mathbf{in}\ \mathsf{mkAtomic}(e_1, \ldots, e_n)$$

where the overbar notation represents a (possibly empty) list, and where

$$\begin{array}{rcl}
\mathsf{acq} & \triangleq & \mathbf{rec}\ f(x).\ \mathbf{if}\ \mathbf{cas}(x, \mathbf{false}, \mathbf{true})\ \mathbf{then}\ ()\ \mathbf{else}\ f(x) \\
\mathsf{rel} & \triangleq & \lambda x.\ x := \mathbf{false} \\
\mathsf{withLock}(\mathsf{lock}, e) & \triangleq & \lambda x.\ \mathsf{acq}(\mathsf{lock});\ \mathbf{let}\ r = e(x)\ \mathbf{in}\ \mathsf{rel}(\mathsf{lock});\ r \\
\mathsf{mkAtomic}(e_1, \ldots, e_n) & \triangleq & \mathbf{let}\ \mathsf{lock} = \mathbf{new}\ (\mathbf{false})\ \mathbf{in} \\
 & & (\mathsf{withLock}(\mathsf{lock}, e_1),\ \ldots,\ \mathsf{withLock}(\mathsf{lock}, e_n))
\end{array}$$

The idea is that a "canonical atomic spec" is just a data structure protected by a global lock, for which mutual exclusion between method executions (and hence their atomicity) is trivially assured. Since global locking is the simplest way to achieve atomicity,[18] such specs are reasonable "reference implementations." More flexible definitions are certainly possible, but this simple one suffices for our purposes.

[18] See the discussion at the beginning of §2.4.

To see how this definition plays out concretely, we turn back to concurrent counters. Recall the simple (but surprisingly scalable) optimistic counter from §2.4.2:

$$\begin{array}{rcl}
\mathsf{casCnt} & = & \mathbf{let}\ \ r = \mathbf{new}\ 0 \\
 & & \quad \mathsf{inc} = \lambda().\ \mathbf{let}\ n = \mathbf{get}\ r \\
 & & \qquad\qquad \mathbf{in\ if}\ \mathbf{cas}(r, n, n+1)\ \mathbf{then}\ ()\ \mathbf{else}\ \mathsf{inc}() \\
 & & \quad \mathsf{read} = \lambda().\ \mathbf{get}\ r \\
 & & \mathbf{in}\ (\mathsf{inc}, \mathsf{read})
\end{array}$$

A canonical atomic spec for the counter is as follows:

$$\mathsf{atomCnt} = \mathbf{let}\ r = \mathbf{new}\ 0\ \mathbf{in}\ \mathsf{mkAtomic}(\lambda().\ r := \mathbf{get}\ r + 1,\ \lambda().\ \mathbf{get}\ r)$$

And indeed, we will show as a warmup example (§6.2) that

$$\cdot \vDash \mathsf{casCnt} \le \mathsf{atomCnt} : (\mathbf{unit} \to \mathbf{unit}) \times (\mathbf{unit} \to \mathbf{nat})$$

which means that a client can safely link against casCnt while reasoning as if it were linked against atomCnt. Admittedly, the "simplification" is rather underwhelming in this case, but for even slightly more sophisticated algorithms, the canonical atomic spec is much easier to reason about than the implementation.

Although the definition of the locks used for mkAtomic internally use **cas**, in practice we reason about it more abstractly, *e.g.,* by using very simple Hoare triples that summarize their effect.[19]

[19] This is, in fact, an example of the "further abstraction" that a client of a reference implementation can employ (§3.3).

3.4.1 *The problem with atomic blocks*

Despite the fact that we ultimately reason about global locks in an abstract way, it is still a bit unsatisfying that canonical atomic specs need to talk about locks at all. Why not instead add an **atomic** keyword (to be used by specifications only) with a single-step operational semantics, *e.g.,*

$$\frac{h; e \hookrightarrow^* h'; v}{h; K[\mathbf{atomic}\ e] \hookrightarrow h'; K[v]}$$

Although canonical atomic specs would still be written in essentially the same way (trading **atomic** for mkAtomic), their interpretation would be more trivially atomic. In fact, such "atomic blocks" provide *strong atomicity*, meaning that they appear atomic even if they access memory that is also access outside of any atomic block. The mkAtomic sugar, by contrast, supports only *weak atomicity*: the methods within it only execute atomically with respect to each other, not with respect to arbitrary other code.[20]

Unfortunately, the semantic strength of such an **atomic** keyword is also its downfall: it empowers contexts to make unrealistic observations, and in particular to observe the use of cooperation in scalable concurrent algorithms. To see why, consider an elimination stack (§2.4.5)[21] that provides push and tryPop methods, together with the obvious canonical atomic spec for a stack. In F^{μ}_{cas}, the elimination stack refines its spec, as one would expect. But if we added an **atomic** keyword, we could write a client like the following:

```
fork push(0); fork push(1);
atomic { push(2); tryPop() }
```

When linked with the stack specification, this client's main thread *always* returns **some**(2), because its definition of tryPop always pops off the top item of the stack, if one is available. But when linked with the elimination stack implementation, this client could return **some**(0), **some**(1), or **some**(2)! After all, the elimination-based tryPop does *not* always look at the top of the stack—it can instead be eliminated against concurrent push operations. Since the **fork**ed push operations are concurrent with the embedded tryPop operation, they may therefore be eliminated against it.

In short, **atomic** is too powerful to be allowed as a program context.[22]

While it is possible to integrate **atomic** in a more restricted (and hence less problematic) way, doing so is tricky in a higher-order language like F^{μ}_{cas}. We have investigated the use of a type-and-effect system like the one studied by Moore and Grossman (2008): types with a "not safe for atomic" effect classify terms that may *use* **atomic** but not be placed *within* other **atomic** blocks. In the end, though, the added complexity of a type-and-effect system together with (surmountable) step-indexing problems[23] convinced us to stick with the simpler lock-based treatment of atomicity.

3.4.2 *Refinement versus linearizability*

We have proposed that the correctness[24] of a scalable data structure should be expressed by refinement of a canonical atomic spec, but nearly all of the existing literature instead takes linearizability (§2.5.1) as the key safety property. We close out the chapter by arguing in favor of the refinement methodology.

At first blush, refinement and linearizability look rather different:

- While refinement has a fairly standard *definition* across languages, the *meaning* of the definition is quite language-dependent. The previous

[20] See Blundell *et al.* (2006) for a more detailed discussion of weak and strong atomicity.

[21] In particular, we consider a slight variant of the elimination stack that may try elimination *before* trying its operation on the actual stack.

[22] This observation should serve as a warning for attempts to integrate STM (which can provide a strong atomic facility) with scalable concurrent data structures.

[23] See Birkedal *et al.* (2012) for a step-indexed treatment of **atomic**.

[24] In terms of safety §2.5.

section gives a perfect example: by adding **atomic** to the language, we would drastically change the refinement relationships, invalidating some refinements that held in the absence of **atomic**.

- Linearizability, on the other hand, is defined in terms of quite abstract "histories," seemingly without reference to any particular language.

But to actually prove linearizability for specific examples, or to benefit formally from it as a client, *some* connection is needed to a language. In particular, there must be some (language-dependent) way to extract the possible histories of a given concurrent data structure—giving a kind of "history semantics" for the language. Once a means of obtaining histories is defined, it becomes possible to formally compare linearizability with refinement.

Filipović *et al.* (2010) study the relationship for a particular pair of first-order concurrent languages. In both cases, linearizability turns out to be sound for refinement, and in one case it is also complete.[25] But such a study has not been carried out for a higher-order polymorphic language like $F^{\mu}_{\mathbf{cas}}$, and it is not immediately clear how to generate or compare histories for such a language. In particular, the arguments and return values of functions may include other functions (themselves subject to the question of linearizability), or abstract types, for which a direct syntactic comparison is probably too strong.

[25] The difference hinges on whether clients can communicate through a side-channel; linearizability is complete in the case that they can.

The key message of Filipović *et al.* (2010) is that refinement is the property that clients of a data structure desire:

> *Programmers expect that the behavior of their program does not change whether they use experts' data structures or less-optimized but obviously-correct data structures.*
>
> —Ivana Filipović *et al.*, "Abstraction for Concurrent Objects"

and that, consequently, linearizability is only of interest to clients insofar as it implies refinement.

▸ If linearizability is a proof technique for refinement, its soundness is a kind of "context lemma"[26] saying the observable behavior of a data structure with hidden state can be understood entirely in terms of (concurrent) invocations of its operations; the particular contents of the heap can be ignored. The problem is that the behavior of its operations—from which its histories are generated—*is* dependent on the heap. Any proof method that uses linearizability as a component must reason, at some level, about the heap. Moreover, linearizability is defined by quantifying over *all* histories, a quantification that cannot be straightforwardly tackled through induction. Practical ways of proving linearizability require additional technical machinery, the validity of which must be separately proved.

[26] Robin Milner (1977), "Fully abstract models of the lambda calculus"

A similar progression of techniques has been developed for reasoning *directly* about refinement (or equivalence), mainly based on (bi)simulations[27] or (Kripke) logical relations.[28] These methods directly account for the role of hidden state in the evolving behavior of a data structure, and they also scale to higher-order languages with abstract and recursive types. Recent logical relations have, in particular, given rise to an abstract and visual characterization of protocols governing hidden state, based on state-transition systems.[29]

The next two chapters show how to extend logical relations to deal with sophisticated concurrent algorithms—connecting the correctness of these algorithms to the theory of data abstraction while avoiding linearizability altogether.

[27] Abramsky (1990); Sumii and Pierce (2005); Sangiorgi *et al.* (2007); Koutavas and Wand (2006)

[28] Pitts and Stark (1998); Ahmed (2006); Ahmed *et al.* (2009); Dreyer, Neis, and Birkedal (2010)

[29] Dreyer, Neis, and Birkedal (2010), "The impact of higher-order state and control effects on local relational reasoning"

4

Local protocols

▸ Synopsis This chapter introduces local protocols and develops, through examples, the key ideas we use to handle scalable algorithms: role playing via tokens (§4.3), spatial locality via local life stories (§4.2), thread locality via specification resources (§4.4), and temporal locality via speculation (§4.5).

"A programmer should be able to prove that his programs have various properties and do not malfunction, solely on the basis of what he can see from his private bailiwick."

—James H Morris Jr., "Protection in programming languages"

4.1 Overview

The motivation for proving a refinement $e_I \preceq e_S$ is to enable a client of a library e_I to reason about it in terms of some more abstract specification e_S. But to actually prove refinement, it is vital to view *the client* more abstractly:

- Definitionally, refinement requires consideration of *every* well-typed context C into which a library e_I can be placed. In the execution of the resulting program $C[e_I]$, the client C may invoke the library e_I many times and in many ways—perhaps even forking threads that use it concurrently. In between these interactions with the library, the client may perform its own local computations.
- Intuitively, refinement should not depend on the execution of any *particular* clients, since the library must be well-behaved for *all* (well-typed) clients. Moreover, libraries that hide their state within an abstraction barrier greatly limit the scope of interaction: a client context can neither observe nor alter the hidden state directly, so all interactions are mediated by the library's methods. From the perspective of a library, then, the behavior of a client can be reduced to a collection of possibly-concurrent method invocations.

To bring definition and intuition into alignment, we need a way of modeling arbitrary client behavior without enumerating particular clients.

Protocols are the answer. They characterize hidden state: what it is, and how it can change. The only way a client can interact with hidden state is through the library, so protocols need only explain the behavior of the library implementation[1] with respect to its hidden state. A *particular* client is abstractly modeled by the moves in the protocol it makes through calls to the library. An *arbitrary* client can then modeled by considering arbitrary protocol moves. Protocols enable us to reason about method invocations in isolation. Instead of considering an arbitrary sequence of prior method invocations, we simply start from an arbitrary protocol state. And instead of

"Protocol is everything."

—Francois Giuliani

[1] Morris Jr. (1973)'s "private bailiwick."

considering arbitrary concurrent invocations, we simply force our reasoning to withstand arbitrary "rely" (environment) moves in the protocol.

4.1.1 *The state transition system approach*

To make our discussion more concrete (and give some useful background), we briefly review Dreyer, Neis, and Birkedal (2010)'s state transition system (STS) approach to reasoning about hidden state in sequential languages.

Suppose we want to prove that oddCnt ≤ cnt (§2.2.2), *i.e.,* that:

```
let r    = new 37                          let r    = new 0
    inc  = λ(). r := get r + 2                 inc  = λ(). r := get r + 1
    read = λ(). (get r − 37)/2        ≤        read = λ(). get r
    test = λ(). isOdd(get r)                   test = λ(). true
in (inc, read, test)                       in (inc, read, test)
```

The hidden state of the oddCnt "library" is just the hidden reference r, which is embedded in the exported closures (inc, read, test) but is not itself exported directly. Consequently, the values it can take on are entirely determined by (1) its initial value and (2) the modifications made possible by the library methods. Since inc is the only method than can update the reference, we describe r with the following STS:

Although we have labeled the nodes suggestively, in Dreyer *et al.*'s approach the states of such transition systems are *abstract*: each state is interpreted as an assertion on heaps. In Chapter 5 we will introduce a syntax and semantics for assertions, but in the meantime we will describe them as needed.

For the above STS, the interpretation for a node n is

$$I(n) \triangleq r_{\mathrm{I}} \mapsto_{\mathrm{I}} n \;*\; r_{\mathrm{S}} \mapsto_{\mathrm{S}} \left(\frac{n - 37}{2}\right)$$

The interpretation I reveals a fundamental fact: refinement is proved via "relational" reasoning, meaning in particular that we relate the state of the implementation heap (using separation logic-style[2] heap assertions $x \mapsto_{\mathrm{I}} y$) to the state of the specification heap (using heap assertions $x \mapsto_{\mathrm{S}} y$).[3] After all, like oddCnt, the specification cnt has hidden state that affects its behavior. Any execution of the implementation must be mimicked by its specification, in the style of a simulation. In particular, there are dual obligations for the implementation and specification of a method:[4]

1. For every step an implementation method makes, any changes to the implementation heap must be permitted by the STS, possibly by moving to a new STS state. (*Impl. step* ⇒ *STS step*)
2. For any move made in the STS, it must be possible to take zero or more steps in the specification method yielding a new specification heap satisfying the new STS state. (*STS step* ⇒ *Spec. step*)

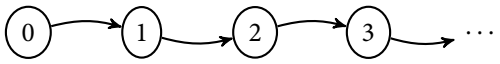
[2] In separation logic, the assertion $x \mapsto y$ is satisfied by a heap with a single location, x, that contains the value y (Reynolds 2002), and $*$ is a version of conjunction for which the two conjuncts must be satisfied by disjoint portions of the heap.

[3] To distinguish the uses of the bound variable r in oddCnt and cnt, we α-rename it to r_{I} and r_{S} respectively.

[4] Thus the interpretation I plays the role of a "linking invariant" (or "refinement map"; Hoare 1972; Abadi and Lamport 1991)—as one would expect to find in any refinement proof—but one whose meaning is relative to the current STS state.

To prove that oddCnt refines cnt, we must in particular show refinement for each method. Suppose we execute inc in oddCnt starting from an arbitrary STS state n, and hence a heap where r_I has value n. In its second step of execution, inc will update r_I to $n + 2$, which requires moving to the next STS state—and hence, showing that cnt's version of inc can be executed to take r_S from $\frac{n-37}{2}$ to $\frac{(n+2)-37}{2}$, which indeed it can.[5] The proof of refinement for read is even more pleasantly trivial.[6] Finally, refinement for test is also easy, since the STS is constrained enough to imply that r_I is always odd.

As an aside, we could have instead labeled the states by their "abstract" value, and make the interpretation do the work of picking out the corresponding concrete value:



with interpretation

$$I(n) \triangleq r_I \mapsto_I (2n + 37) * r_S \mapsto_S n$$

This "alternative" STS/interpretation is semantically identical to the original one.

It turns out that, however, that the original STS is overkill: the fact that oddCnt refines its spec does not in any way depend on the monotonically-increasing nature of r. All that is really needed is an invariant. Simple invariants can be expressed by one-state systems, *e.g.*,

with the interpretation giving the invariant:

$$I(\mathsf{DummyState}) \triangleq \exists n.\, r_I \mapsto_I (2n + 37) * r_S \mapsto_S n$$

Even though there are no "moves" in the STS, a change to the implementation heap requires a corresponding change to the specification heap in order for the dummy state's interpretation to be satisfied—so the proof obligations for refinement using this STS are essentially the same as for the original one.

▸ As is well known, invariants are not always enough, especially when reasoning about concurrency.[7] We close the discussion of Dreyer, Neis, and Birkedal (2010) with a classic, so-called "awkward" example,[8] where invariants fail even in the sequential setting:

$$\textbf{let } x = \textbf{new } (0) \textbf{ in } \lambda f.\, x := 1;\ f();\ \mathsf{get}(x) \quad \preceq \quad \lambda f.\, f();\ 1$$

The example captures, in the most trivial way possible, hidden state that is lazily initialized. It is contrived, but illustrative and subtle. The subtlety arises from the dynamics of control. The implementation[9] first allocates x with value 0, and then yields control to the client, returning a closure. When the client later invokes the closure, the internal state is updated to 1, but then

[5] We know, at the outset, that $n = 2m+37$ for some m, due to our choice of node labels.

[6] As if the interpretation I was designed specifically to support it. . .

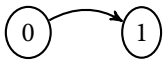
[7] Jones (1983), "Tentative steps toward a development method for interfering programs"

[8] The example is due to Pitts and Stark (1998), who dubbed it "awkward" because their invariant-based method could not cope with it.

[9] The term on the left.

control is *again* returned to the client (by invoking the callback, f). How do we know, upon return, that x has the value 1?[10]

Intuitively, x must have the value 1 because the only code that can change x is the very function under consideration—which sets it to 1.[11] Formally, we can capture the irreversible state change of x via a simple STS:



with the interpretation $I(n) = x \mapsto_1 n$.[12] According to this STS, after the expression $x := 1$ is executed, the protocol is in its final state—and, no matter what the callback does, there's no going back.

[10] This is the problem of sharing (§2.2.2) in action, mitigated by abstraction (§2.2.4).

[11] Notice that this function could very well be invoked from the client's callback. Higher-order programming leads to reentrancy.

[12] With this example, the implementation's internal state has no observable effect on its behavior, and so the specification has no hidden state at all.

4.1.2 *Scaling to scalable concurrency*

Our "local protocols" build on Dreyer, Neis, and Birkedal (2010)'s state transition systems,[13] generalizing them along several dimensions in order to handle scalable concurrent algorithms. We next briefly outline these extensions, and then devote the rest of the chapter to explaining them through examples. Chapter 5 then formalizes protocols and a model built around them.

[13] Including its use of a step-indexed Kripke logical relation as a foundation for proving refinement, as we will see in Chapter 5. The "Kripke" frame is essentially the protocol itself.

- SPATIAL LOCALITY As we stressed in §1.2.1, we seek to understand fine-grained concurrency at a fine grain, *e.g.,* by characterizing concurrent interaction at the level of individual nodes. Compared to Dreyer, Neis, and Birkedal (2010)'s work, then, we deploy our protocols at a much finer granularity and use them to tell *local life stories* about individual nodes of a data structure (§4.2). Of course, there are also "global" constraints connecting up the life stories of the individual nodes, but to a large extent we are able to reason at the level of local life stories and their local interactions with one another.

- ROLE-PLAYING Many concurrent algorithms require the protocols governing their hidden state to support *role-playing*—that is, a mechanism by which different threads participating in the protocol can dynamically acquire certain "roles". These roles may enable them to make certain transitions that other threads cannot. Consider for example a locking protocol, under which the thread that acquires the lock adopts the unique role of "lock-holder" and thus knows that no other thread has the ability to release the lock. To account for role-playing, we enrich our protocols with a notion of *tokens* (§4.3), which abstractly represent roles. The idea is that, while the transition system defines the basic ways hidden state can change, some transitions "reward" a thread by giving it ownership of a token, whereas other transitions require threads to give up a token as a "toll" for traversal.[14]

- THREAD- AND TEMPORAL LOCALITY In proving refinement for an operation on a concurrent data structure, a key step is identifying the point during

[14] This idea is highly reminiscent of recent work on "concurrent abstract predicates" (Dinsdale-Young *et al.* 2010), but we believe our approach is simpler and more direct. See Chapter 7 for further discussion.

its execution at which the operation can be considered to have "committed", *i.e.,* the point at which its canonical atomic spec can be viewed as having executed.[15] With scalable algorithms, these commit points can be hard to identify in a thread-local and temporally-local way. For example, in an elimination stack (§2.4.5) a thread attempting to pop an element might eliminate its operation against a concurrent push—thereby committing a push operation that, semantically, belongs to another thread. In other algorithms like CCAS[16] (which we discuss in Chapter 6), the nondeterminism induced by shared-state concurrency makes it impossible to determine *when* the commit has occurred until after the fine-grained operation has completed its execution.

[15] This is often called the "linearization point" of the algorithm when proving linearizability (§2.5.1).

[16] Fraser and Tim Harris (2007), "Concurrent programming without locks"

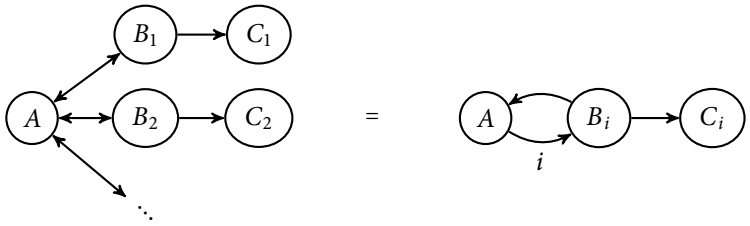
On the other hand, the protocol method *bakes in* thread- and temporal locality. At each point in time, the protocol is in a definite state, and changes to that state are driven by *single* steps of implementation code without regard to past or future steps. External threads are treated entirely abstractly—they merely "cast a shadow" on the protocol by updating its state in between the steps of the thread of interest. So the question is: how can the inherent locality of protocols be reconciled with the apparent nonlocality of sophisticated algorithms?

The whole point of protocols (or, more generally, Kripke logical relations) is to provide a way of describing local knowledge about hidden resources, but in prior work those "hidden resources" have been synonymous with "local variables" or "a private piece of the heap". To support thread- and temporally-local reasoning about scalable algorithms, we make two orthogonal generalizations to the notion of resources:

- We extend resources to include *specification code* (§4.4). This extension makes it possible for "the right to commit an operation" (*e.g.,* push, in the example above) to be treated as an abstract, shareable resource, which one thread may pass to other "helper" threads to run on its behalf.
- We extend resources to include *sets of specification states* (§4.5). This extension makes it possible to *speculate* about all the possible specification states that an implementation could be viewed as refining, so that we can wait until the implementation has finished executing to decide which one we want to choose.

4.1.3 *A note on drawing transition systems*

In our examples, we use a compact notation to draw structured branches:



4.2 SPATIAL LOCALITY VIA LOCAL LIFE STORIES

4.2.1 *A closer look at linking: Michael and Scott's queue*

The granularity of a concurrent data structure is a measure of the locality of synchronization between threads accessing it. Coarse-grained data structures provide exclusive, global access for the duration of a critical section: a thread holding the lock can access as much of the data structure as needed, secure in the knowledge that it will encounter a consistent, frozen representation. By contrast, fine-grained data structures localize or eliminate synchronization, forcing threads to do their work on the basis of limited knowledge about its state—sometimes as little as what the contents of a single word are at a single moment.

While we saw some simple examples of scalable, fine-grained data structures in §2.4, we now examine a more complex example—a variant of Michael and Scott's lock-free queue[17]—that in particular performs a lock-free traversal of concurrently-changing data. The queue, given in Figure 4.1, is represented by a *nonempty* linked list; the first node of the list is considered a "sentinel" whose data does not contribute to the queue. Here the list type is mutable,

$$\mathsf{list}(\tau) \triangleq \mu\alpha.\mathbf{ref}_?((\mathbf{ref}_?(\tau), \alpha)$$

in addition to the mutable head reference.

[17] Michael and Scott (1998), "Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors"

```
MSQ: ∀α. (unit → ref?(α)) × (α → unit)
MSQ ≜ Λ.
  let head = new (new (null, null))              (* initial sentinel *)

    deq  = λ(). let n = get head
                in case get(n[2])
                     of n'   ⇒ if cas(head, n, n')
                               then get(n'[1]) else deq()
                      | null ⇒ null,             (* queue is empty *)

    enq  = λx. let n = new (new x, null)         (* node to link in *)
               let try = λc. case get(c[2])
                   of c'   ⇒ try(c')             (* c is not the tail *)
                    | null ⇒ if cas(c[2], null, n)
                             then () else try(c)
               in try(get head)                  (* start search from head *)

  in (deq, enq)
```

Figure 4.1: A simplified variant of Michael and Scott (1998)'s lock-free queue

Nodes are dequeued from the front of the list, so we examine the deq code first. If the queue is *logically* nonempty, it contains at least two nodes: the sentinel (physical head), and its successor (logical head). Intuitively, the deq operation should atomically update the head reference from the sentinel to

its successor; after doing so, the old logical head becomes the new sentinel, and the next node, if any, becomes the new logical head. Because there is no lock protecting head, however, a concurrent operation could update it at any time. Thus, deq employs optimistic concurrency (§2.4.2): after gaining access to the sentinel by dereferencing head, it does some additional work—finding the logical head—while optimistically assuming that head has not changed behind its back. In the end, optimism meets reality through **cas**, which performs an atomic update only when head is unchanged. If its optimism was misplaced, deq must start from scratch. After all, the queue's state may have entirely changed in the interim.

The key thing to notice is just how little knowledge deq has as it executes. Immediately after reading head, the most that can be said is that the resulting node *was once* the physical head of the queue. The power of **cas** is that it mixes instantaneous knowledge—the head *is now* n—with instantaneous action—the head *becomes* n'. The weakness of **cas** is that this potent mixture applies only to a single word of memory. For deq, this weakness is manifested in the lack of knowledge **cas** has about the new value n', which *should* still be the successor to the physical head n at the instant of the **cas**. Because **cas** cannot check this fact, it must be established *pessimistically*, *i.e.,* guaranteed to be true on the basis of the queue's internal protocol. We will see in a moment how to formulate the protocol, but first, we examine the more subtle enq.

In a singly-linked queue implementation, one would expect to have both head and tail pointers, and indeed the full Michael-Scott queue includes a "tail" pointer. However, because **cas** operates on only one word at a time, it is impossible to use a single **cas** operation to both link in a new node and update a tail pointer. The classic algorithm allows the tail pointer to lag behind the true tail by at most one node, while the implementation in java.util.concurrent allows multi-node lagging. These choices affect performance, of course, but from a correctness standpoint one needs to make essentially the same argument whether one has a lagging tail, or simply traverses from head as we do.

In all of these cases, it is necessary to *find* the actual tail of the list (whose successor is **null**) by doing some amount of traversal. Clearly, this requires at least that the actual tail be reachable from the starting point of the traversal; the loop invariant of the traversal is then that the tail is reachable from the current node. But in our highly-concurrent environment, we must account for the fact that the data structure is changing under foot, even as we traverse it. The node that was the tail of the list when we *began* the traversal might not even be *in* the data structure by the time we finish.

4.2.2 *The story of a node*

We want to prove that MSQ refines its coarse-grained specification CGQ, shown in Figure 4.2. Ideally, the proof would proceed in the same way one's intuitive reasoning does, *i.e.,* by considering the execution of a single function by a single thread one line at a time, reasoning about what is known at

```
CGQ ≜ Λ.let head = new (null)
          deq  = case get head
                   of n     ⇒ head := get(n[2]); new (get(n[1]))
                    | null  ⇒ null,            (* queue is empty *)
          enq  = λx. let enq′ = λc. case get(c[2])
                                     of c′    ⇒ enq′(c′)   (* c is not the tail *)
                                      | null  ⇒ c[2] := new (x, null)
                                   in case get(head) of n    ⇒ enq′(n)
                                                      | null ⇒ head := new (x, null)
       in mkAtomic(deq, enq)
```

Figure 4.2: A coarse-grained queue

each program point. To achieve this goal, we must solve two closely-related problems: we must characterize the possible interference from concurrent threads, and we must characterize the knowledge that our thread can gain.

We solve both of these problems by introducing a notion of *protocol*, based on the abstract STSs of Dreyer, Neis, and Birkedal, but with an important twist: we apply these transition systems at the level of *individual nodes*, rather than as a description of the entire data structure. The transition systems describe what we call the *local life stories* of each piece of a data structure. The diagram in Figure 6.6 is just such a story. Every heap location can be seen as a *potential* node in the queue, but all but finitely many are "unborn" (state $\bot$). After birth, nodes go through a progression of life changes. Some changes are manifested physically. The transition from $\mathsf{Live}(v, \mathbf{null})$ to $\mathsf{Live}(v, \ell)$, for example, occurs when the successor field of the node is updated to link in a new node. Other changes reflect evolving relationships. The transition from Live to Sentinel, for example, does not represent an internal change to the node, but rather a change in the node's position in the data structure. Finally, a node "dies" when it becomes unreachable.

"There's always the chance you could die right in the middle of your life story."

—Chuck Palahniuk

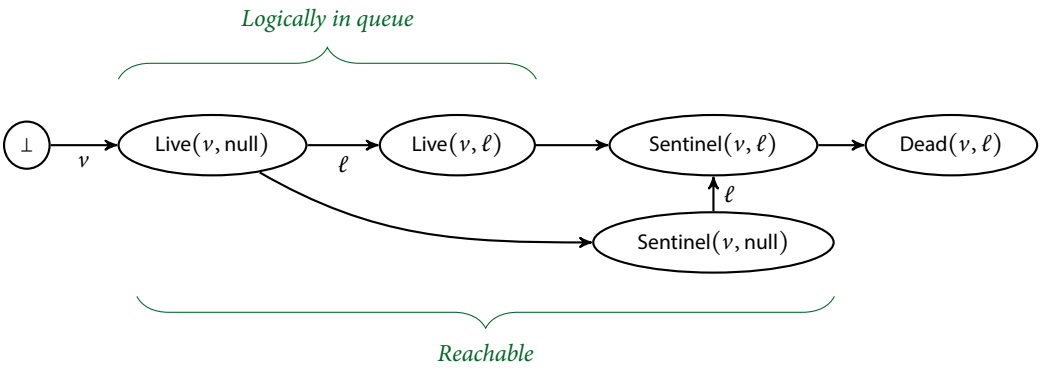


Figure 4.3: A protocol for *each node* of the Michael-Scott queue—one per possible memory location.

The benefit of these life stories is that they account for knowledge and interference together, in a local and abstract way. Knowledge is expressed by asserting that a given node is *at least* at a certain point in its life story. This kind of knowledge is inherently stable under interference, because *all* code must conform to the protocol, and is therefore constrained to a forward

march through the STS. The life story gathers together in one place all the knowledge and interference that is relevant to a given node, even knowledge about an ostensibly global property like "reachability". This allows us to draw global conclusions from local information, which is precisely what is needed when reasoning about scalable concurrency. For example, notice that no node can die with a **null** successor field. A successful **cas** on the successor field from **null** to some location—like the one performed in enq—entails that the successor field was instantaneously **null** (local information), which by the protocol means the node was instantaneously reachable (global information), which entails that the **cas** makes a new node reachable. Similarly, the protocol makes it immediately clear that the queue is free from any ABA problems (§2.4.3), because nodes cannot be reincarnated and their fields, once non-**null**, never change.

▸ To formalize this reasoning, we must connect the abstract account of knowledge and interference provided by the protocol to concrete constraints on the queue's representation, ensuring that "Dead" and "Live" mean what we think they do. For the queue, we have the following set of states for each node's local STS:

$$\begin{aligned} S_0 \triangleq\ & \{\bot\} \\ \cup\ & \{\mathsf{Live}(v, v') \mid v, v' \in \mathrm{Val}\} \\ \cup\ & \{\mathsf{Sentinel}(v, v') \mid v, v' \in \mathrm{Val}\} \\ \cup\ & \{\mathsf{Dead}(v, \ell) \mid v \in \mathrm{Val}, \ell \in \mathrm{Loc}\} \end{aligned}$$

along with the transition relation $\rightsquigarrow_0$ as given in the diagram (Figure 6.6).[18] These local life stories are then systematically lifted into a *global* protocol: the data structure as a whole is governed by a *product* STS with states $S \triangleq \mathrm{Loc} \xrightarrow{\text{fin}} S_0$, where $\xrightarrow{\text{fin}}$ indicates that all but finitely many locations are in the $\bot$ (unborn) state in their local STS.[19] The transition relation $\rightsquigarrow$ for the product STS lifts the one for each node's STS, pointwise:

$$s \rightsquigarrow s' \quad \text{iff} \quad \forall \ell.\ s(\ell) \rightsquigarrow_0 s'(\ell) \ \vee\ s(\ell) = s'(\ell)$$

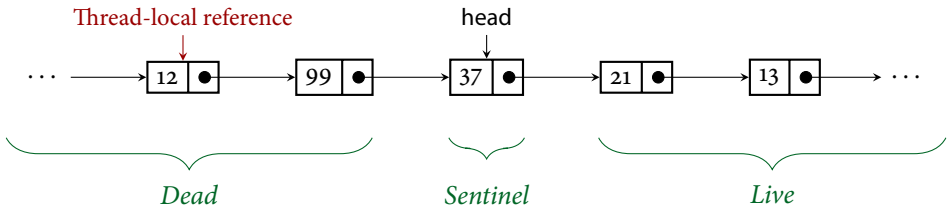
Thus, at the abstract level, the product STS is simply a collection of independent, local STSs.

At the concrete level of the interpretation I, however, we record the constraints that tie one node's life story to another's; see Figure 4.4. As the example at the top of the figure shows, the linked list managed by the protocol does *not* begin at head. Instead it begins with a prefix of the Dead nodes—nodes that are no longer reachable from head, but that may still be referenced locally by a snapshot in some thread. The node pointed to be head is the sole Sentinel, while the remaining suffix of the list constitute the Live nodes. The interpretation guarantees such a configuration of nodes by first breaking the product state s into three disjoint pieces:

$$s \quad = \quad s_D \ \uplus\ [\ell \mapsto \mathsf{Sentinel}(v_0, v_1)] \ \uplus\ s_L$$

[18] Recall that the annotated edges denote branches for choosing particular concrete v and ℓ values (§4.1.3).

[19] That is, we pun the $\bot$ state in the local STS with the partiality of a product state s as a function: the domain of s is just those locations that have been "born" into the queue.



$$I(s) \triangleq DeadSeg(s_D, -, \ell) * \begin{pmatrix} \mathsf{head}_\mathrm{I} \mapsto_\mathrm{I} \ell * \ell \mapsto_\mathrm{I} (v_0, v_\mathrm{I}) * \\ \mathsf{head}_\mathrm{S} \mapsto_\mathrm{S} v_\mathrm{S} * \mathsf{lock}_\mathrm{S} \mapsto_\mathrm{S} \mathbf{false} \end{pmatrix} * LiveSeg(s_L, v_\mathrm{I}, v_\mathrm{S}, \mathbf{null}, \mathbf{null})$$

for any $\ell, v_0, v_\mathrm{I}, v_\mathrm{S}, s_D, s_L$
with $s = s_D \uplus [\ell \mapsto \mathsf{Sentinel}(v_0, v_\mathrm{I})] \uplus s_L$, where:

$$\begin{aligned} DeadSeg(\varnothing,\ \ell,\ \ell'') &\triangleq \mathsf{emp} \wedge \ell = \ell'' \\ DeadSeg(s_D \uplus [\ell \mapsto \mathsf{Dead}(v, \ell')],\ \ell,\ \ell'') &\triangleq \ell \mapsto_\mathrm{I} (v, \ell') * DeadSeg(s_D, \ell', \ell'') \end{aligned}$$

$$\begin{aligned} LiveSeg(\varnothing,\ v_\mathrm{I},\ v_\mathrm{I}'',\ v_\mathrm{S},\ v_\mathrm{S}'') &\triangleq \mathsf{emp} \wedge v_\mathrm{I} = v_\mathrm{I}'' \wedge v_\mathrm{S} = v_\mathrm{S}'' \\ LiveSeg(s_L \uplus [v_\mathrm{I} \mapsto \mathsf{Live}(v_0, v_\mathrm{I}')],\ v_\mathrm{I},\ v_\mathrm{I}'',\ v_\mathrm{S},\ v_\mathrm{S}'') &\triangleq \exists x_\mathrm{I}, x_\mathrm{S}, v_\mathrm{S}'.\ x_\mathrm{I} \preceq^{\mathcal{V}} x_\mathrm{S} : \alpha * v_0 \mapsto_\mathrm{I} x_\mathrm{I} \\ &\quad * v_\mathrm{I} \mapsto_\mathrm{I} (v_0, v_\mathrm{I}') * v_\mathrm{S} \mapsto_\mathrm{S} (x_\mathrm{S}, v_\mathrm{S}') \\ &\quad * LiveSeg(s_L, v_\mathrm{I}', v_\mathrm{I}'', v_\mathrm{S}', v_\mathrm{S}'') \end{aligned}$$

Figure 4.4: Interpreting the lifted, global protocol

Due to the other constraints of I, the s_D part of the state must contain exactly the Dead nodes, while s_L contains exactly the Live ones—and there must, therefore, be exactly one sentinel:

- The *DeadSeg* predicate is essentially the same as the recursive "list segment" predicate from separation logic:[20] $DeadSeg(s_D, \ell, \ell'')$ is satisfied iff s_D represents a linked list segment of Dead nodes, starting at ℓ and terminating at ℓ'', and each such node actually exists on the implementation heap.[21]
- In contrast, the more complicated *LiveSeg* predicate is a *relational* version of the list segment predicate: $LiveSeg(s_L, v_\mathrm{I}, v_\mathrm{I}'', v_\mathrm{S}, v_\mathrm{S}'')$ holds iff s_L represents a linked list segment of Live nodes starting at v_I and ending at v_I'' on the implementation heap, with a corresponding list segment starting at v_S and ending at v_S'' on the spec heap. The predicate is indexed by general values, rather than just locations, because the Live segment can terminate with **null** (which is not a location). Since the queue is parametric over the type α of its data, the data stored in each Live implementation node must *refine* the data stored in the spec node at type α, written $x_\mathrm{I} \preceq^{\mathcal{V}} x_\mathrm{S} : \alpha$; see Chapter 5.

The interpretation also accounts for two representation differences between MSQ and CGQ. First, the node data in the implementation is stored in a $\mathbf{ref}_?(\alpha)$, while the specification stores the data directly. Second, the specification has a lock. The invariant requires that the lock is always free (**false**) because, as we show that MSQ refines CGQ, we always run entire critical sections of CGQ at once, going from unlocked state to unlocked state. These "big steps" of the CGQ correspond to the linearization points of the MSQ.

[20] Reynolds (2002), "Separation logic: a logic for shared mutable data structures"

[21] We could also pick out the corresponding "garbage" nodes on the spec side, but there is no reason to do so.

▸ A VITAL CONSEQUENCE OF THE INTERPRETATION is that Dead nodes must have non-**null** successor pointers whose locations are in a non-⊥ state (*i.e.*, they are at least Live). This property is the key for giving a simple, *local* loop invariant for enq, namely, that the current node c is *at least* in a Live state. It follows that if the successor pointer of c is not **null**, it must be another node at least in the Live state. If, on the other hand, the successor node of c *is* **null**, we know that c cannot be Dead, but is at least Live, which means that c must be (at that instant) reachable from the implementation's head pointer. Thus we gain global reachability information on the basis of purely local knowledge about the protocol state.

More generally, while the interpretation I is clearly global, it is designed to support compositional, spatially-local reasoning. Every part of its definition is based on *decomposing* the product state s into disjoint pieces, with only neighbor-to-neighbor interactions. Thus when reasoning about updating a node, for example, it is possible to break s into a piece s_N corresponding to the node (and perhaps its immediate neighbor) and a "frame" s_F for the rest of the data structure. The interpretation can then be shown to hold on the basis of some updated s'_N with the same frame s_F—meaning that the rest of the data structure need never be examined in verifying the local update. The detailed proof outline for MSQ (as well as the other examples in this chapter) is given in Chapter 6, and it includes a more detailed treatment of the mechanics of spatial locality.

4.3 ROLE-PLAYING VIA TOKENS

"The essence of a role-playing game is that it is a group, cooperative experience."

—Gary Gygax

Although Michael and Scott's queue is already tricky to verify, there is a specific sense in which its protocol in Figure 4.1 is simple: it treats all threads equally. All threads see a level playing field with a single notion of "legal" transition, and any thread is free to make any legal transition according to the protocol. Many concurrent algorithms, however, require more refined protocols in which different threads can play different *roles*—granting them the rights to make different sets of transitions—and in which threads can acquire and release these roles dynamically as they execute.

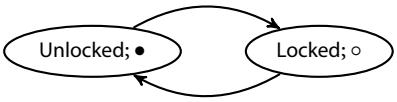
In fact, one need not look to scalable concurrency for instances of this dynamic role-playing—the simple lock used in the coarse-grained "spec" of the Michael-Scott queue is a perfect and canonical example. In a protocol governing a single lock (*e.g.*, lock, in CGQ), there are two states: Unlocked and Locked. Starting from the Unlocked state, all threads should be able to acquire the lock and transition to the Locked state. But not vice versa: once a thread has acquired the lock and moved to the Locked state, it has adopted the role of "lock-holder" and should know that *it* is the only thread with the right to release the lock and return to Unlocked.

To support this kind of role-playing, our protocols enrich STSs with a notion of *tokens*, which are used to grant authority over certain types of actions in a protocol. Each STS may employ its own appropriately chosen

set of tokens, and each thread may *privately own* some subset of these tokens. The idea, then, is that certain transitions are only legal for the thread that privately owns certain tokens. Formally this is achieved by associating with each state in the STS a set of tokens that are currently *free*, *i.e.,* not owned by any thread.[22] We then stipulate *the law of conservation of tokens*: for a thread to legally transition from state s to state s', the (disjoint) union of its private tokens and the free tokens must be the same in s and in s'.

[22] Another perspective is that the free tokens are owned by the STS itself, as opposed to the threads participating in the protocol; *cf.* concurrent separation logic (O'Hearn 2007).

For instance, in the locking protocol, there is just a single token—call it TheLock. In the Unlocked state, the STS asserts that TheLock must belong to the free tokens and thus that *no* thread owns it privately, whereas in the Locked state, the STS asserts that TheLock does *not* belong to the free tokens and thus that *some* thread owns it privately. Pictorially, • denotes that TheLock is in the free tokens, and ○ denotes that it is not:



When a thread acquires the physical lock and transitions to the Locked state, it must add TheLock to its private tokens in order to satisfy conservation of tokens—and it therefore takes on the abstract role of "lock holder". Thereafter, no other thread may transition back to Unlocked because doing so requires putting TheLock back into the free tokens of the STS, which is something only the private owner of TheLock can do. For a typical coarse-grained data structure, the interpretation for the Unlocked state would assert ownership of all of the hidden state for the data structure, while the Locked state would own nothing. Thus, a thread taking the lock also acquires the resources it protects, but must return these resources on lock release (in the style of concurrent separation logic[23]).

[23] O'Hearn (2007), "Resources, concurrency, and local reasoning"

As this simple example suggests, tokens induce very natural *thread-relative* notions of *rely* and *guarantee* relations on states of an STS. For any thread i, the total tokens A of an STS must equal the disjoint union of i's private tokens A_i, the free tokens A_{free} in the current state s, and the "frame" tokens A_{frame} (*i.e.,* the combined private tokens of all other threads but i). The *guarantee* relation says which future states thread i may transition to, namely those that are accessible by a series of transitions that i can "pay for" using its private tokens A_i. Dually, the *rely* relation says which future states other threads may transition to, namely those that are accessible by a series of transitions that can be paid for *without* using i's private tokens A_i (*i.e.,* only using the tokens in A_{frame}). These two relations play a central role in our model (Chapter 5).

4.4 THREAD LOCALITY VIA SPECIFICATIONS-AS-RESOURCES

As explained in §2.4.5, some algorithms use side channels, separate from the main data structure, to enable threads executing different operations to cooperate. To illustrate this, we use a toy example—inspired specifically by

elimination stacks[24]—that isolates the essential challenge of reasoning about cooperation, minus the full-blown messiness of a real data structure.

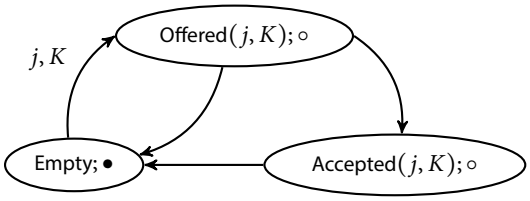
[24] Hendler *et al.* (2004), "A scalable lock-free stack algorithm"

▸ THE FLAG IMPLEMENTATIONS.

```
redFlag ≜
  let flag  = new true,
      chan  = new 0,
      flip  = λ(). if cas(chan, 1, 2) then () else
                   if cas(flag, true, false) then () else
                   if cas(flag, false, true) then () else
                   if cas(chan, 0, 1) then
                     if cas(chan, 1, 0) then flip() else chan := 0
                   else flip(),
      read  = λ(). get flag
  in (flip, read)
```

```
≤ blueFlag ≜
    let flag  = new true,
        flip  = λ(). flag := not (get flag),
        read  = λ(). get flag
    in mkAtomic(flip, read)
```

▸ THE PROTOCOL.



▸ THE PROTOCOL STATE INTERPRETATION.

$$
\begin{aligned}
I(\mathsf{Empty}) &\triangleq Q * \mathsf{chan} \mapsto_{\mathrm{I}} 0 \\
I(\mathsf{Offered}(j, K)) &\triangleq Q * \mathsf{chan} \mapsto_{\mathrm{I}} 1 * j \rightarrowtail_{\mathrm{S}} K[\mathsf{flip}_{\mathrm{S}}()] \\
I(\mathsf{Accepted}(j, K)) &\triangleq Q * \mathsf{chan} \mapsto_{\mathrm{I}} 2 * j \rightarrowtail_{\mathrm{S}} K[()]
\end{aligned}
$$

$$
Q \triangleq \exists x : \mathbf{bool}.\ \mathsf{flag}_{\mathrm{I}} \mapsto_{\mathrm{I}} x * \mathsf{flag}_{\mathrm{S}} \mapsto_{\mathrm{S}} x * \mathsf{lock} \mapsto_{\mathrm{S}} \mathbf{false}
$$

Figure 4.5: Red flags versus blue flags

Figure 4.5 shows the example, in which redFlag is a lock-free implementation of blueFlag.[25] The latter is a very simple data structure, which maintains a hidden boolean flag, and provides operations to flip it and read it. One obvious lock-free implementation of flip would be to keep running **cas**(flag, **true**, **false**) and **cas**(flag, **false**, **true**) repeatedly until one of them succeeds. However, to demonstrate cooperation, redFlag does something more "clever": in addition to maintaining flag, it also maintains a side channel chan, which it uses to enable two flip operations to cancel each other out without ever modifying flag at all!

More specifically, chan adheres to the following protocol, which is visualized in Figure 4.5 (ignore the K's for now). If chan $\mapsto_{\mathrm{I}} 0$, it means the side channel is not currently being used (it is in the Empty state). If chan $\mapsto_{\mathrm{I}} 1$, it means that some thread j has offered to perform a flip using the side channel and moved it into the Offered$(j, -)$ state. If chan $\mapsto_{\mathrm{I}} 2$, it means that another thread has accepted thread j's offer and transitioned to Accepted$(j, -)$—thus silently performing both flip's at once (since they cancel out)—but that thread j has not yet acknowledged that its offer was accepted.

Like the locking example, this protocol uses a single token—call it Offer—which is free in state Empty but which thread j moves into its private tokens

[25] This example was proposed by Jacob Thamsborg, who named the two data structures according to the colors associated with left- and right-wing political stances in Europe respectively. Readers in the U.S. should, therefore, swap the colors.

when it transitions to the Offered$(j, -)$ state. After that transition, due to its ownership of Offer, thread j is the only thread that has the right to revoke that offer by setting chan back to 0 and returning to Empty. On the other hand, *any* thread may transition from Offered$(j, -)$ to Accepted$(j, -)$, since the two states have identical free tokens, namely, none. Once in the Accepted$(j, -)$ state, though, thread j is again the only thread able to Empty the channel.

The implementation of flip in redFlag then works as follows. First, we use **cas** to check if another thread has offered to flip (*i.e.,* if chan $\mapsto_1$ 1), and if so, we accept the offer by setting chan to 2. We then immediately return, having implicitly committed both flips right then and there, without ever accessing flag. If that fails, we give up temporarily on the side-channel shenanigans and instead try to perform a bona fide flip by doing **cas**(flag, **true**, **false**) and **cas**(flag, **false**, **true**) as suggested above. If *that* fails as well, then we attempt to make an offer on the side channel by changing chan from 0 to 1. If our attempt succeeds, then we (rather stupidly[26]) try to immediately revoke the offer and loop back to the beginning. If perchance another thread has preempted us at this point and accepted our offer—*i.e.,* if **cas**(chan, 1, 0) fails, implying that another thread has updated chan to 2—then that other thread must have already committed our flip on our behalf. So we simply set chan back to 0, freeing up the side channel for other threads, and return. Finally, if all else fails, we loop again.

[26] At this point in a real implementation, it would make sense to wait a while for other threads to accept our offer, but we elide that detail since it is irrelevant for reasoning about correctness.

As far as the refinement proof is concerned, there are two key points here.

The first concerns the **cas**(chan, 1, 0) step. As we observed already, the failure of this **cas** implies that chan must be 2, a conclusion we can draw because of the way our protocol uses tokens. After the previous **cas**(chan, 0, 1) succeeded, we knew that we had successfully transitioned to the Offered$(j, -)$ state, and thus that our thread j now controls the Offer token. Local ownership of Offer means that other threads can only transition to a limited set of states via the rely ordering (*i.e.,* without owning Offer): they can either leave the state where it is, or they can transition to Accepted$(j, -)$. Thus, when we observe that chan is not 1, we know it *must* be 2.

The second, more interesting point concerns the *semantics* of cooperation. If we make an offer on chan, which is accepted by another thread, it should imply that the other thread performed our flip for us, so we don't have to. At least that's the intuition, but how is that intuition enforced by the protocol? That is, when we observe that our offer has been accepted, we do so merely by inspecting the current value of chan. But how do we know that the other thread that updated chan from 1 to 2 actually "performed our flip" for us? For example, as perverse as this sounds, what is to prevent redFlag from performing chan := 2 as part of its implementation of read?

▸ Our technique for enforcing the semantics of cooperation is to treat *specification code* as a kind of resource. We introduce a new assertion, $j \rightarrowtail_s e$,

which describes the knowledge that thread j (on the spec side) is poised to run the term e. Ordinarily, this knowledge is kept private to thread j itself, but in a cooperative protocol, the whole idea is that j should be able to pass control over its spec code e to other threads, so that they may execute some steps of e on its behalf.

Specifically, this assertion is used to give semantic meaning to the $\mathsf{Offered}(j, K)$ and $\mathsf{Accepted}(j, K)$ states in our protocol (see the interpretation I in Figure 4.5). In the former state, we know that $j \rightarrowtail_{\mathrm{s}} K[\mathsf{flip}_{\mathrm{s}}()]$, which tells us that thread j has offered its spec code $K[\mathsf{flip}_{\mathrm{s}}()]$ to be run by another thread, whereas in the latter state, we know that $j \rightarrowtail_{\mathrm{s}} K[()]$, which tells us that j's flip has been executed. (The K is present here only because we do not want to place any restrictions on the evaluation context of the flip operation.) These interpretations demand that whatever thread accepts the offer by transitioning from $\mathsf{Offered}(j, K)$ to $\mathsf{Accepted}(j, K)$ must take the responsibility not only of updating chan to 2 but also of executing $\mathsf{flip}_{\mathrm{s}}()$—and *only* $\mathsf{flip}_{\mathrm{s}}()$—on j's behalf. When j subsequently moves back to the Empty state, it regains private control over its specification code, so that other threads may no longer execute it.

4.5 TEMPORAL LOCALITY VIA SPECULATION

"If the world were good for nothing else, it is a fine subject for speculation."

—William Hazlitt

Another tricky aspect of reasoning about concurrent algorithms (like the "conditional CAS" example we consider in §6.6) is dealing with nondeterminism. The problem is that when proving that an algorithm refines some coarse-grained spec, we want to reason in a temporally-local fashion—*i.e.,* using something akin to a simulation argument, by which the behavior of each step of implementation code is matched against zero or more steps of spec code—but nondeterminism, it would seem, foils this plan.

To see why, consider the following example, which does not maintain any hidden state (hence no protocol), but nevertheless illustrates the core difficulty with nondeterminism:

$$\begin{aligned} \mathsf{rand} &\triangleq \lambda().\ \mathbf{let}\ y = \mathbf{new}\ \mathbf{false}\ \mathbf{in}\ (\mathbf{fork}\ y := \mathbf{true}); \mathbf{get}\ y \\ \mathsf{lateChoice} &\triangleq \lambda x.\ x := 0; \mathsf{rand}() \\ \mathsf{earlyChoice} &\triangleq \lambda x.\ \mathbf{let}\ r = \mathsf{rand}()\ \mathbf{in}\ x := 0; r \end{aligned}$$

We want to show that lateChoice refines earlyChoice. Both functions flip a coin (*i.e.,* use rand() to nondeterministically choose a boolean value) and set a given variable x to 0, but they do so in opposite orders. Intuitively, though, the order shouldn't matter: there is no way to observe the coin flip until the functions return. However, if we try to reason about the refinement using a simulation argument, we run into a problem. The first step of lateChoice is the setting of x to 0. To simulate this step in earlyChoice, we need to match the assignment of x to 0 as well, since the update is an externally observable effect. But to do that we must first flip earlyChoice's coin. While we have the freedom to choose the outcome of the flip,[27] the trouble is that we don't know

[27] For *every* implementation execution, we must construct *some* spec execution.

what the outcome should be: lateChoice's coin flip has yet to be executed.

The solution is simple: speculate! That is, if you don't know which spec states to step to in order to match an implementation step, then keep your options open and maintain a *speculative set of specification states* that are reachable from the initial spec state and consistent with any observable effects of the implementation step. In the case of lateChoice/earlyChoice, this means that we can simulate the first step of lateChoice (the setting of x to 0) by executing the *entire* earlyChoice function *twice*. In both speculative states x is set to 0, but in one the coin flip returns **true**, and in the other it returns **false**.

This reasoning is captured in the following Hoare-style proof outline:

$\{x_{\mathrm{I}} \preceq^{\mathcal{V}} x_{\mathrm{S}} : \mathbf{ref}(\mathbf{nat}) \land j \rightarrowtail_{\mathrm{S}} K[\mathsf{earlyChoice}(x_{\mathrm{S}})]\}$
$\quad x_{\mathrm{I}} := 0$
$\{x_{\mathrm{I}} \preceq^{\mathcal{V}} x_{\mathrm{S}} : \mathbf{ref}(\mathbf{nat}) \land (j \rightarrowtail_{\mathrm{S}} K[\mathbf{true}] \oplus j \rightarrowtail_{\mathrm{S}} K[\mathbf{false}])\}$
$\quad \mathsf{rand}()$
$\{\mathsf{ret}.\ (\mathsf{ret} = \mathbf{true} \lor \mathsf{ret} = \mathbf{false}) \land (j \rightarrowtail_{\mathrm{S}} K[\mathbf{true}] \oplus j \rightarrowtail_{\mathrm{S}} K[\mathbf{false}])\}$
$\{\mathsf{ret}.\ j \rightarrowtail_{\mathrm{S}} K[\mathsf{ret}]\}$

The precondition $j \rightarrowtail_{\mathrm{S}} K[\mathsf{earlyChoice}(x_{\mathrm{S}})]$—an instance of the assertions on specification code introduced in the previous section—denotes that initially the spec side is poised to execute earlyChoice. After we execute $x := 0$ in lateChoice, we speculate that the coin flip on the spec side could result in earlyChoice either returning **true** or returning **false**. This is represented by the speculative assertion $(j \rightarrowtail_{\mathrm{S}} K[\mathbf{true}] \oplus j \rightarrowtail_{\mathrm{S}} K[\mathbf{false}])$ appearing in the postcondition of this first step, in which the $\oplus$ operator provides a *speculative choice* between two subassertions characterizing possible spec states. In the subsequent step, lateChoice flips its coin, yielding a return value ret of either **true** or **false**. We can then refine the speculative set of specification states to whichever one (either $j \rightarrowtail_{\mathrm{S}} K[\mathbf{true}]$ or $j \rightarrowtail_{\mathrm{S}} K[\mathbf{false}]$) matches ret, and simply drop the other state from consideration. In the end, what matters is that we are left with at least one spec state that has been produced by a sequence of steps matching the observable behavior of the implementation's steps.

While the lateChoice/earlyChoice example is clearly contrived, it draws attention to an important point: client contexts are insensitive to the "branching structure" of nondeterminism. There is no way for a client to tell *when* a nondeterministic choice was actually made.[28] In practice, scalable algorithms sometimes employ deliberate nondeterminism, *e.g.,* during backoff (§2.4.4) or for complex helping schemes (§2.4.5), but their specifications generally do not—and so it is crucial that clients cannot tell the difference. Moreover, even when a given method does not toss coins directly, the thread scheduler does, and in some cases the appropriate moment to execute spec code depends on future scheduling decisions (§6.6). Speculation handles all of these cases with ease, but its validity rests on the weak observational power of clients.

The *idea* of speculation is not new: it is implicit in Lynch and Vaandrager's notion of *forward-backward simulation*,[29] and a form of it was even proposed

[28] This point is actually rather subtle: its validity rests on our definition of refinement, which simply observes the final answer a client context returns. See Chapter 7 for an in-depth discussion.

[29] Lynch and Vaandrager (1995), "Forward and Backward Simulations: Part I: Untimed Systems"

in Herlihy and Wing's original paper on linearizability[30] (although the proposal has not been used in subsequent formal logics for linearizability). What is new here is our particular treatment of speculation, which is designed for composability:[31]

- Previous simulation-based accounts of speculation apply it only at a *global* level, working monolithically over the entire specification state. Our treatment, by contrast, builds on our idea of specifications-as-resources (§4.4) which allows the specification state to be broken up into pieces that can be owned locally by threads or shared within a protocol. Speculation enriches these resources into *sets* of *pieces* of the specification. The combination of two sets of spec resources is then just the set of combined resources.

- Previous assertional accounts of speculation (*e.g.*, prophecy[32]) generally use "ghost" variables (also known as "auxiliary state") to linearize the branching structure of nondeterminism by recording, in advance, the outcome of future nondeterministic choices.[33] It is not clear how to generalize this approach to thread-local reasoning about a part of a program running in some unknown context.[34] Our treatment, by contrast, does *not* collapse the branching structure of the program being executed, but instead simply records the set of feasible spec executions that could coincide with it. Specification resources are crucial for enabling thread-local speculation, because they make it possible for a refinement proof for an implementation thread to consider only the corresponding spec thread to later be composed with refinement proofs for additional threads.

We give a more detailed comparison to related work in Chapter 7.

[30] Herlihy and Wing (1990), "Linearizability: a correctness condition for concurrent objects"

[31] Unfortunately, "composability" has several different meanings in this setting. Here we mean composability of refinements for parts of a program into refinements for their composition, as opposed to *transitive* composability of refinement, which previous accounts of speculation certainly supported.

[32] Abadi and Lamport (1991), "The existence of refinement mappings"

[33] Manolios (2003), "A compositional theory of refinement for branching time"

[34] Ley-Wild and Nanevski (2013), "Subjective Auxiliary State for Coarse-Grained Concurrency"

5

A logic for local protocols

"Logic takes care of itself; all we have to do is to look and see how it does it."

—Ludwig Wittgenstein

▸ Synopsis This chapter defines the syntax (§5.2) and semantics (§5.3 and §5.4) of a logic for refinement based on local protocols. The logic ties together a Kripke logical relation (traditionally used for showing refinement of one program by another) with Hoare triples (traditionally used for reasoning about a single program). The chapter sketches some proof theory for the logic (§5.5) and outlines a proof of soundness for refinement (§5.6). The full logic is summarized in Appendix B, and detailed proofs are given in Appendix C.

5.1 Overview

All the work we did in the last chapter rests on essentially two formal structures: transition systems and assertions. In this chapter, we deepen our understanding of these structures by:

1. Formalizing our transition systems and the syntax and semantics of assertions—and thereby formalizing local protocols, which are just transition systems with assertional interpretations. The result is a logic for local protocols.
2. Demonstrating that when assertions about refinement hold according to our semantics (which involves local protocols), the corresponding contextual refinement really does hold—*i.e.,* showing the soundness of the logic for its intended domain.
3. Sketching enough proof theory to illustrate the core reasoning principles for the logic.

In program logics for first-order languages, there is usually a strict separation between assertions about *data* (*e.g.,* heap assertions) and assertions about *code* (*e.g.,* Hoare triples). But the distinction makes less sense for higher-order languages, where code *is* data and hence claims about data must include claims about code. Our logic is therefore built around a single notion of assertion, P (shown in Figure 5.1), that plays several disparate roles. This uniformity places claims about resources, refinement, and Hoare triples on equal footing, which makes it easy to express a central idea of the logic: that refinement reasoning can be carried out using the combination of Hoare triples and specification resources.

We begin, in §5.2, with an informal tour of assertions.

Assertions	P	::=	$v = v$	Equality of values
		|	emp	Empty resource
$(\underline{\ell} ::= \ell \mid x)$		|	$\underline{\ell} \mapsto_{\mathrm{I}} u$	Singleton implementation heap
		|	$\underline{\ell} \mapsto_{\mathrm{S}} u$	Singleton specification heap
$(\underline{i} ::= i \mid x)$		|	$\underline{i} \rightarrowtail_{\mathrm{S}} e$	Singleton specification thread
		|	$\underline{i} \mapsto \iota$	Island assertion
		|	$P * P$	Separating conjunction
		|	$P \Rightarrow P$	Implication
		|	$P \wedge P$	Conjunction
		|	$P \vee P$	Disjunction
		|	$\exists x.P$	Existential quantification
		|	$\forall x.P$	Universal quantification
		|	$P \oplus P$	Speculative disjunction
		|	φ	Pure code assertion
		|	$\triangleright P$	Later modality
		|	$T@m\ \{x.\ P\}$	Threadpool simulation
Pure code assertions	φ	::=	$\{P\}\ e\ \{x.\ Q\}$	Hoare triple
		|	$v \preceq^{\mathcal{V}} v : \tau$	Value refinement
		|	$\Omega \vdash e \preceq^{\mathcal{E}} e : \tau$	Expression refinement
Island descriptions	ι	::=	(θ, I, s, A)	
			where $I \in \theta.S \to \mathrm{Assert}$,	State interpretation
			$s \in \theta.S$,	Current state (rely-lower-bound)
			$A \subseteq \theta.A$,	Owned tokens
			$A \# \theta.F(s)$	(which must not be free)
State transition systems	θ	::=	$(S, A, \rightsquigarrow, F)$	
			where S a set,	States
			A a set,	Tokens
			$\rightsquigarrow \subseteq S \times S$,	Transition relation
			$F \in S \to \wp(A)$	Free tokens
Main thread indicators	m	::=	i	ID of main thread
		|	none	No main thread

Figure 5.1: Syntax of assertions

5.2 ASSERTIONS

Assertions are best understood one role at a time.

5.2.1 *Characterizing the implementation heap*

The first role our assertions play is similar to that of heap assertions in separation logic: they capture knowledge about a part of the (implementation's) heap, *e.g.,* $x \mapsto_{\text{I}} 0$, and support the composition of such knowledge, *e.g.,* $x \mapsto_{\text{I}} 0 * y \mapsto_{\text{I}} 1$. In this capacity, assertions make claims contingent on the current state, which may be invalidated in a later state.

5.2.2 *Characterizing implementation code*

On the other hand, some assertions are *pure*, meaning that if they hold in a given state, they will hold in any possible future state. The syntactic subclass of *(pure) code assertions* φ all have this property, and they include *Hoare triples* $\{P\}\ e\ \{x.\ Q\}$. The Hoare triple says: for any future state satisfying P, if the (implementation) expression e is executed until it terminates with a result, the final state will satisfy Q (where x is the value e returned). So, for example, $\{\mathsf{emp}\}$ **new** $0\ \{x.\ x \mapsto_{\text{I}} 0\}$ is a *valid* assertion, *i.e.,* it holds in any state. More generally, the usual rules of separation logic apply, including the frame rule, the rule of consequence—and sequencing. The sequencing rule works, even in our concurrent setting, because heap assertions describe a portion of heap that is *privately owned* by the expression in the Hoare triple. In particular, that portion of the heap is guaranteed to be neither observed nor altered by threads concurrent with the expression.

5.2.3 *Characterizing (protocols on) shared resources*

The next role assertions play is expressing knowledge about *shared* resources. *All* shared resources are governed by a protocol. For hidden state, the protocol can be chosen freely, modulo proving that exported methods actually follow it. For visible state, however, *e.g.,* a reference that is returned directly to the context, the protocol is forced to be a trivial one—roughly, one that allows the state to take on any well-typed value at any time, accounting for the arbitrary interference an unknown context could cause (see Section 5.4).

Claims about shared resources are made through *island assertions* $i \mapsto \iota$ (inspired by LADR[1]). We call each shared collection of resources an *island*, because each collection is disjoint from all the others and is governed by an independent protocol. Each island has an identifier—a natural number—and in the island assertion $i \mapsto \iota$ the number i identifies a particular island described by ι.[2] The *island description* ι gives the protocol governing its resources, together with knowledge about the protocol's current state:

[1] Dreyer, Neis, Rossberg, *et al.* (2010), "A relational modal logic for higher-order stateful ADTs"

[2] We will often leave off the identifier as shorthand for an existential quantification, *i.e.,* when we treat ι as an assertion we mean $\exists x.\ x \mapsto \iota$.

- The component $\theta = (S, A, \rightsquigarrow, F)$ formalizes the STS for the protocol, where S is its set of states, A is its set of possible tokens, $\rightsquigarrow$ is its transition relation, and F is a function telling which tokens are free at each state.[3] The dynamics of STSs, *e.g.,* conservation of tokens, is formalized in the next section (§5.3).

- The component I tells how each state of the STS is *interpreted* as an assertion characterizing the concrete resources that are actually owned by the island in that state.[4]

- The components s and A express knowledge about the state of the protocol (which is at least s) from the perspective of a thread owning tokens A at that state. This knowledge is only a *lower bound* on the actual state of the protocol, which may in fact by in any "rely-future" state, *i.e.,* any state that can be reached from s by the thread's environment without using the thread's privately-owned tokens A.

[3] We will use dot notation like $\theta.S$ to project named components from compound objects.

[4] The assertion interpreting a state is assumed to not make claims about token ownership at *any* island, although it can make claims about the protocol or state of any island. The reason for this restriction is explained in §5.3.2.

5.2.4 *Characterizing refinement and spec resources*

Finally, assertions play two refinement-related roles.

The first is to express refinement itself, either between two closed values ($v_I \preceq^{\mathcal{V}} v_S : \tau$) or between open expressions ($\Omega \vdash e_I \preceq^{\mathcal{E}} e_S : \tau$)—a syntactic claim of semantic refinement ($\Omega \vDash e_I \preceq e_S : \tau$ in §3.3).

Until this point, we have avoided saying anything about spec terms, but in order to prove refinement we need to show that the observable behavior of an implementation can be mimicked by its spec. This brings us to an essential idea:

THE SLOGAN

By treating spec code as a resource, we can reduce refinement reasoning to Hoare-style reasoning.

THE MATH

$$e_I \preceq^{\mathcal{E}} e_S : \tau \quad \approx \quad \text{(roughly!)}$$
$$\forall j.\ \{j \rightarrowtail_S e_S\}\ e_I\ \{x_I.\ \exists x_S.\ x_I \preceq^{\mathcal{V}} x_S : \tau \wedge j \rightarrowtail_S x_S\}$$

Thus, the final role assertions play is to express knowledge about—and ownership of—spec resources, which include portions both of the heap (*e.g.,* $x \mapsto_S 0$) and of the threadpool (*e.g.,* $j \rightarrowtail_S e_S$). These resources can be shared and hence governed by protocols, just as implementation-side resources can.

When proving refinement for a data structure, we will prove something like the above Hoare triple for an arbitrary application of each of its methods—usually in the scope of an island assertion giving the protocol for its shared, hidden state. For each method invocation, we start from an arbitrary state of that protocol, and are *given ownership of* the spec code corresponding to the invocation, which we may choose to transfer to the protocol to support cooperation (as explained in Section 4.4). But in the end, when the

implementation's invocation has finished and returned a value x_I, we must have regained exclusive control over its spec, which must have mimicked it by producing a value x_S that x_I refines.

5.2.5 *The remaining miscellany*

The remaining forms of assertions include standard logical connectives, and two more technical forms of assertions—$\triangleright P$ and $T@m\ \{x.\ P\}$—which we explain in the Section 5.4.

5.3 SEMANTIC STRUCTURES

The semantics of assertions is given using two judgments, one for general assertions ($W, \eta \vDash^{\rho} P$) and the other for code assertions ($U \vDash^{\rho} \varphi$), where P and φ contain no free term variables but may contain free type variables bound by ρ. To explain these judgments, we begin with the semantic structures of *worlds* W, *resources* η and *environments* ρ, together with operations on them needed to interpret assertions.

5.3.1 *Resources*

The *resources* $\eta = (h, \Sigma)$ that assertions claim knowledge about and ownership of include both implementation heaps h, and speculative sets Σ of spec configurations,[5] as shown in Figure 5.2.

[5] Recall that a *configuration* $\varsigma = h; T$ consists of a heap and a threadpool.

Figure 5.2: Resources and their composition

▸ DOMAINS

$$\begin{array}{l} \text{StateSet} \triangleq \{\ \Sigma \subseteq \text{Heap} \times \text{ThreadPool} \mid \Sigma \text{ finite, nonempty}\ \} \\ \text{Resource} \triangleq \{\ \eta \in \text{Heap} \times \text{StateSet}\ \} \end{array}$$

▸ COMPOSITION *(These operations are partial)*

$$\begin{array}{lll} \text{State sets} & \Sigma_1 \otimes \Sigma_2 & \triangleq \{\ h_1 \uplus h_2; T_1 \uplus T_2 \mid h_i; T_i \in \Sigma_i\ \} \\ & & \quad \text{(if } \textit{all} \text{ compositions are defined)} \\ \text{Resources} & (h_1, \Sigma_1) \otimes (h_2, \Sigma_2) & \triangleq (h_1 \uplus h_2, \Sigma_1 \otimes \Sigma_2) \end{array}$$

Resources can be combined at every level, which is necessary for interpreting the $*$ operator on assertions:

- For heaps and threadpools, composition is done via $\uplus$, the usual disjoint union.
- The composition of state sets is just the set of state compositions—but it is only defined when *all* such state compositions are defined, so that speculative sets Σ have a single "footprint" consisting of all the locations/threads

existing in *any* speculative state.[6] To ensure that this footprint is finite, we require that speculation is itself finite.

- Composition of general resources is the composition of their parts.

[6] This is a rather technical point, but it is an important one. The point is that the speculative combination $\oplus$ should insist that *all* of its constituent assertions are satisfiable in combination with the rest of the current state.

5.3.2 *Islands and possible worlds*

All assertions are interpreted in the context of some *possible world* W, which contains a collection ω of islands. Both are defined in Figure 5.3.

▸ Domains

$$\begin{array}{lcl}
\text{Island}_n & \triangleq & \left\{ \iota = (\theta, J, s, A) \;\middle|\; \begin{array}{l} \theta \in \text{STS},\;\; s \in \theta.S,\;\; J \in \theta.S \to \text{UWorld}_n \overset{\text{mon}}{\to} \wp(\text{Resource}), \\ A \subseteq \theta.A,\;\; A \# \theta.F(s),\;\; J(s) \neq \varnothing \end{array} \right\} \\
\text{World}_n & \triangleq & \left\{ W = (k, \omega) \;\middle|\; k < n,\;\; \omega \in \mathbb{N} \overset{\text{fin}}{\rightharpoonup} \text{Island}_k \right\} \\
\text{UWorld}_n & \triangleq & \{\, U \in \text{World}_n \mid U = |U| \,\}
\end{array}$$

▸ Stripping tokens and decrementing step-indices

$$\begin{array}{lll}
|(\theta, J, s, A)| \triangleq (\theta, J, s, \varnothing) & & \lfloor (\theta, J, s_0, A) \rfloor_k \triangleq (\theta, \lambda s.J(s) \upharpoonright \text{UWorld}_k, s_0, A) \\
|(k, \omega)| \triangleq (k, \lambda i.|\omega(i)|) & & \triangleright(k+1, \omega) \triangleq (k, \lambda i.\lfloor \omega(i) \rfloor_k)
\end{array}$$

▸ Composition

$$\begin{array}{lll}
\text{Islands} & (\theta, J, s, A) \otimes (\theta', J', s', A') \triangleq (\theta, J, s, A \uplus A') & \text{when } \theta = \theta',\; s = s',\; J = J' \\
\text{Worlds} & (k, \omega) \otimes (k', \omega') \triangleq (k, \lambda i.\omega(i) \otimes \omega'(i)) & \text{when } k = k',\; \text{dom}(\omega) = \text{dom}(\omega')
\end{array}$$

Figure 5.3: Islands and worlds

Semantic islands look very much like syntactic island assertions—so much so that we use the same metavariable ι for both.[7] The only difference is that semantic islands interpret STS states semantically via J, rather than syntactically via I. Unfortunately, this creates a circularity: J is meant to interpret its syntactic counterpart I, and since assertions are interpreted in the contexts of worlds, the interpretation must be relative to the "current" world—but we are in the middle of *defining* worlds! The "step index" k in worlds is used to stratify away circularities in the definition of worlds and the logical relation; it and its attendant operators $\triangleright$ and $\lfloor - \rfloor_k$ are completely standard, but we briefly review the basic idea.[8]

The technique of step indexing was introduced by Appel and McAllester (2001)[9] and greatly expanded by Ahmed (2004). Boiled down, the idea is as follows. Programs exhibit or observe circularity one step of execution at a time. Therefore, for safety properties—which are inherently about finite execution (§2.5)—circularities can be broken by defining the properties relative to the number of execution steps remaining, *i.e.,* a "step index." Properties defined in this way are generally vacuous at step-index 0 (as, intuitively, there is no time left to make any observations), and serve as increasingly good approximations of the originally-intended property as the step index is increased. Following this strategy, we define worlds as a step-indexed family

[7] When the intended use matters, it will always be clear from context.

[8] We direct the interested reader to earlier work for a more in-depth explanation (Ahmed 2004; Dreyer, Neis, and Birkedal 2010; Dreyer, Neis, Rossberg, *et al.* 2010). The less interested reader should simply ignore step indices from here on.

[9] Similar techniques had already appeared for dealing with circularity in rely-guarantee (also called assume-guarantee) reasoning, *e.g.,* Abadi and Lamport (1995); Abadi and Lamport (1993).

of predicates. Each recursion through the world drives down the step index, the idea being: for the program to observe the actual heap described by the world, it must take at least one step. Put differently, we check conformance to protocols not in absolute terms, but rather for n steps of execution. Ultimately, if a program conforms for *every* choice of n, that is good enough for us.

With those ideas in mind, the definitions of $\triangleright$ and $\lfloor - \rfloor_k$ are straightforward. The "later" operator $\triangleright$ decrements the step-index of a world (assuming it was non-zero), *i.e.,* it constructs a version of the world as it will appear "one step later." Its definition relies on the auxiliary restriction operator $\lfloor - \rfloor_k$, which just throws away all data in the world at index larger than k (and thereby ensures that the resulting world is a member of World_k).

There is an additional subtlety with the definition of worlds: it is crucial that all participants in a protocol agree on the protocol's interpretation of a state, which must therefore be insensitive to which tokens a particular participant owns. We guarantee this by giving the interpretation J access to only the *unprivileged* part of a participant's world, $|W|$, which has been stripped of any tokens; see the constraint on the type of J. The monotonicity requirement $\overset{\text{mon}}{\to}$ is explained in §5.3.4.

Finally, to determine the meaning of assertions like $x \mapsto \iota * x \mapsto \iota'$, we must allow islands to be composed. Semantic island composition $\otimes$ is defined only when the islands agree on all aspects of the protocol, including its state; their owned tokens are then (disjointly) combined. Note, however, that because island *assertions* are rely-closed, an assertion like $x \mapsto \iota * x \mapsto \iota'$ does *not* require ι and ι' to assert the same state. It merely requires that there is some common state that is in both of their rely-futures. Worlds are composable only when they define the same islands and those islands are composable.

5.3.3 *Environments*

The terms that appear within assertions may include free type variables, which are interpreted by an *environment* ρ mapping them to relations

$$V \in \text{VRel}_n \triangleq \left\{ V \in \text{UWorld}_n \overset{\text{mon}}{\to} \wp(\text{Val} \times \text{Val}) \right\}$$

This interpretation of types is standard for logical relations, and in particular supports relational parametricity,[10] in which the interpretation of an abstract type may relate values of potentially different types on the implementation and specification sides.

We explain the monotonicity requirement $\overset{\text{mon}}{\to}$ below.

[10] Reynolds (1983), "Types, abstraction and parametric polymorphism"

5.3.4 *Protocol conformance*

The judgment $\theta \vdash (s, A) \rightsquigarrow (s', A')$ given in Figure 5.4 codifies the law of conservation of tokens (§4.3) for a single step.[11] We use this judgment in defining two relations governing changes to an island's state:

[11] We use the more readable notation $s \rightsquigarrow_\theta s'$ in place of $\theta.\rightsquigarrow(s, s')$.

▸ ISLAND AND WORLD FRAMING

$$\mathrm{frame}(\theta, J, s, A) \triangleq (\theta,\ J,\ s,\ \theta.A - \theta.F(s) - A) \qquad \mathrm{frame}(k, \omega) \triangleq (k,\ \lambda i.\mathrm{frame}(\omega(i)))$$

▸ PROTOCOL CONFORMANCE

Protocol step	$\theta \vdash (s, A) \rightsquigarrow (s', A')$	$\triangleq s \rightsquigarrow_\theta s',\quad \theta.F(s) \uplus A = \theta.F(s') \uplus A'$
Island guarantee move	$(\theta, J, s, A) \overset{\text{guar}}{\sqsubseteq} (\theta', J', s', A')$	$\triangleq \theta = \theta',\quad J = J',\quad \theta \vdash (s, A) \rightsquigarrow^* (s', A')$
Island rely move	$\iota \overset{\text{rely}}{\sqsubseteq} \iota'$	$\triangleq \mathrm{frame}(\iota) \overset{\text{guar}}{\sqsubseteq} \mathrm{frame}(\iota')$
World guarantee move	$(k, \omega) \overset{\text{guar}}{\sqsubseteq} (k', \omega')$	$\triangleq k \geq k',\quad \forall i \in \mathrm{dom}(\omega).\ \lfloor \omega(i) \rfloor_{k'} \overset{\text{guar}}{\sqsubseteq} \omega'(i)$
World rely move	$W \overset{\text{rely}}{\sqsubseteq} W'$	$\triangleq \mathrm{frame}(W) \overset{\text{guar}}{\sqsubseteq} \mathrm{frame}(W')$

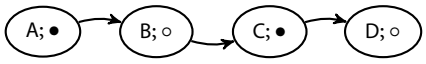
Figure 5.4: Protocol conformance

- The *guarantee* relation $\overset{\text{guar}}{\sqsubseteq}$, which characterizes the changes an expression can make to an island given the tokens it owns. A guarantee "move" may include changes to both the state of the STS and the privately-owned tokens, as the expression gains or loses tokens in making the move.
- An expression can likewise *rely* on its environment to only change an island ι according to the tokens that the environment owns, *i.e.,* the tokens in $\mathrm{frame}(\iota)$. The rely relation $\overset{\text{rely}}{\sqsubseteq}$ only allows the STS state to change. The tokens stay fixed because a rely move is performed by the *environment* of an expression, which gains or loses its own tokens but cannot affect the tokens owned by an expression.

IT IS CRUCIAL that the basic stepping judgment

$$\theta \vdash (s, A) \rightsquigarrow (s', A')$$

is defined over *single* steps in the transition system ($s \rightsquigarrow_\theta s'$), and only *afterwards* transitively-closed (in $\overset{\text{guar}}{\sqsubseteq}$). In other words, the transition relation itself is not required or considered to be transitively-closed. Consider, for example, the following STS:



The set of tokens here is just $\{\bullet\}$. Suppose the island is in state B and that we own the token, *i.e.,*

$$\iota = (\theta, J, \mathsf{B}, \{\bullet\})$$

where θ is the STS above and J is some interpretation. The expectation is that the environment is "stuck" with respect to the protocol: it would need to own the token to make progress, but *we* own the token. In other words, the protocol should guarantee that

$$\text{if} \quad \iota = (\theta, J, \mathsf{B}, \{\bullet\}) \quad \text{then} \quad \iota \overset{\text{rely}}{\sqsubseteq} \iota' \iff \iota = \iota'$$

If, however, the stepping relation was transitively closed *internally*, it would include an edge directly from B to D, thereby allowing the environment to make such a move without owning any tokens!

On the other hand, the fact that the rely and guarantee relations apply transitive closure *externally* allows threads to take multiple steps through the protocols as long as *each individual step* is permitted based on the conservation of tokens. So here, for example, our thread can move from B to D because it has enough tokens to first move to C.

There is thus an important case in which the basic transition system *is* effectively transitively closed: for any party that owns *all* of the non-free tokens. In such cases, the conservation law has no effect and the protocol degenerates to a simple STS in the style of Dreyer, Neis, and Birkedal (2010). In practice, the situation usually arises not because some thread owns all of the non-free tokens, but rather because it owns *none* of them—and so its environment (the frame) owns them all. In particular, for any *unprivileged* world U, the rely relation coincides with the (reflexive, transitive closure of) the raw transition relation.

▸ THE RELY AND GUARANTEE VIEWS OF A PROTOCOL give rise to *two* notions of future worlds. In both cases, the world may grow to include new islands, but any existing islands are constrained by their rely and guarantee relations, respectively. While the rely relation on worlds is defined using framing and the guarantee relation on worlds, it could have equivalently used the rely relation on islands instead:

$$(k, \omega) \overset{\text{rely}}{\sqsubseteq} (k', \omega') \quad = \quad k \geq k', \quad \forall i \in \mathrm{dom}(\omega).\ \lfloor \omega(i) \rfloor_{k'} \overset{\text{rely}}{\sqsubseteq} \omega'(i)$$

Island interpretations J are required to be monotone with respect to the rely relation on worlds (written $\overset{\text{mon}}{\to}$). Since the interpretations are applied to unprivileged worlds, the monotonicity requirement ensures that making *any* move in one island cannot possibly invalidate the interpretation of another.

5.3.5 *World satisfaction*

Worlds describe shared state abstractly, in terms of protocol states. Expressions, on the other hand, are executed against some *concrete* resources. The *world satisfaction relation* $\eta \ : \ W, \eta'$ defines when a given collection of concrete resources η "satisfies" a world,

$$\begin{aligned} \eta : (W, \eta') \ \triangleq \ & W.k > 0 \implies \\ & \eta = \eta' \otimes \overline{\eta_i},\ \forall i \in \mathrm{dom}(W.\omega).\ \eta_i \in \mathrm{interp}(W.\omega(i))(\triangleright |W|) \end{aligned}$$

meaning that η breaks into a disjoint portion for each island, with each portion satisfying its island's current interpretation. The parameter η' represents additional resources that are *private*, and therefore disjoint from those governed by the world—a notational convenience for defining the semantics of assertions below.

5.4 SEMANTICS

The semantics of assertions satisfies a fundamental closure property: if $W, \eta \models^\rho P$ and $W \overset{\text{rely}}{\sqsubseteq} W'$ then $W', \eta \models^\rho P$. All assertions are therefore "stable" under arbitrary interference from other threads. This should not be a surprise: assertions are either statements about private resources (for which interference is impossible) or about shared islands (for which interference is assumed, *e.g.,* we are careful to only assert *lower bounds* on the state of an island). The only subtlety is in the semantics of implication, which must be explicitly rely-closed to ensure stability.[12]

In general, the semantics of assertions is defined inductively using a lexicographic order: by step index (a component of the world), then by the syntactic structure of assertions. For interpreting refinement assertions, however, things are more complicated, as we explain in §5.4.2.

[12] This kind of closure is standard in logical relations, as well as Kripke-style interpretations of intuitionistic implication.

5.4.1 *Resources, protocols, and connectives*

The semantics of the basic assertions about private resources are straightforward, as are those for the basic logical connectives; see Figure 5.5.

R	$W, \eta \models^\rho R$ iff
φ	$\lvert W\rvert \models^\rho \varphi$
$v_1 = v_2$	$v_1 = v_2$
emp	$W = \lvert W\rvert, \eta = (\varnothing, \{\varnothing; \varnothing\})$
$P \wedge Q$	$W, \eta \models^\rho P$ and $W, \eta \models^\rho Q$
$P \vee Q$	$W, \eta \models^\rho P$ or $W, \eta \models^\rho Q$
$\forall x.P$	$\forall v.\ W, \eta \models^\rho P[v/x]$
$\exists x.P$	$\exists v.\ W, \eta \models^\rho P[v/x]$
$\triangleright P$	$W.k > 0 \implies \triangleright W, \eta \models^\rho P$

R	$W, \eta \models^\rho R$ iff
$P \Rightarrow Q$	$\forall W' \overset{\text{rely}}{\sqsupseteq} W.\ W', \eta \models^\rho P \implies W', \eta \models^\rho Q$
$\ell \mapsto_{\text{I}} u$	$\eta = ([\ell \mapsto u], \{\varnothing; \varnothing\})$
$\ell \mapsto_{\text{s}} u$	$\eta = (\varnothing, \{[\ell \mapsto u]; \varnothing\})$
$i \rightarrowtail_{\text{s}} e$	$\eta = (\varnothing, \{\varnothing; [i \mapsto e]\})$
$i \mapsto (\theta, I, s, A)$	$W.\omega(i) \overset{\text{rely}}{\sqsupseteq} (\theta, [\![I]\!], s, A)$ where $[\![I]\!] \triangleq \lambda s.\lambda U.\{\eta \mid U, \eta \models^\rho I(s)\}$
$P_1 * P_2$	$W = W_1 \otimes W_2,\ \eta = \eta_1 \otimes \eta_2,\ W_i, \eta_i \models^\rho P_i$
$P_1 \oplus P_2$	$\eta.\Sigma = \Sigma_1 \cup \Sigma_2,\ W, (\eta.h, \Sigma_i) \models^\rho P_i$

Figure 5.5: The semantics of resource and protocol assertions, and the connectives

There are just a few interesting points to take note of:

- The meaning island assertions is just as informally described: the protocol in the world differs only in giving a semantic state interpretation I, rather than a syntactic one J,[13] and the assertion's state s and tokens A together give only a *rely-lower-bound* on the true state of the island.
- While in general properties may be interesting even when the world's index is 0 (*i.e.,* $W.k = 0$), the later modality $\triangleright P$ is defined to be vacuously true in such cases.[14]
- In a speculative choice $P \oplus Q$, the two choices P and Q are understood in the same world and under the same implementation resources, but they

[13] There is a rather subtle point here: we assume that the denotation function $[\![I]\!]$ constructs a semantic $\text{Island}_{W.k}$, *i.e.,* that the argument U is drawn from $\text{UWorld}_{W.k}$. This ensures the well-foundedness of assertion semantics.

[14] This aspect of step-indexing makes it highly reminiscent of coinduction.See Hur *et al.* (2012) for a recent investigation of the connections.

must each be satisfied by a (nonempty, possibly nondisjoint) subset of the set of spec states Σ. The lack of disjointness makes speculative choice quite different from separating conjunction; in particular, it is idempotent, *i.e.*, $P \oplus P$ is equivalent to P.

5.4.2 *Refinement*

The value refinement assertion $v_{\text{I}} \preceq^{\mathcal{V}} v_{\text{S}} : \tau$ requires that any observations a context can make of v_{I} at type τ can also be made of v_{S}, as evidenced by its semantics in Figure 5.6.

τ_0	v_{I}	v_{S}	$U \vDash^{\rho} v_{\text{I}} \preceq^{\mathcal{V}} v_{\text{S}} : \tau_0$ iff
τ_b	v	v	$\vdash v : \tau_b$ for $\tau_b \in \{\textbf{unit}, \textbf{bool}, \textbf{nat}\}$
α	v_{I}	v_{S}	$(v_{\text{I}}, v_{\text{S}}) \in \rho(\alpha)(U)$
$\tau_1 \times \tau_2$	$(v_1^{\text{I}}, v_2^{\text{I}})$	$(v_1^{\text{S}}, v_2^{\text{S}})$	$U \vDash^{\rho} \triangleright(v_1^{\text{I}} \preceq^{\mathcal{V}} v_1^{\text{S}} : \tau_1 \ \wedge \ v_2^{\text{I}} \preceq^{\mathcal{V}} v_2^{\text{S}} : \tau_2)$
$\tau \to \tau'$	$\textbf{rec}\ f\ x.e_{\text{I}}$	$\textbf{rec}\ f\ x.e_{\text{S}}$	$U \vDash^{\rho} \triangleright(x : \tau \vdash e_{\text{I}}[v_{\text{I}}/f] \preceq^{\mathcal{E}} e_{\text{S}}[v_{\text{S}}/f] : \tau')$
$\forall \alpha.\tau$	$\Lambda.e_{\text{I}}$	$\Lambda.e_{\text{S}}$	$U \vDash^{\rho} \triangleright(\alpha \vdash e_{\text{I}} \preceq^{\mathcal{E}} e_{\text{S}} : \tau)$
$\mu\alpha.\tau$	v_{I}	v_{S}	$U \vDash^{\rho} v_{\text{I}} \preceq^{\mathcal{V}} v_{\text{S}} : \tau[\mu\alpha.\tau/\alpha]$
$\textbf{ref}_?(\overline{\tau})$	$\textbf{null}$	$\textbf{null}$	always
	ℓ_{I}	ℓ_{S}	$U \vDash^{\rho} \triangleright \ell_{\text{I}} \preceq^{\mathcal{V}} \ell_{\text{S}} : \textbf{ref}(\overline{\tau})$
$\textbf{ref}(\overline{\tau})$	ℓ_{I}	ℓ_{S}	$U \vDash^{\rho} \text{inv}(\exists \overline{x}, \overline{y}.\ \left(\wedge \overline{x \preceq^{\mathcal{V}} y : \tau}\right) \wedge \ell_{\text{I}} \mapsto_{\text{I}} (\overline{x}) * \ell_{\text{S}} \mapsto_{\text{S}} (\overline{y}))$
$\tau_1 + \tau_2$	ℓ_{I}	ℓ_{S}	$\exists i.\ U \vDash^{\rho} \exists x, y.\ \triangleright x \preceq^{\mathcal{V}} y : \tau_i \wedge \text{inv}(v_{\text{I}} \mapsto_{\text{I}} \textbf{inj}_i\ x * v_{\text{S}} \mapsto_{\text{S}} \textbf{inj}_i\ y)$

where $\text{inv}(P) \triangleq (((\{\text{dummy}\}, \varnothing, \varnothing, \lambda\_.\varnothing), \lambda\_.P, \text{dummy}, \varnothing)$

Figure 5.6: The semantics of value refinement

Those readers familiar with Kripke logical relations will recognize the semantics of value refinement as essentially the standard definition of logical approximation between values. Base type values must be identical. For function types, we check that the bodies of the functions are related when given related arguments,[15] which, due to the semantics of implication, might happen in a rely-future world. For recursive types, we check that the values are related at the unfolded type (we explain why this is well-founded below). Values that are exposed to the context at heap-allocated type—**ref** and sum types—are forced to be governed by a trivial island[16] allowing *all* type-safe updates (in the case of **ref**'s) and *no* updates (in the case of sums). Hidden state, on the other hand, is by definition state that does not escape directly to the context, and so we need say nothing about it for value refinement.

To see why the definition of value refinement is not circular, we must take note of several facts. First, almost every mention of value refinement on the right-hand side of the definition is within either a $\triangleright$ or an inv, with the sole exception of recursive types. Both $\triangleright$ and inv have the effect of

[15] The quantification over related arguments is performed within expression refinement, Figure 5.7

[16] The "invariant island" $\text{inv}(P)$ for an assertion P that we use to constrain exposed heap data is just a formalization of the "DummyState" protocol discussed in the Overview of Chapter 4.

strictly decreasing the step index when interpreting the assertion within them (recall that semantic island interpretations are given at strictly smaller step indexes, to avoid circularity); decreasing step indexes provide a well-founded measure. For recursive types the argument is a bit more subtle: it relies on the F^{μ}_{cas} requirement that all recursive types be productive, meaning that a recursive type variable can only appear under a non-μ type constructor. That means, in particular, that a recursive type must always consist of some finite outer nest of μ-bindings, followed *immediately* by a non-μ, non-variable type constructor. As we just argued, the interpretation of all such type constructors drives the step index down.

The fact that refinement is a "pure" assertion, *i.e.,* insensitive to the state of private resources or the ownership of private tokens, is essential for soundness. The reason is simple: once a value has reached the context, it can be copied and used concurrently. We therefore cannot claim that any *one* copy of the value privately owns some resources. If P is impure, we say

$$U \vDash^{\rho} P \qquad \triangleq \qquad \forall \eta, W \# U.\ W, \eta \vDash^{\rho} P$$

where # as usual means that the composition $W \otimes U$ is defined.[17]

[17] For any $W \# U$ we have $W \otimes U = W$.

Ω	$U \vDash^{\rho} \Omega \vdash e_{\mathrm{I}} \preceq^{\mathcal{E}} e_{\mathrm{S}} : \tau$ iff
$\cdot$	$\forall K, j.\ U \vDash^{\rho} \{j \rightarrowtail_{\mathrm{S}} K[e_{\mathrm{S}}]\}\ e_{\mathrm{I}}\ \{x.\ \exists y.\ x \preceq^{\mathcal{V}} y : \tau \wedge j \rightarrowtail_{\mathrm{S}} K[y]\}$
$x : \tau', \Omega'$	$\forall v_{\mathrm{I}}, v_{\mathrm{S}}.\ U \vDash^{\rho} v_{\mathrm{I}} \preceq^{\mathcal{V}} v_{\mathrm{S}} : \tau' \Rightarrow \Omega' \vdash e_{\mathrm{I}}[v_{\mathrm{I}}/x] \preceq^{\mathcal{E}} e_{\mathrm{S}}[v_{\mathrm{S}}/x] : \tau$
α, Ω'	$\forall V.\ U \vDash^{\rho[\alpha \mapsto V]} \Omega' \vdash e_{\mathrm{I}} \preceq^{\mathcal{E}} e_{\mathrm{S}} : \tau$

Figure 5.7: The semantics of expression refinement

For expression refinement $\Omega \vdash e_{\mathrm{I}} \preceq^{\mathcal{E}} e_{\mathrm{S}} : \tau$, shown in Figure 5.7, we first close off any term or type variables bound by Ω with the appropriate universal quantification. Closed expression refinement is then defined in terms of a Hoare triple, following our sketch in §5.2.4. The main difference in the actual definition is that we additionally quantify over the unknown evaluation context K in which a specification is running; this annoyance appears to be necessary for proving that refinement is a precongruence. *Important note*: this is *not* the same kind of quantified contextual property that we started with (*i.e.,* in the definition of contextual refinement). In particular, it is *not even possible* to examine the context we are given. Rather, it is a way of forcing refinement proofs to use exactly the part of the spec expression involved in the refinement, leaving any surrounding evaluation context intact.

5.4.3 *Hoare triples and threadpool simulation*

Hoare triples are defined via the *threadpool simulation* assertion $T@m\ \{x.\ Q\}$, which is the engine that powers our model:

$$U \vDash^{\rho} \{P\}\ e\ \{x.\ Q\} \quad \triangleq \quad \forall i.\ U \vDash^{\rho} P \Rightarrow [i \mapsto e]@i\ \{x.\ Q\}$$

Threadpool simulation (Figure 5.8) accounts for the fact that an expression can fork threads as it executes, but that we care about the return value only from some "main" thread m, which is the initial thread i here.

$$W_0, \eta \vDash^\rho T@m\ \{x.\ Q\} \triangleq \forall W \overset{\text{rely}}{\sqsupseteq} W_0,\ \eta_F \# \eta. \quad \text{if } W.k > 0 \text{ and } h, \Sigma : W, \eta \otimes \eta_F \text{ then:}$$
$$h; T \to h'; T' \implies \exists \Sigma', \eta', W' \overset{\text{guar}}{\exists_1} W.\ \Sigma \rightrightarrows \Sigma',\ \ h', \Sigma' : W', \eta' \otimes \eta_F,\ \ W', \eta' \vDash^\rho T'@m\ \{x.\ Q\}$$
$$T = T_0 \uplus [m \mapsto v] \implies \exists \Sigma', \eta', W' \overset{\text{guar}}{\exists_0} W.\ \Sigma \rightrightarrows \Sigma',\ \ h, \Sigma' : W', \eta' \otimes \eta_F,\ \ W', \eta' \vDash^\rho Q[v/x] * T_0@\mathsf{none}\ \{x.\ \mathsf{tt}\}$$

$$\text{where } W' \overset{\text{guar}}{\exists_n} W \triangleq W' \overset{\text{guar}}{\sqsupseteq} W \wedge W.k = W'.k + n$$
$$\Sigma \rightrightarrows \Sigma' \triangleq \forall \varsigma' \in \Sigma'.\ \exists \varsigma \in \Sigma.\ \varsigma \to^* \varsigma'$$

Figure 5.8: Threadpool simulation

To satisfy $T@m\ \{x.\ Q\}$ at some W and η, the threads in T must first of all continuously obey the protocols of W, assuming they share private ownership of η. That is, every atomic step taken by a thread in T must:

- transform its shared resources in a way that corresponds to a guarantee move in the protocol ($W' \overset{\text{guar}}{\sqsupseteq} W$), and
- preserve as a frame any private resources η_F of its environment;
- but it may change private resources η in any way it likes.

In between each such atomic step, the context might get a chance to run, which we model by quantifying over an arbitrary rely-future world.[18] If at any point the main thread m terminates, it must do so in a state satisfying the postcondition Q, where x is bound to the value the main thread returned. Afterwards, any lingering threads are still required to obey the protocol using the remaining resources, but the main thread identifier is replaced by none.

That threadpool simulation is, in fact, a simulation is due to its use of the *speculative stepping relation* $\Sigma \rightrightarrows \Sigma'$, which requires any changes to the spec state to represent feasible execution steps: every new state must be reachable from some old state, but we are free to introduce multiple new states originating in the same old state, and we are free to drop irrelevant old states on the floor. As a result of how simulation is defined, such changes to the spec state can only be made to those parts that are under the threadpool's control, either as part of its private resources (allowing arbitrary feasible updates) or its shared ones (allowing only protocol-permitted updates).

[18] Recall the discussion in the Overview of Chapter 4: this is how we model the behavior of an arbitrary client while making a *single* pass through the implementation code.

5.5 BASIC REASONING PRINCIPLES

"A formal manipulator in mathematics often experiences the discomforting feeling that his pencil surpasses him in intelligence."

—Howard Eves

Although we will not pursue an in-depth study of proof theory for the logic of local protocols,[19] in this section we sketch some of the basic reasoning principles supported by the logic.

[19] We leave this to future work, and expect that some choices in the design of assertions will need to change to accommodate a clean proof theory; see Chapter 13.

5.5.1 *Hypothetical reasoning and basic logical rules*

Most of the inference rules will be presented in hypothetical style, *e.g.*,

$$\frac{\mathcal{P} \vdash P \qquad \mathcal{P} \vdash Q}{\mathcal{P} \vdash P \wedge Q}$$

The metavariable $\mathcal{P}$ ranges over hypotheses,[20]

$$\mathcal{P} \quad ::= \quad \cdot \mid \mathcal{P}, P$$

giving rise to the following semantic interpretations,

$$\begin{aligned} W, \eta \vDash^{\rho} \mathcal{P} &\triangleq \forall P \in \mathcal{P}.\ W, \eta \vDash^{\rho} P \\ \mathcal{P} \vDash Q &\triangleq \forall W, \eta, \rho, \gamma.\ W, \eta \vDash^{\rho} \gamma\mathcal{P} \implies W, \eta \vDash^{\rho} \gamma Q \end{aligned}$$

where γ ranges over variable-to-value substitutions and ρ and γ are constrained to close both $\mathcal{P}$ and Q. Thus the soundness of the rule above means that the following implication holds:

$$\frac{\mathcal{P} \vDash P \qquad \mathcal{P} \vDash Q}{\mathcal{P} \vDash P \wedge Q} \quad \textit{i.e.,} \quad \mathcal{P} \vDash P \text{ and } \mathcal{P} \vDash Q \text{ implies } \mathcal{P} \vDash P \wedge Q$$

With those preliminaries in place, we give in Figure 5.9 the basic laws for intuitionistic first-order logic and for separating conjunction.[21] These laws are easy to prove, either directly or through appeal to standard model-theoretic arguments. Since they are standard, we do not discuss them further here.

In addition, we include the fundamental laws governing the "later" modality $\triangleright P$. The first is a monotonicity law saying that anything true at the current step index will remain true at a smaller index; after all, decreasing step indices represent decreasing observational power on the part of programs. Step indexing also gives rise to proofs with a coinductive flavor via the Löb rule,[22] which makes it possible to prove P while assuming that it holds one step later. We will see in §5.5.7 how the Löb rule supports reasoning about recursion.

5.5.2 *Reasoning about programs: an overview*

Program reasoning works in three layers:

- The top layer is refinement, which often serves as the end-goal of a proof. The proof rules for introducing refinements are just reformulations of the semantics of refinement.[23] In particular, expression refinement requires proving a Hoare triple, which we do using lower-level concurrent Hoare logic.
- The middle layer is "concurrent Hoare logic," in which we prove Hoare triples $\{P\}\ \ e\ \ \{x.Q\}$. The "concurrent" nature of this logical layer is

[20] Despite the presence of separating conjunction, we keep things simple here and do not introduce bunched contexts (O'Hearn and Pym 1999), instead including a set of axioms for separating conjunction. This is one of the reasons the proof theory we are presenting is just a sketch.

[21] O'Hearn and Pym (1999), "The logic of bunched implications"

[22] Appel *et al.* (2007), "A very modal model of a modern, major, general type system"

[23] This should not be surprising: as it is, refinement is a thin veneer over the rest of the logic, and could be treated entirely as a derived form if we moved to a second order (relational) logic (like Plotkin and Abadi (1993), Dreyer *et al.* (2009), or Dreyer, Neis, Rossberg, *et al.* (2010).

▸ LAWS OF INTUITIONISTIC FIRST-ORDER LOGIC.

$$\frac{P \in \mathcal{P}}{\mathcal{P} \vdash P} \qquad \frac{\mathcal{P} \vdash P[v/x] \qquad \mathcal{P} \vdash v = v'}{\mathcal{P} \vdash P[v'/x]} \qquad \frac{\mathcal{P} \vdash P \qquad \mathcal{P} \vdash Q}{\mathcal{P} \vdash P \land Q} \qquad \frac{\mathcal{P} \vdash P \land Q}{\mathcal{P} \vdash P} \qquad \frac{\mathcal{P} \vdash P \land Q}{\mathcal{P} \vdash Q}$$

$$\frac{\mathcal{P} \vdash P \lor Q \qquad \mathcal{P}, P \vdash R \qquad \mathcal{P}, Q \vdash R}{\mathcal{P} \vdash R} \qquad \frac{\mathcal{P} \vdash P}{\mathcal{P} \vdash P \lor Q} \qquad \frac{\mathcal{P} \vdash Q}{\mathcal{P} \vdash P \lor Q} \qquad \frac{\mathcal{P}, P \vdash Q}{\mathcal{P} \vdash P \Rightarrow Q} \qquad \frac{\mathcal{P} \vdash P \Rightarrow Q \qquad \mathcal{P} \vdash P}{\mathcal{P} \vdash Q}$$

$$\frac{\mathcal{P} \vdash P[y/x] \qquad y \text{ fresh}}{\mathcal{P} \vdash \forall x.P} \qquad \frac{\mathcal{P} \vdash \forall x.P}{\mathcal{P} \vdash P[v/x]} \qquad \frac{\mathcal{P} \vdash \exists x.P \qquad \mathcal{P}, P[y/x] \vdash Q \qquad y \text{ fresh}}{\mathcal{P} \vdash Q} \qquad \frac{\mathcal{P} \vdash P[v/x]}{\mathcal{P} \vdash \exists x.P}$$

▸ AXIOMS FROM THE LOGIC OF BUNCHED IMPLICATIONS.

$$\begin{aligned} P * Q &\iff Q * P \\ (P * Q) * R &\iff P * (Q * R) \\ P * \mathsf{emp} &\iff P \end{aligned}$$

$$\begin{aligned} (P \lor Q) * R &\iff (P * R) \lor (Q * R) \\ (P \land Q) * R &\implies (P * R) \land (Q * R) \\ (\exists x.\, P) * Q &\iff \exists x.\, (P * Q) \\ (\forall x.\, P) * Q &\implies \forall x.\, (P * Q) \end{aligned}$$

$$\frac{\mathcal{P}, P_1 \vdash Q_1 \qquad \mathcal{P}, P_2 \vdash Q_2}{\mathcal{P}, P_1 * P_2 \vdash Q_1 * Q_2}$$

▸ LAWS FOR THE "LATER" MODALITY.

$$\text{MONO}\ \frac{\mathcal{P} \vdash P}{\mathcal{P} \vdash \triangleright P} \qquad \text{LÖB}\ \frac{\mathcal{P}, \triangleright P \vdash P}{\mathcal{P} \vdash P}$$

$$\begin{aligned} \triangleright(P \land Q) &\iff \triangleright P \land \triangleright Q \\ \triangleright(P \lor Q) &\iff \triangleright P \lor \triangleright Q \end{aligned}$$

$$\begin{aligned} \triangleright \forall x.P &\iff \forall x.\triangleright P \\ \triangleright \exists x.P &\iff \exists x.\triangleright P \\ \triangleright(P * Q) &\iff \triangleright P * \triangleright Q \end{aligned}$$

Figure 5.9: The basic logical laws

reflected in the fact that all of the heap assertions in pre- and post-conditions are understood to characterize thread-private state, while all claims about shared state are made through island assertions. That means that *all* assertions are automatically "stable" under concurrent interference. Concurrent Hoare logic is used primarily to glue together the results of reasoning in the lower-level atomic Hoare logic.

- THE BOTTOM LAYER is "atomic Hoare logic," which is used to reason about atomic steps of execution *without regard to concurrent threads*. Atomic Hoare logic uses an alternative, "atomic" Hoare triple $\langle\!|P|\!\rangle\ a\ \langle\!|x.Q|\!\rangle$ in which pre- and post-conditions may characterize private *and* shared state alike, in terms of *concrete* heap/code resources rather than through island assertions. Atomic triples are restricted to the following atomic expressions, whose execution is guaranteed to take exactly one step:

  $$a \ ::= \ \mathsf{new}\ \overline{v} \mid \mathsf{get}(v[i]) \mid v[i] := v \mid \mathsf{cas}(v[i], v, v) \mid \mathsf{inj}_i\ v$$

  Since the pre- and post-conditions of atomic triples are not stable under concurrent interference, atomic Hoare logic does not provide rules for sequencing. Instead, atomic triples must be lifted to concurrent triples, which requires showing that any changes made to a shared resource must be permitted by the protocol governing it. Since protocols govern

execution one atomic step at a time, the restriction of atomic triples to atomic expressions is a vital one.

When we say "Hoare triple" or "Hoare-style reasoning" without qualification, we mean "concurrent Hoare triple" and "concurrent Hoare logic."

Recall that the semantics of refinement and Hoare triples is given with respect to an unprivileged world U and without reference to any private resources (§5.4). Consequently, the rules for program reasoning uses a restricted form of the hypothetical style: instead of an arbitrary set of hypothesis $\mathcal{P}$, the rules use a set of "pure" hypothesis Φ. A hypothesis P is *pure* if its meaning is insensitive to the ownership of tokens or private resources, *i.e.,* if for every W, η and ρ

$$W, \eta \vDash^{\rho} P \quad \iff \quad |W|, \varnothing \vDash^{\rho} P$$

We next elaborate on each layer of program reasoning, working top-down.

5.5.3 *Reasoning about refinement*

Since there are two kinds of refinement—one between values, one between expressions—we begin with a rule that relates them:

$$\frac{\Phi \vdash v_1 \preceq^{\mathcal{V}} v_2 : \tau}{\Phi \vdash v_1 \preceq^{\mathcal{E}} v_2 : \tau}$$

The rule shows that expression refinement contains value refinement. As it turns out, this rule will be derivable from the following one (SpecIntro), together with the rule Return in §5.5.4:

SpecIntro
$$\frac{\forall K, j.\ \Phi \vdash \{j \mapsto_{\mathrm{s}} K[e_{\mathrm{s}}]\}\ e_{\mathrm{I}}\ \{x.\ \exists y.\ x \preceq^{\mathcal{V}} y : \tau \wedge j \rightarrowtail_{\mathrm{s}} K[y]\}}{\Phi \vdash e_{\mathrm{I}} \preceq^{\mathcal{E}} e_{\mathrm{s}} : \tau}$$

SpecIntro rule merely restates the definition of (closed) expression refinement in terms of Hoare triples (§5.2). We omit the other rules recapitulating the definition of (open) expression refinement.

The rules in Figure 5.10 for introducing value refinement are also just recapitulations of its definition; there are also a set of rules in the other direction for eliminating refinements, which we omit. We also omit rules for type variables; see Chapter 13. Since the rules are so closely correlated with the types of the language, we don't bother to separately name them.

5.5.4 *Concurrent Hoare logic*

The "glue" rules for concurrent Hoare logic, shown in Figure 5.11, are completely straightforward. Since we are working in an expression-oriented rather than statement-oriented language, we have a Bind rule for connecting

$$\Phi \vdash () \preceq^{\mathcal{V}} () : \mathbf{unit} \qquad \Phi \vdash \mathbf{true} \preceq^{\mathcal{V}} \mathbf{true} : \mathbf{bool} \qquad \Phi \vdash \mathbf{false} \preceq^{\mathcal{V}} \mathbf{false} : \mathbf{bool} \qquad \Phi \vdash n \preceq^{\mathcal{V}} n : \mathbf{nat}$$

$$\frac{\Phi \vdash \triangleright v_1^{\mathrm{I}} \preceq^{\mathcal{V}} v_1^{\mathrm{S}} : \tau_1 \qquad \Phi \vdash \triangleright v_2^{\mathrm{I}} \preceq^{\mathcal{V}} v_2^{\mathrm{S}} : \tau_2}{\Phi \vdash (v_1^{\mathrm{I}}, v_2^{\mathrm{I}}) \preceq^{\mathcal{V}} (v_1^{\mathrm{S}}, v_2^{\mathrm{S}}) : \tau_1 \times \tau_2} \qquad \frac{v_{\mathrm{I}} = \mathbf{rec}\ f(x_{\mathrm{I}}).e_{\mathrm{I}} \qquad v_{\mathrm{S}} = \mathbf{rec}\ f(x_{\mathrm{S}}).e_{\mathrm{S}} \qquad \Phi, x_{\mathrm{I}} \preceq^{\mathcal{V}} x_{\mathrm{S}} : \tau \vdash \triangleright e_{\mathrm{I}}[v_{\mathrm{I}}/f] \preceq^{\mathcal{E}} e_{\mathrm{S}}[v_{\mathrm{S}}/f] : \tau'}{\Phi \vdash v_{\mathrm{I}} \preceq^{\mathcal{V}} v_{\mathrm{S}} : \tau \to \tau'}$$

$$\frac{\Phi \vdash \triangleright e_{\mathrm{I}} \preceq^{\mathcal{E}} e_{\mathrm{S}} : \tau}{\Phi \vdash \Lambda.e_{\mathrm{I}} \preceq^{\mathcal{V}} \Lambda.e_{\mathrm{S}} : \forall \alpha.\tau} \qquad \frac{\Phi \vdash v_{\mathrm{I}} \preceq^{\mathcal{V}} v_{\mathrm{S}} : \tau[\mu\alpha.\tau/\alpha]}{\Phi \vdash v_{\mathrm{I}} \preceq^{\mathcal{V}} v_{\mathrm{S}} : \mu\alpha.\tau} \qquad \Phi \vdash \mathbf{null} \preceq^{\mathcal{V}} \mathbf{null} : \mathbf{ref}_?(\overline{\tau}) \qquad \frac{\Phi \vdash \triangleright v_{\mathrm{I}} \preceq^{\mathcal{V}} v_{\mathrm{S}} : \mathbf{ref}(\overline{\tau})}{\Phi \vdash v_{\mathrm{I}} \preceq^{\mathcal{V}} v_{\mathrm{S}} : \mathbf{ref}_?(\overline{\tau})}$$

$$\frac{\Phi \vdash \mathrm{inv}\left(\exists \overline{x}, \overline{y}.\ \bigwedge \overline{x \preceq^{\mathcal{V}} y : \tau} \wedge v_{\mathrm{I}} \mapsto_{\mathrm{I}} (\overline{x}) * v_{\mathrm{S}} \mapsto_{\mathrm{S}} (\overline{y})\right)}{\Phi \vdash v_{\mathrm{I}} \preceq^{\mathcal{V}} v_{\mathrm{S}} : \mathbf{ref}(\overline{\tau})} \qquad \frac{\Phi \vdash \exists x, y.\ \triangleright x \preceq^{\mathcal{V}} y : \tau_i \wedge \mathrm{inv}(v_{\mathrm{I}} \mapsto_{\mathrm{I}} \mathbf{inj}_i\ x * v_{\mathrm{S}} \mapsto_{\mathrm{S}} \mathbf{inj}_i\ y)}{\Phi \vdash v_{\mathrm{I}} \preceq^{\mathcal{V}} v_{\mathrm{S}} : \tau_1 + \tau_2}$$

Figure 5.10: Introduction rules for value refinement

an expression to an evaluation context (rather than a sequencing rule for statements).[24] The only other unusual feature is that we funnel all use of hypotheses through the rule Hypo, which allows them to be brought into the precondition of a triple. (A kind of converse rule, HypoOut, allows pure assertions to be brought out of the purview of Hoare logic and into the general proof context.)

The "primitive" rules shown in the bottom of Figure 5.11 are more interesting: as the name suggests, they provide the means of proving triples about primitive operations. Primitives fall into two categories: pure and imperative.

Pure operations (*e.g.*, addition, conditionals and function application) neither inspect nor alter the heap. As such, they have only one interaction with Hoare-style reasoning: they decrease the number of steps remaining. The rule Pure supports reasoning about pure steps of computation: if $e \overset{\text{pure}}{\hookrightarrow} e'$ (note the lack of heap),[25] and e' satisfies a particular Hoare triple, then e satisfies the same Hoare triple one step earlier.

Imperative operations—all those in the grammar of atomic expressions a (§5.5.2)—interact with the heap in some way, which requires us to take concurrency into account. Consequently, we have two basic rules for reasoning about imperative operations, depending on whether they are applied to private or shared resources:

- Private resources must be mentioned explicitly and concretely in the pre- and post-conditions of a concurrent Hoare triple. Atomic Hoare triples $\langle\!|P|\!\rangle\ \ a\ \ \langle\!|x.Q|\!\rangle$, as we will see shortly, operate on precisely such concrete assertions. Therefore the Private rule simply lifts atomic Hoare reasoning.[26] Additional, potentially-shared resources can then be framed in using Frame.

[24] Of course, statement-style sequencing is a derived form of **let**, and likewise the standard Hoare-style sequencing rule is derivable from Bind.

[25] This auxiliary stepping relation is given in Appendix A.

[26] The use of $\triangleright$ is justified by the fact that an atomic expression always takes one step to execute.

▸ "Glue" (logical and structural) rules for concurrent Hoare logic.

$$\textsc{Bind}\ \frac{\{P\}\ e\ \{x.\ Q\} \qquad \forall x.\ \{Q\}\ K[x]\ \{y.\ R\}}{\{P\}\ K[e]\ \{y.\ R\}} \qquad \textsc{Return}\ \frac{}{\{\mathsf{emp}\}\ v\ \{x.\ x = v \land \mathsf{emp}\}}$$

$$\textsc{Consequence}\ \frac{P \vdash P' \qquad \{P'\}\ e\ \{x.\ Q'\} \qquad Q' \vdash Q}{\{P\}\ e\ \{x.\ Q\}} \qquad \textsc{Disjunction}\ \frac{\{P_1\}\ e\ \{x.\ Q\} \qquad \{P_2\}\ e\ \{x.\ Q\}}{\{P_1 \lor P_2\}\ e\ \{x.\ Q\}} \qquad \textsc{Frame}\ \frac{\{P\}\ e\ \{x.\ Q\}}{\{P * R\}\ e\ \{x.\ Q * R\}}$$

$$\textsc{Hypo}\ \frac{\Phi \vdash P \qquad \{P \land Q\}\ e\ \{x.\ R\}}{\Phi \vdash \{Q\}\ e\ \{x.\ R\}} \qquad \textsc{HypoOut}\ \frac{P\ \text{pure} \qquad \Phi, P \vdash \{Q\}\ e\ \{x.\ R\}}{\Phi \vdash \{P \land Q\}\ e\ \{x.\ R\}} \qquad \textsc{Weaken}\ \frac{\Phi \vdash \{P\}\ e\ \{x.\ Q * R\}}{\Phi \vdash \{P\}\ e\ \{x.\ Q\}}$$

▸ Primitive rules for concurrent Hoare logic.

$$\textsc{Pure}\ \frac{e \overset{\text{pure}}{\hookrightarrow} e' \qquad \{P\}\ e'\ \{Q\}}{\{\triangleright P\}\ e\ \{Q\}} \qquad \textsc{Private}\ \frac{(\!|P|\!)\ a\ (\!|x.\ Q|\!)}{\{\triangleright P\}\ a\ \{x.\ Q\}}$$

$$\textsc{Shared}\ \frac{\forall \iota \overset{\text{rely}}{\sqsupseteq} \iota_0.\ \exists \iota' \overset{\text{guar}}{\sqsupseteq} \iota.\ \exists Q.\ (\!|\iota.I(\iota.s) * P|\!)\ a\ (\!|x.\ \triangleright \iota'.I(\iota'.s) * Q|\!) \land (\underline{i} \mapsto \iota' * Q) \vdash R}{\{\underline{i} \mapsto \iota_0 * \triangleright P\}\ a\ \{x.\ R\}} \qquad \textsc{NewIsland}\ \frac{\{P\}\ e\ \{x.\ Q * \triangleright \iota.I(\iota.s)\}}{\{P\}\ e\ \{x.\ Q * \iota\}}$$

$$\textsc{PrivateSumElim}\ \frac{i \in \{1,2\} \qquad \{\underline{\ell} \mapsto_{\mathrm{I}} \mathsf{inj}_i\ x * P\}\ e_i\ \{\mathsf{ret}.\ Q\}}{\{\underline{\ell} \mapsto_{\mathrm{I}} \mathsf{inj}_i\ x * \triangleright P\}\ \mathsf{case}(\underline{\ell}, \mathsf{inj}_1\ x \Rightarrow e_1, \mathsf{inj}_2\ x \Rightarrow e_2)\ \{\mathsf{ret}.\ Q\}}$$

Figure 5.11: Concurrent Hoare logic

- Shared resources, on the other hand, are characterized indirectly through island assertions, *i.e.,* controlled by protocols. In reasoning using the Shared rule, we employ our rely and guarantee preorders to reason about concurrent interference and the permissible state changes according to a protocol. The rule focuses on a particular island with ID $\underline{i}$ described by ι_0. Of course, the real state of the island is *some* $\iota \overset{\text{rely}}{\sqsupseteq} \iota_0$. For *each* such possible state, the operation a we are performing must move the protocol to a guarantee-future state $\iota' \overset{\text{guar}}{\sqsupseteq} \iota$, at the same time establishing some postcondition Q.[27] In checking the operation a against the protocol, we use an atomic Hoare triple as applied to the *interpretations* of the protocol states—exposing the concrete shared resources of the island. Although each possible protocol state ι might lead to a different concrete new state ι' and postcondition Q, all such final states must be summarized by a single, abstract postcondition R. The use of the $\triangleright$ modality around the island's interpretation in the postcondition is justified by the fact that island interpretations are delayed one step; see the definition of world satisfaction (§5.3.5).

[27] This postcondition could contain, for example, resources whose ownership just passed from the island to the local thread.

Of course, shared resources are initially created as private ones and later exposed to other threads. The rule NewIsland allows a private resource to be "downgraded" into a shared one.

Finally, the rule PrivateSumElim accounts for the **case** construct on sum types, which interacts with the heap but is not an atomic expression. The rule eliminates a sum whose value is known and whose heap representation is privately-owned. By applying the rule of Disjunction, it can be made to operate on a sum whose value is not known in advance. We omit the similar rule that works on sums whose heap representation is managed as part of a shared island.

5.5.5 *Atomic Hoare logic*

Finally, we have atomic Hoare triples, whose definition is loosely a one-step, single-thread version of threadpool simulation:

$$\begin{aligned}&\vDash (\!|P|\!)\ a\ (\!|x.\ Q|\!) \triangleq \forall W, \eta, \eta_F \# \eta, \rho.\\ &\quad \text{if } W.k > 0 \text{ and } \triangleright W, \eta \vDash^\rho P \text{ and } (\eta \otimes \eta_F).h; a \hookrightarrow h'; v \text{ then } \exists \eta' \# \eta_F.\\ &\qquad h' = (\eta' \otimes \eta_F).h, \quad (\eta \otimes \eta_F).\Sigma \rightrightarrows (\eta' \otimes \eta_F).\Sigma, \quad \triangleright W, \eta' \vDash^\rho Q[v/x]\end{aligned}$$

where P and Q are assumed to be *token-pure*, *i.e.,* not to include any island assertions claiming ownership of tokens.[28] We also assume that P, Q and a have no free term variables, but in the inference rules below we implicitly universally-quantify any free variables (just as with hypothetical reasoning, §5.5.1). Finally, a subtle point regarding step-indexing: we interpret P as being *implicitly* wrapped in a $\triangleright$ modality, which allows us to avoid cluttering the rules of atomic Hoare logic with the modality—but note that, when we lift atomic triples in the Private and Shared rules (§5.5.4), the modality becomes explicit.

[28] Recall that this is the same restriction placed on the interpretations of islands.

The most important difference from threadpool simulation (and, thus, from concurrent triples) is that atomic triples *do not interpret the islands of* W. All of the resources they interact with must be concretely described by the precondition P. There is therefore no mention of rely or guarantee relations—protocols play no role here. We ultimately get away with this because atomic Hoare triples deal only with one-step executions, whose protocol conformance can be checked when they are lifted into concurrent Hoare triples (§5.5.4).

The rules in Figure 5.12 spell out the atomic Hoare logic. They are pleasantly straightforward—essentially the standard "small axioms" of sequential separation logic,[29] with the notable absence of a rule for sequencing. As with the PrivateSumElim rule, we axiomatize **cas** using a pair of rules for its two possible outcomes, but we can derive a rule that covers both by applying AConsequence and ADisjunction.

[29] Reynolds (2002), "Separation logic: a logic for shared mutable data structures"

▸ Reasoning about atomic expressions.

Inject
$(\!|\mathsf{emp}|\!)\ \mathbf{inj}_i\ v\ (\!|\mathsf{ret}.\ \mathsf{ret} \mapsto_{\mathrm{I}} \mathbf{inj}_i\ v|\!)$

Alloc
$(\!|\mathsf{emp}|\!)\ \mathbf{new}\ \overline{v}\ (\!|\mathsf{ret}.\ \mathsf{ret} \mapsto_{\mathrm{I}} (\overline{v})|\!)$

Deref
$(\!|v \mapsto_{\mathrm{I}} (\overline{v})|\!)\ \mathbf{get}(v[i])\ (\!|\mathsf{ret}.\ \mathsf{ret} = v_i \wedge v \mapsto_{\mathrm{I}} (\overline{v})|\!)$

Assign
$(\!|v \mapsto_{\mathrm{I}} (v_1, \ldots, v_n)|\!)\ v[i] \ :=\ v_i'\ (\!|\mathsf{ret}.\ \mathsf{ret} = () \wedge v \mapsto_{\mathrm{I}} (v_1, \ldots, v_{i-1}, v_i', v_{i+1}, \ldots, v_n)|\!)$

CASTrue
$(\!|v \mapsto_{\mathrm{I}} (v_1, \ldots, v_n)|\!)\ \mathbf{cas}(v[i], v_i, v_i')\ (\!|\mathsf{ret}.\ \mathsf{ret} = \mathbf{true} \wedge v \mapsto_{\mathrm{I}} (v_1, \ldots, v_{i-1}, v_i', v_{i+1}, \ldots, v_n)|\!)$

CASFalse
$(\!|v \mapsto_{\mathrm{I}} (\overline{v}) \wedge v_o \neq v_i|\!)\ \mathbf{cas}(v[i], v_o, v_i')\ (\!|\mathsf{ret}.\ \mathsf{ret} = \mathbf{false} \wedge v \mapsto_{\mathrm{I}} (\overline{v})|\!)$

▸ Logical and structural rules.

AConsequence
$$\frac{P \vdash P' \qquad (\!|P'|\!)\ a\ (\!|x.\ Q'|\!) \qquad Q' \vdash Q}{(\!|P|\!)\ a\ (\!|x.\ Q|\!)}$$

AFrame
$$\frac{(\!|P|\!)\ a\ (\!|x.\ Q|\!)}{(\!|P * R|\!)\ a\ (\!|x.\ Q * R|\!)}$$

ADisjunction
$$\frac{(\!|P_1|\!)\ a\ (\!|x.\ Q|\!) \qquad (\!|P_2|\!)\ a\ (\!|x.\ Q|\!)}{(\!|P_1 \vee P_2|\!)\ a\ (\!|x.\ Q|\!)}$$

Figure 5.12: Atomic Hoare logic

5.5.6 *Reasoning about specification code*

Our treatment of program reasoning thusfar has focused on implementation code, but specifications are programs, too.

Execution of specification code is always tied, simulation-style, to the execution of implementation code, as can most vividly be seen in the definition of threadpool simulation (§5.4.3). That definition presents two opportunities for executing spec code: either in response to atomic steps of implementation code, or when the (main thread of the) implementation produces a value. We likewise have two proof rules for spec execution, one for atomic triples and one for concurrent triples. The rules allow the spec to take steps "in the postcondition" of a triple:

AExecSpec
$$\frac{(\!|P|\!)\ a\ (\!|x.\ Q|\!) \qquad Q \Rrightarrow R}{(\!|P|\!)\ a\ (\!|x.\ R|\!)}$$

ExecSpec
$$\frac{\{P\}\ e\ \{x.\ Q\} \qquad Q \Rrightarrow R}{\{P\}\ e\ \{x.\ R\}}$$

The ExecSpec and AExecSpec rules have something of the flavor of the rule of consequence, but do not be deceived: $P \Rightarrow Q$ and $P \Rrightarrow Q$ are quite different claims. The latter is defined as follows:

$$\vDash P \Rrightarrow Q \ \triangleq\ \forall W, h, \Sigma, \Sigma_F \# \Sigma, \rho.$$
$$\text{if } W, (h, \Sigma) \vDash^{\rho} P \text{ then } \exists \Sigma'.\ (\Sigma \otimes \Sigma_F) \Rrightarrow (\Sigma' \otimes \Sigma_F) \text{ and } W, (h, \Sigma') \vDash^{\rho} Q$$

So $P \Rrightarrow Q$ means that the (set of) spec resources in P be can be (speculatively) executed to produce new spec resources satisfying Q. Its definition is essentially an excerpt from the definition of atomic triples.

We will not give a detailed set of proof rules for spec execution $\Rrightarrow$, instead leaving such claims as proof obligations to be carried out "in the model."

5.5.7 *Reasoning about recursion*

The Löb rule provides the essential ingredient for reasoning about recursion, but it may not be obvious how to usefully apply it. While there are many useful recursion schemes derivable from Löb, the one we will use applies to Hoare triples on the bodies of recursive functions:

$$\textsc{UnfoldRec}\quad \frac{\Phi \vdash \forall f, x.\ \{P \land \forall x.\ \{P\}\ f\ x\ \{\mathsf{ret}.\ Q\}\}\ e\ \{\mathsf{ret}.\ Q\}}{\Phi \vdash \forall x.\ \{P\}\ e[\mathbf{rec}\ f(x).e/f]\ \{\mathsf{ret}.\ Q\}}$$

The UnfoldRec rule provides a coinductive reasoning principle for recursive functions, allowing a triple to be proved under the assumption that it holds whenever the function is applied. It is derivable using Löb, Pure and Hypo (together with several of the standard logical rules).

We omit the derivation, which is somewhat tedious, but the intuition is simple. If we instantiate the Löb rule to the conclusion of UnfoldRec, we have the derived rule

$$\frac{\Phi, \triangleright \forall x.\ \{P\}\ e[\mathbf{rec}\ f(x).e/f]\ \{\mathsf{ret}.\ Q\} \vdash \forall x.\ \{P\}\ e[\mathbf{rec}\ f(x).e/f]\ \{\mathsf{ret}.\ Q\}}{\Phi \vdash \forall x.\ \{P\}\ e[\mathbf{rec}\ f(x).e/f]\ \{\mathsf{ret}.\ Q\}}$$

But a function application takes a step to actually perform! The assumption we gain from the Löb rule characterizes a function application just after β-reduction, and is wrapped with a use of "later." By using the Pure rule (§5.5.4), we can replace it with an assumption characterizing a function application just *before* β-reduction—and thereby "eat up" the extra $\triangleright$ modality at just the right moment.

5.5.8 *Derived rules for pure expressions*

We close the discussion of proof theory with a few derived rules dealing with common pure expression forms:

$$\frac{\Phi \vdash \{P\}\ e\ \{x.\ \triangleright Q\} \qquad \Phi \vdash \forall x.\ \{Q\}\ e'\ \{y.\ R\}}{\Phi \vdash \{P\}\ \mathbf{let}\ x = e\ \mathbf{in}\ e'\ \{y.\ R\}}$$

$$\frac{\Phi \vdash \{P\}\ e\ \{x.\ (x = \mathbf{true} \land \triangleright Q_1) \lor (x = \mathbf{false} \land \triangleright Q_2)\} \quad \Phi \vdash \{Q_1\}\ e_1\ \{\mathsf{ret}.\ R\} \qquad \Phi \vdash \{Q_2\}\ e_2\ \{\mathsf{ret}.\ R\}}{\Phi \vdash \{P\}\ \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\ \{\mathsf{ret}.\ R\}}$$

$$\frac{\Phi \vdash \{P\}\ e\ \{x.\ (x = \mathbf{null} \land \triangleright Q_1) \lor (\exists \ell.\ x = \ell \land \triangleright Q_2)\} \quad \Phi \vdash \{Q_1\}\ e_1\ \{\mathsf{ret}.\ R\} \qquad \Phi \vdash \forall \ell.\ \{Q_2[\ell/x]\}\ e_2\ \{\mathsf{ret}.\ R\}}{\Phi \vdash \{P\}\ \mathbf{case}(e, \mathbf{null} \Rightarrow e_1, x \Rightarrow e_2)\ \{\mathsf{ret}.\ R\}}$$

These rules all follow through use of Pure, Consequence and Disjunction.

5.6 METATHEORY

"Anything that thinks logically can be fooled by something else that thinks at least as logically as it does."

—Douglas Adams, The Hitchhiker's Guide

Having seen all the layers of our logic, there are now two interesting metatheoretic questions to ask about its soundness:

1. Are the proof rules sound for the semantics of assertions?
2. Do refinement assertions actually imply the corresponding contextual refinements?

Most of our proof-theoretic rules follow quite simply from assertion semantics. The interesting ones—primarily, the "glue" rules for concurrent Hoare logic (§5.5.4)—are built on precisely the same lemmas we use in showing soundness for contextual refinement. So we focus on Question 2.

5.6.1 *Soundness for refinement*

The key theorem is the following:[30]

[30] We abbreviate $\forall U.\ U \vDash^{\varnothing} \varphi$ as $\vDash \varphi$.

Theorem 1 (Soundness). If $\vDash \Omega \vdash e_1 \preceq^{\mathcal{E}} e_2 : \tau$ then $\Omega \vDash e_1 \preceq e_2 : \tau$.

In proving a contextual refinement from a refinement assertion, we cannot assume anything about the world U in which we work. After all, the world is a representation of the context's behavior, about which we must assume nothing.

Soundness relies on the usual decomposition of contextual refinement into two properties: adequacy and compatibility.[31]

[31] Any relation that is adequate and compatible is contained in contextual refinement, which is the largest such relation.

▸ ADEQUACY FOR REFINEMENT ASSERTIONS IS EASY TO SHOW:

Theorem 2 (Adequacy). Suppose $\vDash \cdot \vdash e_1 \preceq^{\mathcal{E}} e_2 : \mathbf{nat}$, and let i, j and n be arbitrary. If we have

$$\exists h_1, T_1.\ \varnothing; [i \mapsto e_1] \to^* h_1; [i \mapsto n] \uplus T_1$$

then we also have

$$\exists h_2, T_2.\ \varnothing; [j \mapsto e_2] \to^* h_2; [j \mapsto n] \uplus T_2.$$

Adequacy just says that when we assert refinement between closed expressions of **nat** type, the directly-observable behavior of the implementation e_1 (as a program) is reproducible by the specification e_2.

▸ COMPATIBILITY IS MUCH MORE DIFFICULT TO SHOW:

Theorem 3 (Compatibility). If $\vDash \Omega \vdash e_1 \preceq^{\mathcal{E}} e_2 : \tau$ and $C : (\Omega, \tau) \rightsquigarrow (\Omega', \tau')$ then $\vDash \Omega' \vdash C[e_1] \preceq^{\mathcal{E}} C[e_2] : \tau'$.

Compatibility captures the idea that, if we claim an implementation refines its spec, layering on additional client-side observations should never enable us to discover an implementation behavior not reproducible by the spec. We prove it by induction over the derivation of $C : (\Omega, \tau) \rightsquigarrow (\Omega', \tau')$, treating each case as a separate "compatibility lemma."

LemFrame

$$\frac{W, \eta \vDash^{\rho} T@m \left\{x.\ Q\right\} \qquad W_f, \eta_f \vDash^{\rho} R \qquad W \# W_f \qquad \eta \# \eta_f}{W \otimes W_f, \eta \otimes \eta_f \vDash^{\rho} T@m \left\{x.\ Q * R\right\}}$$

LemPar

$$\frac{\begin{array}{c} W_1 \# W_2 \qquad \eta_1 \# \eta_2 \qquad T_1 \# T_2 \qquad m_1 \neq \mathsf{none} \Rightarrow m_1 \in \mathrm{dom}(T_1) \\ W_1, \eta_1 \vDash^{\rho} T_1@m_1 \left\{x.\ Q_1\right\} \qquad W_2, \eta_2 \vDash^{\rho} T_2@m_2 \left\{x.\ Q_2\right\} \end{array}}{W_1 \otimes W_2, \eta_1 \otimes \eta_2 \vDash^{\rho} T_1 \uplus T_2@m_1 \left\{x.\ Q_1\right\}}$$

LemSeq

$$\frac{\begin{array}{c} W, \eta \vDash^{\rho} [i \mapsto e] \uplus T@i \left\{x.\ Q\right\} \\ \forall v, W', \eta'.\ W', \eta' \vDash Q[v/x] \implies W', \eta' \vDash^{\rho} [i \mapsto K[v]]@i \left\{x.\ R\right\} \end{array}}{W, \eta \vDash^{\rho} [i \mapsto K[e]] \uplus T@i \left\{x.\ R\right\}}$$

Figure 5.13: Key, low-level lemmas for soundness

5.6.2 *Lemmas for threadpool simulation*

The key to tractability for the compatibility lemmas is isolating a yet-lower-level set of lemmas, working at the level of threadpool simulation, that allow us to glue together computations in various ways. These are the same lemmas we mentioned above that likewise support the glue rules of concurrent Hoare logic. They are shown as inference rules in Figure 5.13.[32]

To prove the lemmas in Figure 5.13, we need another set of lemmas giving some fundamental properties of the rely and guarantee preorders:

Lemma 1 (Rely-closure of Assertions)**.** $W, \eta \vDash^{\rho} P$ and $W \overset{\text{rely}}{\sqsubseteq} W'$ implies $W', \eta \vDash^{\rho} P$.

Lemma 2 (Rely Decomposition)**.** If $W_1 \otimes W_2 \overset{\text{rely}}{\sqsubseteq} W'$ then there are W_1' and W_2' with $W' = W_1' \otimes W_2'$, $W_1 \overset{\text{rely}}{\sqsubseteq} W_1'$ and $W_2 \overset{\text{rely}}{\sqsubseteq} W_2'$.

Lemma 3 (Token Framing)**.** If $W \overset{\text{guar}}{\sqsubseteq} W'$ and $W \# W_f$ then there exists some $W_f' \# W'$ such that $W_f \overset{\text{rely}}{\sqsubseteq} W_f'$ and $W \otimes W_f \overset{\text{guar}}{\sqsubseteq} W' \otimes W_f'$.

The LemPar lemma provides the basis for compatibility of **fork**, while LemSeq is used not only to prove the Bind rule, but also in nearly *every* compatibility lemma that involves subexpressions. For example, consider compatibility of function application:

$$\frac{\vDash \Omega \vdash e_1 \preceq^{\mathcal{E}} e_2 : \tau' \to \tau \qquad \vDash \Omega \vdash e_1' \preceq^{\mathcal{E}} e_2' : \tau'}{\vDash \Omega \vdash e_1\ e_1' \preceq^{\mathcal{E}} e_2\ e_2' : \tau}$$

The expression $e_1\ \ e_1'$ is evaluated by first evaluating e_1 and e_1', but these subexpression evaluations are completely uninteresting: in the proof, we want to simply "drop in" the assumptions about e_1 and e_1' and jump directly to the

[32] The notation # is used in the standard way to claim that the relevant composition is defined. For example, $W \# W'$ means that $W \otimes W'$ is defined.

interesting part, namely β-reduction. The LemSeq lemma lets us do exactly that.[33]

The proof of LemSeq is what motivates the inclusion of an unknown evaluation context K in the definition of expression refinement (§5.4.2): when reasoning about the evaluation of a subexpression e in some $K[e]$, we treat K as part of the unknown context. But when reasoning about the subsequent execution of the continuation $K[v]$, we think of K as part of the expression.

Detailed proofs of the lemmas in Figure 5.13, the compatibility lemmas, and other key glue rules of concurrent Hoare logic are given in Appendix C.

[33] This is a bit like "locally" $\top\top$-closing (Pitts and Stark 1998) the logical relation.

6

Example proofs

▸ Synopsis This chapter exercises the logic of local protocols on a series of realistic examples employing several sophisticated techniques for scalability: elimination backoff (§6.4), lock-free traversal (§6.5), and helping (§6.6).

"None of the programs in this monograph, needless to say, has been tested on a machine."

—Edsger W. Dijkstra

"Beware of bugs in the above code; I have only proved it correct, not tried it."

—Donald E. Knuth

6.1 Proof outlines

Before examining examples, we need to set some ground rules for tractable notation.

We show example proofs as "Hoare proof outlines", which interleave code with (**colored**) annotations making assertions. In reading such a proof outline, one should imagine each bit of code, along with the assertions immediately before and after it, as comprising a concurrent Hoare triple:

- A sequence like
$$\{P\}\ e;\ \{x.\ Q\}\ e'\ \{y.\ R\}$$
then comprises two triples $\{P\}\ \ e\ \ \{x.\ Q\}$ and $\forall x.\ \ \{Q\}\ \ e'\ \ \{y.\ R\}$ together with a use of the derived rule for sequencing the two expressions. It is therefore a proof of $\{P\}\ \ e; e'\ \ \{y.\ R\}$. Since semicolon-sequenced code usually just returns unit values in between, we often omit the binder for x.

- Similarly, a sequence like
$$\{P\}\ \mathbf{let}\ x = e\ \mathbf{in}\ \{Q\}\ e'\ \{y.\ R\}$$
comprises two triples $\{P\}\ \ e\ \ \{x.\ Q\}$ and $\forall x.\ \ \{Q\}\ \ e'\ \ \{y.\ R\}$ together with a use of the derived rule for sequencing the two expressions. It is therefore a proof of $\{P\}\ \ \mathbf{let}\ x = e\ \mathbf{in}\ e'\ \ \{y.\ R\}$. Note that we implicitly use the **let**-binder x in the postcondition Q.

- Uses of the rule of Consequence are given as back-to-back annotations: $\{P\}\{Q\}$. Similarly, uses of ExecSpec are given as back-to-back annotations with $\rightrightarrows$ written in between: $\{P\} \rightrightarrows \{Q\}$.

- Conditionals are handled using a joint precondition and annotating each branch with its own assertion. For example, the outline
$$\{P\}\ \mathbf{if}\ e\ \mathbf{then}\ \{Q_1\}\ e_1\ \{\mathsf{ret}.\ R\}\ \mathbf{else}\ \{Q_2\}\ e_2\ \{\mathsf{ret}.\ R\}$$

comprises the triples

$$\{P\}\ e\ \{x.\ (x = \textsf{true} \land \triangleright Q_1) \lor (x = \textsf{false} \land \triangleright Q_2)\}$$
$$\{Q_1\}\ e_1\ \{\text{ret. } R\} \qquad \{Q_2\}\ e_2\ \{\text{ret. } R\}$$

and constitutes a proof of

$$\{P\}\ \textsf{if}\ e\ \textsf{then}\ e_1\ \textsf{else}\ e_2\ \{\text{ret. } R\}$$

- We consistently use the UnfoldRec rule (§5.5.7) to deal with recursion. Rather than explicitly writing the unfolded recursive assumption nested within an assertion, we leave it implicit. For example, when reasoning about **rec** $f(x).e$, we will use a proof outline like

  $$\{P\}\ e\ \{\text{ret. } Q\}$$

  with f and x free, in which we implicitly assume that

  $$\forall x.\ \{P\}\ f\ x\ \{\text{ret. } Q\}$$

- We implicitly use the Mono rule to strengthen preconditions or weaken postconditions as necessary, and implicitly apply Weaken to throw away unneeded assertions in the postcondition.

We use proof outlines to give the high-level structure of a proof in terms of its step-by-step gluing of concurrent Hoare triples—but of course, the correctness of the *individual* Hoare triples will not always be apparent. Thus we often supplement proof outlines with detailed explanations of how they use the Private and Shared rules (§5.5.4) to lift atomic triples, and in some cases also give details for the atomic triples themselves.

6.2 WARMUP: CONCURRENT COUNTERS

For our first example refinement proof, we return again[1] to concurrent counters—a simple enough example to let us "practice" the basic proof mechanics without getting bogged down in algorithmic details.

[1] And not for the last time! See §10.2.1.

We prove in particular that the following optimistic, **cas**-based counter,

$$\begin{array}{lll} \textsf{casCnt} \triangleq \textsf{let} & r & = \textsf{new}\ 0 \\ & \textsf{inc} & = \lambda().\ \textsf{let}\ n = \textsf{get}\ r \\ & & \qquad \textsf{in if}\ \textsf{cas}(r, n, n+1)\ \textsf{then}\ ()\ \textsf{else}\ \textsf{inc}() \\ & \textsf{read} & = \lambda().\ \textsf{get}\ r \\ \textsf{in}\ (\textsf{inc}, \textsf{read}) & & \end{array}$$

refines the following canonical atomic spec,

$$\textsf{atomCnt} \triangleq \textsf{let}\ r = \textsf{new}\ 0\ \textsf{in}\ \textsf{mkAtomic}(\lambda().\ r := \textsf{get}\ r + 1,\ \lambda().\ \textsf{get}\ r)$$

In other words, we will show

$$\cdot \vdash \textsf{casCnt} \preceq^{\mathcal{E}} \textsf{atomCnt} : (\textsf{unit} \to \textsf{unit}) \times (\textsf{unit} \to \textsf{nat})$$

6.2.1 *The protocol*

Before delving into the details of the proof, we need to determine the protocol governing casCnt's hidden state: the reference r. Fortunately, the protocol is extremely simple: the entire internal state of the data structure is represented by a single value at base type. Consequently, executing a **cas** on r is so informative that the inc operation can withstand essentially arbitrary interference, so long as the spec's state remains linked. In other words, we can use the following invariant (*i.e.,* single-state) protocol:[2]

$$\mathrm{inv}(\exists n.\ r_{\mathrm{I}} \mapsto_{\mathrm{I}} n * r_{\mathrm{S}} \mapsto_{\mathrm{S}} n * \mathsf{lock}_{\mathrm{S}} \mapsto_{\mathrm{S}} \mathbf{false})$$

[2] Invariant protocols $\mathrm{inv}(P)$ were defined in §5.4.2.

As usual, in reasoning about the code we rename variables, adding I and S subscripts to clearly distinguish between implementation- and specification-side names. Here the invariant ensures that the implementation and spec counter values change together, and that every execution of spec code goes through an entire critical section (leaving the lock from mkAtomic free, *i.e.,* **false**).

6.2.2 *The proof*

The proof of refinement here, as with most we will consider, works in essentially two stages:

Construction. First we account for the code making up the "constructor" of an object. For counters, construction includes includes the allocation of r_{I} on the implementation side and r_{S} and $\mathsf{lock}_{\mathrm{S}}$ on the spec side. Once the objects have been constructed, we introduce a new island to link implementation and specification together.

Method refinement. We then prove refinement of the method tuples returned by the spec by those of the implementation. These refinement proofs assume the existence of the island introduced in the first stage.

And so we begin.

▸ CONSTRUCTION. To prove

$$\mathsf{casCnt} \preceq^{\mathcal{E}} \mathsf{atomCnt} : \tau \quad \text{where} \quad \tau = (\mathbf{unit} \to \mathbf{unit}) \times (\mathbf{unit} \to \mathbf{nat})$$

it suffices to show (by SPECINTRO, §5.5.3), for all j and K, that

$$\{j \rightarrowtail_{\mathrm{S}} K[\mathsf{atomCnt}]\}\ \mathsf{casCnt}\ \{x.\ \exists y.\ x \preceq^{\mathcal{V}} y : \tau\ \wedge\ j \rightarrowtail_{\mathrm{S}} K[y]\}$$

Using the derived rule for **let** expressions (§5.5.8), we can first reason about the allocation of r_{I}:

$$(\!|\mathsf{emp}|\!)\ \mathbf{new}\ 0\ (\!|r_{\mathrm{I}}.\ r_{\mathrm{I}} \mapsto_{\mathrm{I}} 0|\!)$$

which, by AFRAME (§5.5.5), gives us

$$(\!|j \rightarrowtail_{\mathrm{S}} K[\mathsf{atomCnt}]|\!)\ \mathbf{new}\ 0\ (\!|r_{\mathrm{I}}.\ r_{\mathrm{I}} \mapsto_{\mathrm{I}} 0 * j \rightarrowtail_{\mathrm{S}} K[\mathsf{atomCnt}]|\!)$$

At this point, the implementation's representation has been constructed, but the spec's has not. Recalling the definition of mkAtomic (§3.4), it is easy to see that

$$j \rightarrowtail_{\mathrm{s}} K[\mathsf{atomCnt}] \quad \rightrightarrows \quad \exists r_{\mathrm{s}}.\ j \rightarrowtail_{\mathrm{s}} K[(\mathsf{inc}_{\mathrm{s}}, \mathsf{read}_{\mathrm{s}})] * \mathsf{lock}_{\mathrm{s}} \mapsto_{\mathrm{s}} \textbf{false} * r_{\mathrm{s}} \mapsto_{\mathrm{s}} 0$$

where

$$\begin{aligned} \mathsf{inc}_{\mathrm{s}} &\triangleq \mathsf{withLock}(\mathsf{lock}_{\mathrm{s}},\ \lambda().\ r_{\mathrm{s}} := \textbf{get}\ r_{\mathrm{s}} + 1) \\ \mathsf{read}_{\mathrm{s}} &\triangleq \mathsf{withLock}(\mathsf{lock}_{\mathrm{s}},\ \lambda().\ \textbf{get}\ r_{\mathrm{s}}) \end{aligned}$$

Thus, by ASpecExec (§5.5.6), we have

$$(\!|j \rightarrowtail_{\mathrm{s}} K[\mathsf{atomCnt}]|\!)\ \textbf{new}\ 0\ (\!|r_{\mathrm{I}}.\ r_{\mathrm{I}} \mapsto_{\mathrm{I}} 0 * \exists r_{\mathrm{s}}.\ j \rightarrowtail_{\mathrm{s}} K[(\mathsf{inc}_{\mathrm{s}}, \mathsf{read}_{\mathrm{s}})] * \mathsf{lock}_{\mathrm{s}} \mapsto_{\mathrm{s}} \textbf{false} * r_{\mathrm{s}} \mapsto_{\mathrm{s}} 0|\!)$$

At this point in the proof, the representations have been constructed as privately-owned resources. We therefore lift our atomic triple to a concurrent one using Private (§5.5.4):

$$\{\triangleright j \rightarrowtail_{\mathrm{s}} K[\mathsf{atomCnt}]\}\ \textbf{new}\ 0\ \{r_{\mathrm{I}}.\ r_{\mathrm{I}} \mapsto_{\mathrm{I}} 0 * \exists r_{\mathrm{s}}.\ j \rightarrowtail_{\mathrm{s}} K[(\mathsf{inc}_{\mathrm{s}}, \mathsf{read}_{\mathrm{s}})] * \mathsf{lock}_{\mathrm{s}} \mapsto_{\mathrm{s}} \textbf{false} * r_{\mathrm{s}} \mapsto_{\mathrm{s}} 0\}$$

Although the representations are not exported directly, the exported closures (*e.g.,* $\mathsf{inc}_{\mathrm{s}}$) have shared access to them—so to finish the construction phase, we need to move them into an island. We do so using NewIsland (and an application of Consequence, which we elide):[3]

$$\begin{aligned} &\{\triangleright j \rightarrowtail_{\mathrm{s}} K[\mathsf{atomCnt}]\}\ \textbf{new}\ 0\ \{r_{\mathrm{I}}.\ \exists r_{\mathrm{s}}.\ j \rightarrowtail_{\mathrm{s}} K[(\mathsf{inc}_{\mathrm{s}}, \mathsf{read}_{\mathrm{s}})] * \iota\} \\ &\quad \text{where } \iota \triangleq \mathsf{inv}(\exists n.\ r_{\mathrm{I}} \mapsto_{\mathrm{I}} n * r_{\mathrm{s}} \mapsto_{\mathrm{s}} n * \mathsf{lock}_{\mathrm{s}} \mapsto_{\mathrm{s}} \textbf{false}) \end{aligned}$$

[3] Recall that we use ι as a shorthand for the assertion $\exists x.\ x \mapsto \iota$.

▸ METHOD REFINEMENT. To show refinement for the pair of methods, we must show refinement for each independently:[4]

$$\iota \vdash \mathsf{inc}_{\mathrm{I}} \preceq^{\mathcal{V}} \mathsf{inc}_{\mathrm{s}} : \textbf{unit} \to \textbf{unit} \quad \text{and} \quad \iota \vdash \mathsf{read}_{\mathrm{I}} \preceq^{\mathcal{V}} \mathsf{read}_{\mathrm{s}} : \textbf{unit} \to \textbf{nat}$$

where $\mathsf{inc}_{\mathrm{I}}$ is the actual (recursive) function definition for inc's implementation, and likewise for $\mathsf{read}_{\mathrm{I}}$.[5] We will examine the proof for inc, from which the proof for read can be easily extrapolated.

Showing refinement amounts to proving, for all j and K, that

$$\forall x_{\mathrm{I}} \preceq^{\mathcal{V}} x_{\mathrm{s}} : \textbf{unit}.\ \{j \rightarrowtail_{\mathrm{s}} K[\mathsf{incBody}_{\mathrm{s}}[\mathsf{inc}_{\mathrm{s}}/\mathsf{inc}]]\}\ \mathsf{incBody}_{\mathrm{I}}[\mathsf{inc}_{\mathrm{I}}/\mathsf{inc}]\ \{\mathsf{ret}_{\mathrm{I}}.\ \exists \mathsf{ret}_{\mathrm{s}}.\ \mathsf{ret}_{\mathrm{I}} \preceq^{\mathcal{V}} \mathsf{ret}_{\mathrm{s}} : \textbf{unit} \wedge j \rightarrowtail_{\mathrm{s}} K[\mathsf{ret}_{\mathrm{s}}]\}$$

where $\mathsf{incBody}_{\mathrm{I}}$ is the body of $\mathsf{inc}_{\mathrm{I}}$ (and likewise for the spec versions). Since $\mathsf{inc}_{\mathrm{I}}$ is a recursive function, we will use UnfoldRec to carry out the proof. As discussed in §6.1, we give the proof for its body, treating inc as a free variable which we implicitly assume to behave according to the triple we prove.

The first, somewhat startling observation about the proof outline (Figure 6.1) is that we do not bother to record any new facts when executing $\textbf{get}\ r_{\mathrm{I}}$. But the reason is simple: there are no *stable* facts to be gained! The moment after reading r_{I}, all we know is that its value was once n, and our one-state protocol (recorded in ι) is too simple to record even this fact.

[4] Here we are implicitly using the rule for introducing value refinement at pair type (§5.5.3), as well as the HypoOut rule to record the assumption ι.

[5] In the rest of the examples for this section, we will continue to follow this convention, using subscripts I and S on method names to stand for the corresponding anonymous, recursive function definitions.

Let $e_s \triangleq K[\mathsf{incBody}_s[\mathsf{inc}_s/\mathsf{inc}]]$:

$\{j \rightarrowtail_s e_s * \iota\}$
  let n = **get** r_I **in**
$\{j \rightarrowtail_s e_s * \iota\}$
  if **cas**$(r_I, n, n+1)$
    then $\{j \rightarrowtail_s K[()] * \iota\}$ $()$ $\{\mathsf{ret}.\ \mathsf{ret} = () \wedge j \rightarrowtail_s K[()]\}$
    else $\{j \rightarrowtail_s e_s * \iota\}$ $\mathsf{inc}()$ $\{\mathsf{ret}.\ \mathsf{ret} = () \wedge j \rightarrowtail_s K[()]\}$

Figure 6.1: A proof outline for $\mathsf{incBody}_I$

Not to worry—it is the **cas** that will tell use the value of r when we really need to know it.

Despite gaining nothing semantically[6] from taking the snapshot of r, it is still worthwhile seeing how the triple

$$\{j \rightarrowtail_s e_s * \iota\}\ \textbf{get}\ r_I\ \{n.\ j \rightarrowtail_s e_s * \iota\}$$

is actually proved, namely, by lifting an atomic triple via the Shared rule. The rule requires that we consider every rely-future state of ι, moving from each to some guarantee-future state. We illustrate applications of Shared by giving tables like the following:

$\{j \rightarrowtail_s e_s * \iota\}\ \textbf{get}\ r_I\ \{n.\ j \rightarrowtail_s e_s * \iota\}$	
$j \rightarrowtail_s e_s * \iota.I(\mathsf{dummy})$	$n.\ j \rightarrowtail_s e_s * \iota.I(\mathsf{dummy})$

At the top of the table we give the concurrent triple we are trying to prove, which must in particular have a pre- and post-condition involving an island assertion. Underneath, we give a row for each possible rely-future state of the island in the precondition. Each row replaces the island assertion in its pre- and post-conditions with its interpretation at the asserted states; the state in the postcondition must be a guarantee move away from the one in the precondition (given the tokens owned in the original island assertion). Here there is only one state, "dummy", because we used inv to construct the island.

Every row represents an additional proof obligation: the corresponding atomic Hoare triple must be proved. So we must show

$$(\!|j \rightarrowtail_s e_s * \iota.I(\mathsf{dummy})|\!)\ \textbf{get}\ r_I\ (\!|n.\ j \rightarrowtail_s e_s * \iota.I(\mathsf{dummy})|\!)$$

where, recalling the definition of ι, we have

$$\iota.I(\mathsf{dummy}) \;=\; \exists n.\ r_I \mapsto_I n * r_s \mapsto_s n * \mathsf{lock}_s \mapsto_s \textbf{false}$$

This is easy to show: we use Deref (§5.5.5) for the read, deriving

$$(\!|r_I \mapsto_I n|\!)\ \textbf{get}\ r_I\ (\!|\mathsf{ret}.\ \mathsf{ret} = n \wedge r_I \mapsto_I n|\!)$$

and construct the rest of the atomic triple by using AFrame and AConsequence.

Next we have the **cas** in the guard of the **if** expression (Figure 6.1):

[6] We could just as well have **let** n = random(); the algorithm would still work from a safety standpoint. Of course, liveness and performance demand that we make a more educated guess as to r's value.

$$\{j \rightarrowtail_s e_s * \iota\}\ \mathbf{cas}(r_I, n, n+1)\ \left\{\mathsf{ret}.\ \begin{pmatrix}(\mathsf{ret} = \mathbf{false} \land j \rightarrowtail_s e_s)\\ \lor\ (\mathsf{ret} = \mathbf{true} \land j \rightarrowtail_s K[()])\end{pmatrix} * \iota\right\}$$

which we prove again using SHARED—but we omit the (again trivial) table, and instead move directly to the required atomic triple:

$$\langle\!| j \rightarrowtail_s e_s * \iota.J(\mathsf{dummy}) |\!\rangle\ \mathbf{cas}(r_I, n, n+1)\ \langle\!|\mathsf{ret}.\ \begin{pmatrix}(\mathsf{ret} = \mathbf{false} \land j \rightarrowtail_s e_s)\\ \lor\ (\mathsf{ret} = \mathbf{true} \land j \rightarrowtail_s K[()])\end{pmatrix} * \iota.J(\mathsf{dummy}) |\!\rangle$$

To prove this atomic triple, we begin by proving two simpler ones, which we join with ADISJUNCTION.

- FIRST, for the case that the **cas** succeeds (CASTRUE):

$$\langle\!| r_I \mapsto_I n |\!\rangle\ \mathbf{cas}(r_I, n, n+1)\ \langle\!|\mathsf{ret}.\ \mathsf{ret} = \mathbf{true} \land r_I \mapsto_I n+1 |\!\rangle$$

Using the AFRAME rule, we have

$$\begin{array}{l}\langle\!| r_I \mapsto_I n * r_s \mapsto_s n * \mathsf{lock}_s \mapsto_s \mathbf{false} * j \rightarrowtail_s e_s |\!\rangle\\ \quad \mathbf{cas}(r_I, n, n+1)\\ \langle\!|\mathsf{ret}.\ \mathsf{ret} = \mathbf{true} \land r_I \mapsto_I n+1 * r_s \mapsto_s n * \mathsf{lock}_s \mapsto_s \mathbf{false} * j \rightarrowtail_s e_s |\!\rangle\end{array}$$

leaving us with a postcondition in which the implementation and specification states differ. But:

$$r_s \mapsto_s n * \mathsf{lock}_s \mapsto_s \mathbf{false} * j \rightarrowtail_s e_s \quad \Rrightarrow \quad r_s \mapsto_s n+1 * \mathsf{lock}_s \mapsto_s \mathbf{false} * j \rightarrowtail_s K[()]$$

so, using ASPECEXEC, we have

$$\begin{array}{l}\langle\!| r_I \mapsto_I n * r_s \mapsto_s n * \mathsf{lock}_s \mapsto_s \mathbf{false} * j \rightarrowtail_s e_s |\!\rangle\\ \quad \mathbf{cas}(r_I, n, n+1)\\ \langle\!|\mathsf{ret}.\ \mathsf{ret} = \mathbf{true} \land r_I \mapsto_I n+1 * r_s \mapsto_s n+1 * \mathsf{lock}_s \mapsto_s \mathbf{false} * j \rightarrowtail_s K[()] |\!\rangle\end{array}$$

- SECOND, for the case that the **cas** fails (CASFALSE):

$$\langle\!| m \neq n \land r_I \mapsto_I m |\!\rangle\ \mathbf{cas}(r_I, n, n+1)\ \langle\!|\mathsf{ret}.\ \mathsf{ret} = \mathbf{false} \land r_I \mapsto_I m |\!\rangle$$

there is nothing to do but frame in the other resources:

$$\begin{array}{l}\langle\!| m \neq n \land r_I \mapsto_I m * r_s \mapsto_s m * \mathsf{lock}_s \mapsto_s \mathbf{false} * j \rightarrowtail_s e_s |\!\rangle\\ \quad \mathbf{cas}(r_I, n, n+1)\\ \langle\!|\mathsf{ret}.\ \mathsf{ret} = \mathbf{false} \land r_I \mapsto_I m * r_s \mapsto_s m * \mathsf{lock}_s \mapsto_s \mathbf{false} * j \rightarrowtail_s e_s |\!\rangle\end{array}$$

- WE FINISH THE PROOF of **cas** by joining the two atomic triples together, using ADISJUNCTION and ACONSEQUENCE. That just leaves the two arms of the **if** expression (Figure 6.1). The first arm is a trivial use of RETURN and FRAME (§5.5.4). The second arm follows from our (implicit!) assumption

about recursive invocations of the function, according to the convention laid out in §6.1.

And so concludes the proof of a simple concurrent counter. We have, of course, gone into considerable detail in order to fully illustrate the proof mechanics. The next two sections (on late/early choice and red/blue flags) will also go into a fair amount of detail, but subsequent examples (the Michael-Scott queue and conditional CAS) will work mostly at the level of proof outlines.

6.3 WARMUP: LATE VERSUS EARLY CHOICE

To get a bit more experience with the proof theory—and spec execution in particular—we give a more detailed proof that lateChoice refines earlyChoice at type $\mathbf{ref}(\mathbf{nat}) \to \mathbf{nat}$:

$$\begin{array}{rcl} \mathsf{rand} & \triangleq & \lambda().\ \mathbf{let}\ y = \mathbf{new}\ \mathbf{false}\ \mathbf{in}\ (\mathbf{fork}\ y := \mathbf{true}); \mathbf{get}(y[1]) \\ \mathsf{lateChoice} & \triangleq & \lambda x.\ x := 0; \mathsf{rand}() \\ \mathsf{earlyChoice} & \triangleq & \lambda x.\ \mathbf{let}\ r = \mathsf{rand}()\ \mathbf{in}\ x := 0; r \end{array}$$

As we noted in §4.5, these functions do not close over any hidden state and thus do not require a protocol (or even, in the terminology of this chapter, a construction phase for the proof).

So we proceed directly to the proof outline, shown in Figure 6.2. Because lateChoice takes a reference as an argument, we begin with an assumption $x_{\mathrm{I}} \preceq^{\mathcal{V}} x_{\mathrm{S}} : \mathbf{ref}(\mathbf{nat})$ which implies (§5.4.2, §5.5.3) the following (implicitly existential) island assertion:

$$\mathrm{inv}(\exists y_{\mathrm{I}}, y_{\mathrm{S}}.\ y_{\mathrm{I}} \preceq^{\mathcal{V}} y_{\mathrm{S}} : \mathbf{nat} \wedge x_{\mathrm{I}} \mapsto_{\mathrm{I}} y_{\mathrm{I}} * x_{\mathrm{S}} \mapsto_{\mathrm{S}} y_{\mathrm{S}})$$

Figure 6.2: Proof outline for refinement of earlyChoice by lateChoice

$\{x_{\mathrm{I}} \preceq^{\mathcal{V}} x_{\mathrm{S}} : \mathbf{ref}(\mathbf{nat}) \wedge j \rightarrowtail_{\mathrm{S}} K[\mathsf{earlyChoiceBody}]\}$
$\quad (\!|(\exists y_{\mathrm{I}}, y_{\mathrm{S}}.\ y_{\mathrm{I}} \preceq^{\mathcal{V}} y_{\mathrm{S}} : \mathbf{nat} \wedge (x_{\mathrm{I}} \mapsto_{\mathrm{I}} y_{\mathrm{I}} * x_{\mathrm{S}} \mapsto_{\mathrm{S}} y_{\mathrm{S}})) * j \rightarrowtail_{\mathrm{S}} K[\mathsf{earlyChoiceBody}]|\!)$
$\quad (\!|x_{\mathrm{I}} \mapsto_{\mathrm{I}} - * x_{\mathrm{S}} \mapsto_{\mathrm{S}} - * j \rightarrowtail_{\mathrm{S}} K[\mathsf{earlyChoiceBody}]|\!)$
$\quad\quad x_{\mathrm{I}} := 0$
$\quad (\!|x_{\mathrm{I}} \mapsto_{\mathrm{I}} 0 * x_{\mathrm{S}} \mapsto_{\mathrm{S}} - * j \rightarrowtail_{\mathrm{S}} K[\mathsf{earlyChoiceBody}]|\!) \rightrightarrows$
$\quad (\!|x_{\mathrm{I}} \mapsto_{\mathrm{I}} 0 * x_{\mathrm{S}} \mapsto_{\mathrm{S}} 0 * (j \rightarrowtail_{\mathrm{S}} K[\mathbf{true}] \oplus j \rightarrowtail_{\mathrm{S}} K[\mathbf{false}])|\!)$
$\{x_{\mathrm{I}} \preceq^{\mathcal{V}} x_{\mathrm{S}} : \mathbf{ref}(\mathbf{nat}) \wedge (j \rightarrowtail_{\mathrm{S}} K[\mathbf{true}] \oplus j \rightarrowtail_{\mathrm{S}} K[\mathbf{false}])\}$
$\{j \rightarrowtail_{\mathrm{S}} K[\mathbf{true}] \oplus j \rightarrowtail_{\mathrm{S}} K[\mathbf{false}]\}$
$\quad \mathsf{rand}()$
$\{\mathsf{ret}.\ (\mathsf{ret} = \mathbf{true} \vee \mathsf{ret} = \mathbf{false}) * (j \rightarrowtail_{\mathrm{S}} K[\mathbf{true}] \oplus j \rightarrowtail_{\mathrm{S}} K[\mathbf{false}])\}$
$\left\{\mathsf{ret}.\ \left(\mathsf{ret} = \mathbf{true} * (j \rightarrowtail_{\mathrm{S}} K[\mathbf{true}] \oplus j \rightarrowtail_{\mathrm{S}} K[\mathbf{false}])\right) \vee \left(\mathsf{ret} = \mathbf{false} * (j \rightarrowtail_{\mathrm{S}} K[\mathbf{true}] \oplus j \rightarrowtail_{\mathrm{S}} K[\mathbf{false}])\right)\right\} \rightrightarrows$
$\{\mathsf{ret}.\ (\mathsf{ret} = \mathbf{true} * j \rightarrowtail_{\mathrm{S}} K[\mathbf{true}]) \vee (\mathsf{ret} = \mathbf{false} * j \rightarrowtail_{\mathrm{S}} K[\mathbf{false}])\}$
$\{\mathsf{ret}.\ \mathsf{ret} \preceq^{\mathcal{V}} \mathsf{ret} : \mathbf{bool} \wedge j \rightarrowtail_{\mathrm{S}} K[\mathsf{ret}]\}$

In the triple for the assignment $x_s := 0$, we use the SHARED rule (§5.5.4) to unpack the concrete resources governed by this simple invariant protocol. Since the protocol has only the single dummy state, we dispense with the rely-guarantee table and instead show the derivation of the necessary atomic triple as a nested proof outline. The key point in the nested outline is that we use ASPECEXEC to speculatively execute *two* versions of the spec—both writing to x_s, but each tossing its coin in a different way.

That just leaves the implementation's use of rand. We assume

$$\{\mathsf{emp}\}\ \mathsf{rand}()\ \{\mathsf{ret}.\ \mathsf{ret} = \mathbf{true} \lor \mathsf{ret} = \mathbf{false}\},$$

a fact that can be readily proved by direct appeal to assertion semantics. After suitably framing this assumption with the assertion $j \rightarrowtail_s K[\mathbf{true}] \oplus j \rightarrowtail_s K[\mathbf{false}]$, all that is left to do is distribute the $*$ over the disjunction, and then use SPECEXEC—not to *execute* spec code, but rather to throw away those speculations that are no longer needed.

6.4 ELIMINATION: RED FLAGS VERSUS BLUE FLAGS

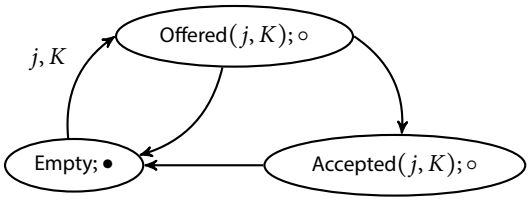
Now we are ready for examples involving sophisticated techniques from scalable concurrency (Chapter 2). We begin with elimination (§2.4.5).

▸ THE FLAG IMPLEMENTATIONS.

```
redFlag ≜
  let flag  =  new true,
      chan  =  new 0,
      flip  =  λ(). if cas(chan, 1, 2) then () else
                    if cas(flag, true, false) then () else
                    if cas(flag, false, true) then () else
                    if cas(chan, 0, 1) then
                      if cas(chan, 1, 0) then flip() else chan := 0
                    else flip(),
      read  =  λ(). get flag
  in (flip, read)
```

```
≤ blueFlag ≜
    let flag  =  new true,
        flip  =  λ(). flag := not (get flag),
        read  =  λ(). get flag
    in mkAtomic(flip, read)
```

▸ THE PROTOCOL.



▸ THE PROTOCOL STATE INTERPRETATION.

$$\begin{aligned} I(\mathsf{Empty}) &\triangleq Q * \mathsf{chan} \mapsto_I 0 \\ I(\mathsf{Offered}(j, K)) &\triangleq Q * \mathsf{chan} \mapsto_I 1 * j \rightarrowtail_s K[\mathsf{flip}_s()] \\ I(\mathsf{Accepted}(j, K)) &\triangleq Q * \mathsf{chan} \mapsto_I 2 * j \rightarrowtail_s K[()] \end{aligned}$$

$$Q \triangleq \exists x : \mathbf{bool}.\ \mathsf{flag}_I \mapsto_I x * \mathsf{flag}_s \mapsto_s x * \mathsf{lock} \mapsto_s \mathbf{false}$$

Figure 6.3: Red flags versus blue flags

For reference, both the code and the protocol for the red/blue flags example are shown in Figure 6.3.[7] One interesting aspect of the example will emerge as we go along: the order of *top-level* CASes in redFlag does not matter. In particular, any failing top-level **cas** leaves us with the same knowledge going out that we had coming in. We let θ be the transition system shown in Figure 6.3, with the single token "•" standing for ownership of the elimination channel.[8]

[7] We use the shorthand x : **bool** for x = **true** $\vee$ x = **false** in defining the protocol interpretation.

[8] We often write • in place of $\{\bullet\}$.

▸ CONSTRUCTION Fixing arbitrary j and K, we have:

$\{j \rightarrowtail_s K[\mathsf{blueFlagBody}]\}$
let flag_I = **new true in**
$\{j \rightarrowtail_s K[\mathsf{blueFlagBody}] * \mathsf{flag}_I \mapsto_I \mathbf{true}\}$
let chan = **new** 0 **in**
$\{j \rightarrowtail_s K[\mathsf{blueFlagBody}] * \mathsf{flag}_I \mapsto_I \mathbf{true} * \mathsf{chan} \mapsto_I 0\} \Rrightarrow$
$\{j \rightarrowtail_s K[(\mathsf{flip}_s, \mathsf{read}_s)] * \mathsf{flag}_I \mapsto_I \mathbf{true} * \mathsf{chan} \mapsto_I 0 * \mathsf{flag}_s \mapsto_s \mathbf{true} * \mathsf{lock} \mapsto_s \mathbf{false}\}$
$\{j \rightarrowtail_s K[(\mathsf{flip}_s, \mathsf{read}_s)] * I(\mathsf{Empty})\}$
$\{j \rightarrowtail_s K[(\mathsf{flip}_s, \mathsf{read}_s)] * (\theta, I, \mathsf{Empty}, \varnothing)\}$

This proof outline follows the same approach we have already seen for the construction phase (§6.2); the last step, in particular, uses the NEWISLAND rule (§5.5.4) to move privately-owned resources into a shared island that can be closed over by exported functions.

▸ METHOD REFINEMENT Now we must show that flip_I refines flip_s (and similarly for read, which is trivial) under the assumption $(\theta, I, \mathsf{Empty}, \varnothing)$. Since flip_I is a recursive function, we implicitly appeal to UNFOLDREC (§6.1). The high-level proof outline is given in Figure 6.4. It is, unfortunately, not terribly enlightening: for all but the last **cas**, success means that the operation is complete, while failure means that nothing—not even our asserted knowledge—has changed.

Let $P = (\theta, I, \mathsf{Empty}, \varnothing) * j \rightarrowtail_s K[\mathsf{flipBody}_s]$:

$\{P\}$ **if cas**(chan, 1, 2) **then** $\{j \rightarrowtail_s K[()]\}$ () $\{\mathsf{ret}.\ \mathsf{ret} = () \wedge j \rightarrowtail_s K[()]\}$
else $\{P\}$ **if cas**(flag, **true**, **false**) **then** $\{j \rightarrowtail_s K[()]\}$ () $\{\mathsf{ret}.\ \mathsf{ret} = () \wedge j \rightarrowtail_s K[()]\}$
else $\{P\}$ **if cas**(flag, **false**, **true**) **then** $\{j \rightarrowtail_s K[()]\}$ () $\{\mathsf{ret}.\ \mathsf{ret} = () \wedge j \rightarrowtail_s K[()]\}$
else $\{P\}$ **if cas**(chan, 0, 1) **then** $\{(\theta, I, \mathsf{Offered}(j, K), \bullet)\}$
if cas(chan, 1, 0)
then $\{P\}$ flip() $\{\mathsf{ret}.\ \mathsf{ret} = () \wedge j \rightarrowtail_s K[()]\}$
else $\{(\theta, I, \mathsf{Accepted}(j, K), \bullet)\}$ chan := 0 $\{\mathsf{ret}.\ \mathsf{ret} = () \wedge ((\theta, I, \mathsf{Empty}, \varnothing) * j \rightarrowtail_s K[()])\}$
else $\{P\}$ flip() $\{\mathsf{ret}.\ \mathsf{ret} = () \wedge j \rightarrowtail_s K[()]\}$

Figure 6.4: Proof outline for redFlag

We can gain more insight by examining the varied uses of the SHARED rule in proving each **cas**.

For the first **cas**, the protocol may be in any state, but the **cas** will only succeed if the state is Offered:

$$\{P\}\ \mathbf{cas}(\mathsf{chan}, 1, 2)\ \{\mathsf{ret}.\ (\mathsf{ret} = \mathbf{true} * j \rightarrowtail_s K[()]) \lor (\mathsf{ret} = \mathbf{false} * P)\}$$

$j \rightarrowtail_s K[\mathsf{flipBody}_s] * I(\mathsf{Empty})$	$\mathsf{ret}.\ \mathsf{ret} = \mathbf{false} * j \rightarrowtail_s K[\mathsf{flipBody}_s] * I(\mathsf{Empty})$
$j \rightarrowtail_s K[\mathsf{flipBody}_s] * I(\mathsf{Offered}(j', K'))$	$\mathsf{ret}.\ \mathsf{ret} = \mathbf{true} * j \rightarrowtail_s K[()] * I(\mathsf{Accepted}(j', K'))$
$j \rightarrowtail_s K[\mathsf{flipBody}_s] * I(\mathsf{Accepted}(j', K'))$	$\mathsf{ret}.\ \mathsf{ret} = \mathbf{false} * j \rightarrowtail_s K[\mathsf{flipBody}_s] * I(\mathsf{Accepted}(j', K'))$

This table is different from those we have seen before in two ways. First, it has multiple rows, giving a case analysis of the possible protocol states given the knowledge in P, *i.e.,* that it is at least in state Empty, which means it may be in any state. Second, and relatedly, in the second row the state in the postcondition differs from that of the precondition, which requires checking that we can move from Offered to Accepted while holding no tokens.

On the other hand, the protocol's state is irrelevant to the success of the next two **cas** expressions, since they attempt to perform the flip directly:

$$\{P\}\ \mathbf{cas}(\mathsf{flag}, \mathbf{true}, \mathbf{false})\ \{\mathsf{ret}.\ (\mathsf{ret} = \mathbf{true} * j \rightarrowtail_s K[()]) \lor (\mathsf{ret} = \mathbf{false} * P)\}$$

$j \rightarrowtail_s K[\mathsf{flipBody}_s] * I(s)$	$\mathsf{ret}.\ \mathsf{ret} = \mathbf{false} * j \rightarrowtail_s K[\mathsf{flipBody}_s] * I(s)$

$$\{P\}\ \mathbf{cas}(\mathsf{flag}, \mathbf{false}, \mathbf{true})\ \{\mathsf{ret}.\ (\mathsf{ret} = \mathbf{true} * j \rightarrowtail_s K[()]) \lor (\mathsf{ret} = \mathbf{false} * P)\}$$

$j \rightarrowtail_s K[\mathsf{flipBody}_s] * I(s)$	$\mathsf{ret}.\ \mathsf{ret} = \mathbf{false} * j \rightarrowtail_s K[\mathsf{flipBody}_s] * I(s)$

The most interesting case is the final top-level **cas**. It attempts to make an offer, which succeeds only when the starting state is Empty, in which case we transfer control of our spec resource:

$$\{P\}\ \mathbf{cas}(\mathsf{chan}, 0, 1)\ \{\mathsf{ret}.\ (\mathsf{ret} = \mathbf{true} \land (\theta, I, \mathsf{Offered}(j, K), \bullet)) \lor (\mathsf{ret} = \mathbf{false} \land P)\}$$

$j \rightarrowtail_s K[\mathsf{flipBody}_s] * I(\mathsf{Empty})$	$\mathsf{ret}.\ \mathsf{ret} = \mathbf{true} * I(\mathsf{Offered}(j, K))$
$j \rightarrowtail_s K[\mathsf{flipBody}_s] * I(\mathsf{Offered}(j', K'))$	$\mathsf{ret}.\ \mathsf{ret} = \mathbf{false} * j \rightarrowtail_s K[\mathsf{flipBody}_s] * I(\mathsf{Offered}(j', K'))$
$j \rightarrowtail_s K[\mathsf{flipBody}_s] * I(\mathsf{Accepted}(j', K'))$	$\mathsf{ret}.\ \mathsf{ret} = \mathbf{false} * j \rightarrowtail_s K[\mathsf{flipBody}_s] * I(\mathsf{Accepted}(j', K'))$

Once the offer is made, we attempt to withdraw it. Withdrawing succeeds only when the offer has not been accepted. *Due to our ownership of the token*, Empty is not a possible state:

$$\{(\theta, I, \mathsf{Offered}(j, K), \bullet)\}\ \mathbf{cas}(\mathsf{chan}, 1, 0)\ \{\mathsf{ret}.\ (\mathsf{ret} = \mathbf{true} * P) \lor (\mathsf{ret} = \mathbf{false} * (\theta, I, \mathsf{Accepted}(j, K), \bullet))\}$$

$I(\mathsf{Offered}(j, K))$	$\mathsf{ret}.\ (\mathsf{ret} = \mathbf{true} \land j \rightarrowtail_s K[\mathsf{flipBody}_s]) * I(\mathsf{Empty})$
$I(\mathsf{Accepted}(j, K))$	$\mathsf{ret}.\ \mathsf{ret} = \mathbf{false} * I(\mathsf{Accepted}(j', K'))$

If we do *not* succeed in withdrawing the offer, we can conclude that the state is at least Accepted. Due to our token ownership, that is the only state we need to consider when subsequently clearing the offer field:

$$\{(\theta, I, \mathsf{Accepted}(j, K), \bullet)\}\ \mathsf{chan} := 0\ \{\mathsf{ret}.\ \mathsf{ret} = () * j \rightarrowtail_s K[()]\}$$

$I(\mathsf{Accepted}(j, K))$	$\mathsf{ret}.\ \mathsf{ret} = () * j \rightarrowtail_s K[()] * I(\mathsf{Empty})$

▸ FINALLY, we give detailed derivations of the most interesting atomic triples needed for the instantiations of the SHARED rule above. Generally, the interesting cases are those where the **cas** succeeds, or where nontrivial information about the protocol state is discovered.

The first top-level **cas** succeeds in the Offered state:

$$\langle\!| I(\mathsf{Offered}(j', K')) * j \rightarrowtail_{\mathrm{s}} K[\mathsf{flipBody}_{\mathrm{s}}] |\!\rangle$$
$$\langle\!| \mathsf{chan} \mapsto_{\mathrm{I}} 1 * Q * j' \rightarrowtail_{\mathrm{s}} K'[\mathsf{flipBody}_{\mathrm{s}}] * j \rightarrowtail_{\mathrm{s}} K[\mathsf{flipBody}_{\mathrm{s}}] |\!\rangle$$
$$\quad \mathbf{cas}(\mathsf{chan}, 1, 2)$$
$$\langle\!| \mathsf{ret}.\ (\mathsf{ret} = \mathbf{true} \wedge \mathsf{chan} \mapsto_{\mathrm{I}} 2) * Q * j' \rightarrowtail_{\mathrm{s}} K'[\mathsf{flipBody}_{\mathrm{s}}] * j \rightarrowtail_{\mathrm{s}} K[\mathsf{flipBody}_{\mathrm{s}}] |\!\rangle$$
$$\langle\!| \mathsf{ret}.\ (\mathsf{ret} = \mathbf{true} \wedge \mathsf{chan} \mapsto_{\mathrm{I}} 2) * \exists x : \mathbf{bool}.\ \mathsf{flag}_{\mathrm{I}} \mapsto_{\mathrm{I}} x * \mathsf{flag}_{\mathrm{s}} \mapsto_{\mathrm{s}} x \quad * j' \rightarrowtail_{\mathrm{s}} K'[\mathsf{flipBody}_{\mathrm{s}}] * j \rightarrowtail_{\mathrm{s}} K[\mathsf{flipBody}_{\mathrm{s}}] |\!\rangle \Rightarrow$$
$$\langle\!| \mathsf{ret}.\ (\mathsf{ret} = \mathbf{true} \wedge \mathsf{chan} \mapsto_{\mathrm{I}} 2) * \exists x : \mathbf{bool}.\ \mathsf{flag}_{\mathrm{I}} \mapsto_{\mathrm{I}} x * \mathsf{flag}_{\mathrm{s}} \mapsto_{\mathrm{s}} \neg x * j' \rightarrowtail_{\mathrm{s}} K'[()] * j \rightarrowtail_{\mathrm{s}} K[\mathsf{flipBody}_{\mathrm{s}}] |\!\rangle \Rightarrow$$
$$\langle\!| \mathsf{ret}.\ (\mathsf{ret} = \mathbf{true} \wedge \mathsf{chan} \mapsto_{\mathrm{I}} 2) * \exists x : \mathbf{bool}.\ \mathsf{flag}_{\mathrm{I}} \mapsto_{\mathrm{I}} x * \mathsf{flag}_{\mathrm{s}} \mapsto_{\mathrm{s}} x \quad * j' \rightarrowtail_{\mathrm{s}} K'[()] * j \rightarrowtail_{\mathrm{s}} K[()] |\!\rangle$$
$$\langle\!| \mathsf{ret}.\ \mathsf{ret} = \mathbf{true} * \mathsf{chan} \mapsto_{\mathrm{I}} 2 * \exists x : \mathbf{bool}.\ \mathsf{flag}_{\mathrm{I}} \mapsto_{\mathrm{I}} x * \mathsf{flag}_{\mathrm{s}} \mapsto_{\mathrm{s}} x \quad * j' \rightarrowtail_{\mathrm{s}} K'[()] * j \rightarrowtail_{\mathrm{s}} K[()] |\!\rangle$$
$$\langle\!| \mathsf{ret}.\ \mathsf{ret} = \mathbf{true} * I(\mathsf{Accepted}(j', K')) * j \rightarrowtail_{\mathrm{s}} K[()] |\!\rangle$$

We prove the second **cas** for any state s:

$$\langle\!| I(s) * j \rightarrowtail_{\mathrm{s}} K[\mathsf{flipBody}_{\mathrm{s}}] |\!\rangle$$
$$\langle\!| \exists x : \mathbf{bool}.\ \mathsf{flag}_{\mathrm{I}} \mapsto_{\mathrm{I}} x * \mathsf{flag}_{\mathrm{s}} \mapsto_{\mathrm{s}} x * I_0(s) * j \rightarrowtail_{\mathrm{s}} K[\mathsf{flipBody}_{\mathrm{s}}] |\!\rangle$$
$$\quad \mathbf{cas}(\mathsf{flag}, \mathbf{true}, \mathbf{false})$$
$$\langle\!| \mathsf{ret}.\ ((\mathsf{ret} = \mathbf{true} * \mathsf{flag}_{\mathrm{I}} \mapsto_{\mathrm{I}} \mathbf{false} * \mathsf{flag}_{\mathrm{s}} \mapsto_{\mathrm{s}} \mathbf{true}) \vee (\mathsf{ret} = \mathbf{false} * \mathsf{flag}_{\mathrm{I}} \mapsto_{\mathrm{I}} \mathbf{false} * \mathsf{flag}_{\mathrm{s}} \mapsto_{\mathrm{s}} \mathbf{false}))$$
$$\quad * I_0(s) * j \rightarrowtail_{\mathrm{s}} K[\mathsf{flipBody}_{\mathrm{s}}] |\!\rangle$$
$$\langle\!| \mathsf{ret}.\ (\mathsf{ret} = \mathbf{true} * I(s) * j \rightarrowtail_{\mathrm{s}} K[()])$$
$$\quad \vee\ (\mathsf{ret} = \mathbf{false} * I(s) * j \rightarrowtail_{\mathrm{s}} K[\mathsf{flipBody}_{\mathrm{s}}]) |\!\rangle$$

The proof for $\mathbf{cas}(\mathsf{flag}, \mathbf{false}, \mathbf{true})$ is symmetric.

That leaves the final top-level **cas**, in which we make an offer:

$$\langle\!| I(\mathsf{Empty}) * j \rightarrowtail_{\mathrm{s}} K[\mathsf{flipBody}_{\mathrm{s}}] |\!\rangle$$
$$\langle\!| \mathsf{chan} \mapsto_{\mathrm{I}} 0 * Q * j \rightarrowtail_{\mathrm{s}} K[\mathsf{flipBody}_{\mathrm{s}}] |\!\rangle$$
$$\quad \mathbf{cas}(\mathsf{chan}, 0, 1)$$
$$\langle\!| \mathsf{ret}.\ (\mathsf{ret} = \mathbf{true} \wedge \mathsf{chan} \mapsto_{\mathrm{I}} 1) * Q * j \rightarrowtail_{\mathrm{s}} K[\mathsf{flipBody}_{\mathrm{s}}] |\!\rangle$$
$$\langle\!| \mathsf{ret}.\ \mathsf{ret} = \mathbf{true} * I(\mathsf{Offered}(j, K)) |\!\rangle$$

We are now in a state where we own the token. For the inner **cas**, we therefore need to consider only two possible future states—Offered and Accepted:

$$\langle\!| I(\mathsf{Offered}(j, K)) |\!\rangle$$
$$\langle\!| \mathsf{chan} \mapsto_{\mathrm{I}} 1 * Q * j \rightarrowtail_{\mathrm{s}} K[\mathsf{flipBody}_{\mathrm{s}}] |\!\rangle$$
$$\quad \mathbf{cas}(\mathsf{chan}, 1, 0)$$
$$\langle\!| \mathsf{ret}.\ (\mathsf{ret} = \mathbf{true} \wedge \mathsf{chan} \mapsto_{\mathrm{I}} 0) * Q * j \rightarrowtail_{\mathrm{s}} K[\mathsf{flipBody}_{\mathrm{s}}] |\!\rangle$$
$$\langle\!| \mathsf{ret}.\ \mathsf{ret} = \mathbf{true} * I(\mathsf{Empty}) * j \rightarrowtail_{\mathrm{s}} K[\mathsf{flipBody}_{\mathrm{s}}] |\!\rangle$$

$\langle\!| I(\mathsf{Accepted}(j, K)) |\!\rangle$
$\langle\!| \mathsf{chan} \mapsto_{\mathrm{I}} 2 * Q * j \rightarrowtail_{\mathrm{S}} K[()] |\!\rangle$
  cas(chan, 1, 0)
$\langle\!| \mathsf{ret}.\ (\mathsf{ret} = \mathbf{false} \wedge \mathsf{chan} \mapsto_{\mathrm{I}} 2) * Q * j \rightarrowtail_{\mathrm{S}} K[()] |\!\rangle$
$\langle\!| \mathsf{ret}.\ \mathsf{ret} = \mathbf{false} * I(\mathsf{Accepted}(j, K)) |\!\rangle$

Finally, if the inner **cas** fails, there is only one rely-future state: Accepted(j, K). Thus, we know exactly what the assignment to the channel will see:

$\langle\!| I(\mathsf{Accepted}(j, K)) |\!\rangle$
$\langle\!| Q * \mathsf{chan} \mapsto_{\mathrm{I}} 2 * j \rightarrowtail_{\mathrm{S}} K[()] |\!\rangle$
  chan := 0
$\langle\!| Q * \mathsf{chan} \mapsto_{\mathrm{I}} 0 * j \rightarrowtail_{\mathrm{S}} K[()] |\!\rangle$
$\langle\!| I(\mathsf{Empty}) * j \rightarrowtail_{\mathrm{S}} K[()] |\!\rangle$

6.5 MICHAEL AND SCOTT'S QUEUE

The examples we have worked through so far use simple (single-word) data representations. To see spatial locality in action, we now examine a *linked* data structure: the Michael-Scott queue. For reference, the code and specification are given in Figure 6.5 and the protocol is given in Figure 6.6.[9]

[9] An explanation of these details is given in §4.2.

6.5.1 *The protocol*

▸ QUEUE SPECIFICATION.

```
CSQ: ∀α. (unit → ref?(α)) × (α → unit)
CGQ ≜ Λ.
  let head = new (null)

    deq  = case get head
             of n    ⇒ head := get(n[2]);
                       new (get(n[1]))
              | null ⇒ null,

    enq  = λx. let enq′ = λc. case get(c[2])
                               of c′   ⇒ enq′(c′)
                                | null ⇒ c[2] := new (x, null)
                         in case get(head)
                              of n    ⇒ enq′(n)
                               | null ⇒ head := new (x, null)

  in mkAtomic(deq, enq)
```

▸ QUEUE IMPLEMENTATION.

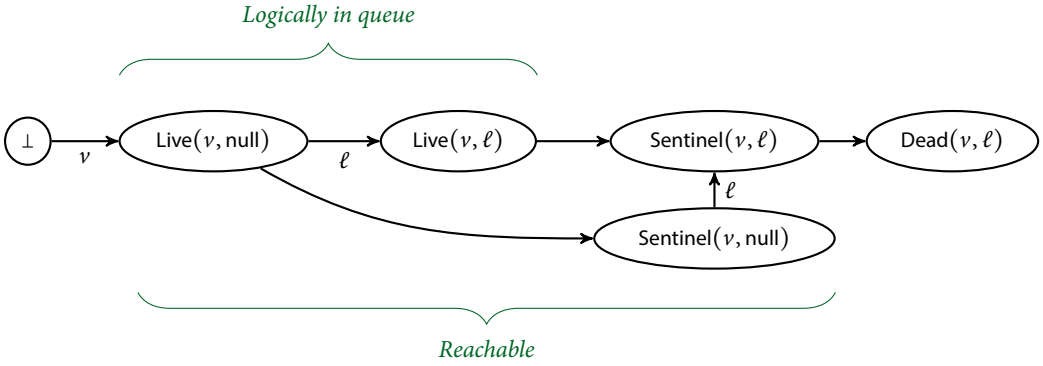
```
MSQ: ∀α. (unit → ref?(α)) × (α → unit)
MSQ ≜ Λ.
  let head = new (new (null, null))

    deq  = let n = get head
           in case get(n[2])
                of n′   ⇒ if cas(head, n, n′)
                          then get(n′[1]) else deq()
                 | null ⇒ null,

    enq  = λx. let n = new (new x, null)
               let try = λc. case get(c[2])
                               of c′   ⇒ try(c′)
                                | null ⇒ if cas(c[2], null, n)
                                         then () else try(c)
               in try(get head)

  in (deq, enq)
```

Figure 6.5: The queues

▸ PER-NODE STATE SPACE

$$S_0 \triangleq \{\bot\} \cup \{\mathsf{Live}(v, v') \mid v, v' \in \mathrm{Val}\} \cup \{\mathsf{Sentinel}(v, v') \mid v, v' \in \mathrm{Val}\} \cup \{\mathsf{Dead}(v, \ell) \mid v \in \mathrm{Val}, \ell \in \mathrm{Loc}\}$$

▸ PER-NODE PROTOCOL



▸ GLOBAL PROTOCOL STATE SPACE AND INTERPRETATION

$S \triangleq \mathrm{Loc} \xrightarrow{\text{fin}} S_0$

$s \rightsquigarrow s' \quad \text{iff} \quad \forall \ell.\ s(\ell) \rightsquigarrow_0 s'(\ell) \ \vee \ s(\ell) = s'(\ell)$

$$I(s) \triangleq \mathit{DeadSeg}(s_D, -, \ell) \ * \begin{pmatrix} \mathsf{head}_\mathrm{I} \mapsto_\mathrm{I} \ell \ * \ \ell \mapsto_\mathrm{I} (v_0, v_\mathrm{I}) \ * \\ \mathsf{head}_\mathrm{S} \mapsto_\mathrm{S} v_\mathrm{S} \ * \ \mathsf{lock}_\mathrm{S} \mapsto_\mathrm{S} \textbf{false} \end{pmatrix} * \mathit{LiveSeg}(s_L, v_\mathrm{I}, v_\mathrm{S}, \textbf{null}, \textbf{null})$$

for any $\ell, v_0, v_\mathrm{I}, v_\mathrm{S}, s_D, s_L$
with $s = s_D \uplus [\ell \mapsto \mathsf{Sentinel}(v_0, v_\mathrm{I})] \uplus s_L$, where:

$$\begin{aligned}
\mathit{DeadSeg}(\varnothing,\ \ell,\ \ell'') &\triangleq \mathsf{emp} \wedge \ell = \ell'' \\
\mathit{DeadSeg}(s_D \uplus [\ell \mapsto \mathsf{Dead}(v, \ell')],\ \ell,\ \ell'') &\triangleq \ell \mapsto_\mathrm{I} (v, \ell') \ * \ \mathit{DeadSeg}(s_D, \ell', \ell'') \\
\\
\mathit{LiveSeg}(\varnothing,\ v_\mathrm{I},\ v_\mathrm{I}'',\ v_\mathrm{S},\ v_\mathrm{S}'') &\triangleq \mathsf{emp} \wedge v_\mathrm{I} = v_\mathrm{I}'' \wedge v_\mathrm{S} = v_\mathrm{S}'' \\
\mathit{LiveSeg}(s_L \uplus [v_\mathrm{I} \mapsto \mathsf{Live}(v_0, v_\mathrm{I}')],\ v_\mathrm{I},\ v_\mathrm{I}'',\ v_\mathrm{S},\ v_\mathrm{S}'') &\triangleq \exists x_\mathrm{I}, x_\mathrm{S}, v_\mathrm{S}'.\ x_\mathrm{I} \preceq^{\mathcal{V}} x_\mathrm{S} : \alpha \ * \ v_0 \mapsto_\mathrm{I} x_\mathrm{I} \\
&\quad * \ v_\mathrm{I} \mapsto_\mathrm{I} (v_0, v_\mathrm{I}') \ * \ v_\mathrm{S} \mapsto_\mathrm{S} (x_\mathrm{S}, v_\mathrm{S}') \\
&\quad * \ \mathit{LiveSeg}(s_L, v_\mathrm{I}', v_\mathrm{I}'', v_\mathrm{S}', v_\mathrm{S}'')
\end{aligned}$$

Figure 6.6: The protocol for MSQ

Recall that the "global" protocol for the queue is given as a product STS of the local protocol (the "local life story") governing individual nodes. So a state $s \in S$ in the global STS is a function from heap locations to node states (drawn from S_0). Viewed differently,[10] s is a *partial* function defined on those locations for which a node has at some point been "born," *i.e.,* locations at non-$\bot$ states.

As discussed in §4.2, while the *transition relation* of the global protocol treats each node[11] independently, the *interpretation* of its states does not. In particular, the interpretation ensures that:

- There is exactly one Sentinel node.

[10] Punning the $\bot$ *state* in S_0 with "undefined".

[11] *i.e.,* each heap location.

- There is exactly one node with a **null** tail.
- A node is Live iff it is reachable from the Sentinel.

6.5.2 *Spatial locality*

Despite providing global guarantees, the interpretation in Figure 6.6 is defined via local (but recursive!) constraints, and is designed to support *spatially-local* reasoning.

Any notion of "locality" is intimately tied to a corresponding notion of "separation": locality demands that we distinguish resources "here" from other resources "somewhere else," a distinction only possible if we can separate resources into smaller pieces (and later recombine them). Given a notion of separation, spatial locality means that updates to resources "here" do not require or affect knowledge about resources "somewhere else."[12]

[12] These observations are the essence of separation logic, and can be formulated in a very abstract way (Calcagno *et al.* 2007).

The MSQ protocol provides an *abstract* notion of separation $\uplus$ at the level of global protocol states that is closely aligned with *physical* separation in the interpretation. Using abstract separation, we can focus attention on the abstract state of some node(s) of interest, while treating the remaining node states as an opaque "frame." The protocol interpretation is defined so that this abstract way of "focusing" on a node corresponds to a physical one as well. For example, suppose we know that the node at location ℓ is at state $\mathsf{Dead}(x, \ell')$, so that the global STS state is $s_F \uplus [\ell \mapsto \mathsf{Dead}(x, \ell')]$ for some "frame" s_F. Then there is some P_F such that

$$I(s_F \uplus [\ell \mapsto \mathsf{Dead}(x, \ell')]) \iff P_F * \mathit{DeadSeg}([\ell \mapsto \mathsf{Dead}(x, \ell')], \ell, \ell')$$

or, more to the point,

$$\forall s_F.\ \exists P_F.\quad I(s_F \uplus [\ell \mapsto \mathsf{Dead}(x, \ell')]) \iff P_F * \ell \mapsto_{\mathrm{I}} (x, \ell')$$

This simple lemma supports local reasoning about dead nodes: from local *abstract* knowledge about such nodes (*e.g.,* $[\ell \mapsto \mathsf{Dead}(x, \ell')]$), we derive local *physical* knowledge—enough to support reading their contents, for example.

To support this kind of localized reasoning in general, we next introduce a bit of shorthand. Assuming that θ is the global STS given in Figure 9.1, we set

$$n \propto s_0 \triangleq (\theta, I, [n \mapsto s_0], \varnothing)$$

Thus, if we want to assert that ℓ is dead, we can say $\ell \propto \mathsf{Dead}(x, \ell')$. Because island assertions are implicitly rely-closed (Chapter 5), the assertion says that the global state is some *rely-future state* of $s = [\ell \mapsto \mathsf{Dead}(x, \ell')]$. Even though s itself sends every location other than ℓ to state $\bot$,[13] a rely-future state of s might send these other locations to a future state of $\bot$—*i.e.,* to any state whatsoever.

[13] Recall the pun with partial functions.

In fact, the global state *must* be strictly in the rely-future of s: the interpretation $I(s)$ is unsatisfiable, so s itself is not a valid state for the STS. (The

interpretation of a global state requires, in particular, that the state has *exactly one* node as Sentinel, which is not true of s). So an assertion $n \propto s_0$ claims not just that n is in at least state s_0, but also that the state of other locations suffice to make the STS interpretation satisfiable. This is a subtle, but key point; it enables us to draw conclusions by using some local knowledge and the interpretation in tandem. For example, we have:

$$n \propto \mathsf{Live}(-, m) \land m \neq \mathbf{null} \implies n \propto \mathsf{Live}(-, m) * m \propto \mathsf{Live}(-, -)$$

Thus, from abstract local knowledge about one node, we gain abstract local knowledge about its neighbor.[14] Note that the *actual* state of both m and n could be, for example, Dead—a state that is in the rely-future of Live. This simple-seeming implication is actually a key property for verifying both enq and deq, as we will see below.

[14] The implication is a simple consequence of the way that *LiveSeg* is defined.

Figure 6.7: Proof for enq

Let $P \triangleq j \rightarrowtail_{\mathrm{s}} K[\mathsf{enqBody}_{\mathrm{s}}] * \exists \ell.\ n \mapsto_{\mathrm{I}} (\ell, \mathbf{null}) * \ell \mapsto_{\mathrm{I}} x_{\mathrm{I}}$

$Q \triangleq \forall c.\ \{P * c \propto \mathsf{Live}(-,-)\}\ \mathsf{try}(c)\ \{\mathsf{ret}.\ \mathsf{ret} = () \land j \rightarrowtail_{\mathrm{s}} K[()]\}$

Outline for try

$\{P * c \propto \mathsf{Live}(-,-)\}$
let t = **get**$(c[2])$
$\{P * c \propto \mathsf{Live}(-, t)\}$
case t
of $c' \Rightarrow$
$\{P * (c \propto \mathsf{Live}(-, c') \land c' \neq \mathbf{null})\}$
$\{P * c \propto \mathsf{Live}(-, c') * c' \propto \mathsf{Live}(-,-)\}$
$\{P * c' \propto \mathsf{Live}(-,-)\}$
$\mathsf{try}(c')$
$\{\mathsf{ret}.\ \mathsf{ret} = () \land j \rightarrowtail_{\mathrm{s}} K[()]\}$
| **null** $\Rightarrow$
$\{P * c \propto \mathsf{Live}(-, \mathbf{null})\}$
if cas$(c[2], \mathbf{null}, n)$
then
$\{j \rightarrowtail_{\mathrm{s}} K[()] * c \propto \mathsf{Live}(-, n) * n \propto \mathsf{Live}(-, \mathbf{null})\}$
$()$
$\{\mathsf{ret}.\ \mathsf{ret} = () \land j \rightarrowtail_{\mathrm{s}} K[()]\}$
else
$\{P * c \propto \mathsf{Live}(-,-)\}$
$\mathsf{try}(c)$
$\{\mathsf{ret}.\ \mathsf{ret} = () \land j \rightarrowtail_{\mathrm{s}} K[()]\}$

Outline for enq

$\{j \rightarrowtail_{\mathrm{s}} K[\mathsf{enqBody}_{\mathrm{s}}] * (\theta, J, \varnothing, \varnothing)\}$
let n = **new** (**new** x, **null**)
$\{P * (\theta, J, \varnothing, \varnothing)\}$
let try = ...
$\{(P * (\theta, J, \varnothing, \varnothing)) \land Q\}$
let t = **get** head
$\{(P * t \propto \mathsf{Live}(-,-)) \land Q\}$
in try(t)
$\{\mathsf{ret}.\ \mathsf{ret} = () \land j \rightarrowtail_{\mathrm{s}} K[()]\}$

6.5.3 *The proof: enq*

We begin with the verification of enq, shown in Figure 6.7, which is less subtle[15] than that for deq.

[15] Surprisingly, given its traversal of the queue!

To prove enq correct, we first characterize its inner loop (try) as follows:

$$\forall c.\ \{j \rightarrowtail_{\mathrm{S}} K[\mathsf{enqBody}_{\mathrm{S}}] * \exists \ell.\ n \mapsto_{\mathrm{I}} (\ell, \mathbf{null}) * \ell \mapsto_{\mathrm{I}} x_{\mathrm{I}} * c \propto \mathsf{Live}(-,-)\}\ \mathsf{try}(c)\ \{\mathsf{ret}.\ \mathsf{ret} = () \wedge j \rightarrowtail_{\mathrm{S}} K[()]\}$$

Since try is a tail-recursive function, the precondition here acts as a loop invariant.[16] Going piece-by-piece, it says that try assumes:

[16] As usual, we unroll the recursion using UnrollRec.

- Private ownership of the spec code $K[\mathsf{enqBody}_{\mathrm{S}}]$ for enqueuing,
- Private ownership of the node n to insert, the first component of which is a privately owned reference to x_{I}, and
- That the "current node" c of the traversal is at least in the Live state. The node c could in reality be the Sentinel or even Dead (*i.e.,* no longer reachable from head), but by placing a lower-bound of Live we guarantee that c was, at some point in the past, part of the queue.

In the code for try, we have added some intermediate **let**-bindings to the code given in Chapter 4, *e.g.,* the binding for t, which helps keep the proof outline clear and concise.

In the first step, we read the second component $c[2]$ from the current node c, allowing us to enrich our knowledge from $c \propto \mathsf{Live}(-,-)$ to $c \propto \mathsf{Live}(-,t)$. How much richer is this new knowledge? It depends on t. If t is **null**, we have not learned much, since the protocol allows c to move from $\mathsf{Live}(-,\mathbf{null})$ to *e.g.,* $\mathsf{Live}(-,x)$ for any x. If, on the other hand, t is non-**null**, then we know that the second component of c will *forever* remain equal to t. (See the protocol in Figure 6.6.)

In the next step, we perform a case analysis on t that tells us which of the above situations obtains:

- Suppose t is some non-**null** value; call it c'. The combined knowledge $c \propto \mathsf{Live}(-,c')$ and $c' \neq \mathbf{null}$ is enough to deduce $c' \propto \mathsf{Live}(-,-)$, as we explained in §6.5.2. And that, in turn, is enough knowledge to satisfy the precondition for a recursive call to try, this time starting from c'. To summarize the story so far: we can safely move from one used-to-be-Live node to the next, using only local, abstract knowledge about the individual nodes—despite the fact that both nodes might be *currently* unreachable from head.
- Suppose instead that t is **null**. As we explained above, according to the protocol this tells us nothing about the *current* value of $c[2]$. *This is the essential reason why the algorithm uses **cas** in the next step*: because **cas** allows us to combine instantaneous knowledge (in this case, a re-check that $c[2]$ is **null**) with instantaneous action (in this case, setting $c[2]$ to n).

- If the **cas** succeeds, then *our* thread is the one that moves c from abstract state $\mathsf{Live}(-, \mathbf{null})$ to $\mathsf{Live}(-, n)$. Making this move in the protocol requires us to move n from abstract state $\bot$ to $\mathsf{Live}(-, \mathbf{null})$,[17] which in turn requires us to transfer ownership of n and its first component into the shared island.
- Finally, if the **cas** fails, we nevertheless still know that $c \propto \mathsf{Live}(-,-)$, and we retain ownership of n and its first component. Thus, we can safely restart traversal from c.

[17] Note that n *must* be in state $\bot$; otherwise, the protocol interpretation would force the shared island to own n, which we own privately.

$\{j \rightarrowtail_{\mathrm{s}} K[\mathsf{deqBody}_{\mathrm{s}}] \wedge (\theta, J, \varnothing, \varnothing)\}$
let n = **get** head
$\{j \rightarrowtail_{\mathrm{s}} K[\mathsf{deqBody}_{\mathrm{s}}] \wedge n \propto \mathsf{Sentinel}(-,-)\}$
let t = **get**$(n[2])$
$\left\{n \propto \mathsf{Sentinel}(-,t) \wedge \left(\begin{array}{ll} & t = \mathbf{null} \wedge j \rightarrowtail_{\mathrm{s}} K[\mathbf{null}] \\ \oplus & t \neq \mathbf{null} \wedge j \rightarrowtail_{\mathrm{s}} K[\mathsf{deqBody}_{\mathrm{s}}] \end{array}\right)\right\}$
in case t
of $n' \Rightarrow$
$\{j \rightarrowtail_{\mathrm{s}} K[\mathsf{deqBody}_{\mathrm{s}}] \wedge n \propto \mathsf{Sentinel}(-,n') \wedge n' \neq \mathbf{null}\}$
$\{j \rightarrowtail_{\mathrm{s}} K[\mathsf{deqBody}_{\mathrm{s}}] \wedge n \propto \mathsf{Sentinel}(-,n') \wedge n' \propto \mathsf{Live}(-,-)\}$
if cas(head, n, n')
then
$\left\{\begin{array}{l} \exists x_{\mathrm{I}}, x_{\mathrm{s}}.\ x_{\mathrm{I}} \preceq^{\mathcal{V}} x_{\mathrm{s}} : \alpha * \exists y_{\mathrm{I}}, y_{\mathrm{s}}.\ y_{\mathrm{I}} \mapsto_{\mathrm{I}} x_{\mathrm{I}} * y_{\mathrm{s}} \mapsto_{\mathrm{s}} x_{\mathrm{s}} * j \rightarrowtail_{\mathrm{s}} K[y_{\mathrm{s}}] \\ * \ n \propto \mathsf{Dead}(-,n') * n' \propto \mathsf{Sentinel}(y_{\mathrm{I}},-) \end{array}\right\}$
$\{\exists y_{\mathrm{I}}, y_{\mathrm{s}}.\ y_{\mathrm{I}} \preceq^{\mathcal{V}} y_{\mathrm{s}} : \mathbf{ref}_?(\alpha) * j \rightarrowtail_{\mathrm{s}} K[y_{\mathrm{s}}] * n' \propto \mathsf{Sentinel}(y_{\mathrm{I}},-)\}$
get$(n'[1])$
$\{y_{\mathrm{I}}.\ \exists y_{\mathrm{s}}.\ y_{\mathrm{I}} \preceq^{\mathcal{V}} y_{\mathrm{s}} : \mathbf{ref}_?(\alpha) * j \rightarrowtail_{\mathrm{s}} K[y_{\mathrm{s}}]\}$
else
$\{j \rightarrowtail_{\mathrm{s}} K[\mathsf{deqBody}_{\mathrm{s}}] \wedge (\theta, J, \varnothing, \varnothing)\}$
deq()
$\{y_{\mathrm{I}}.\ \exists y_{\mathrm{s}}.\ y_{\mathrm{I}} \preceq^{\mathcal{V}} y_{\mathrm{s}} : \mathbf{ref}_?(\alpha) * j \rightarrowtail_{\mathrm{s}} K[y_{\mathrm{s}}]\}$
| **null** $\Rightarrow$
$\{j \rightarrowtail_{\mathrm{s}} K[\mathbf{null}]\}$
null
$\{\mathsf{ret}.\ \mathsf{ret} = \mathbf{null} \wedge j \rightarrowtail_{\mathrm{s}} K[\mathbf{null}]\}$
$\{y_{\mathrm{I}}.\ \exists y_{\mathrm{s}}.\ y_{\mathrm{I}} \preceq^{\mathcal{V}} y_{\mathrm{s}} : \mathbf{ref}_?(\alpha) * j \rightarrowtail_{\mathrm{s}} K[y_{\mathrm{s}}]\}$

Figure 6.8: Proof outline for deq

6.5.4 *The proof: deq*

The code for deq begins by simply reading head. The node read,[18] n, is at least in the sentinel state.

The next step reads the second component of n, which will ultimately tell whether the queue "is" empty. But, of course, a concurrent queue that is momentarily empty may gain elements in the next instant. So when the code

[18] By the interpretation of the protocol, we know there must be such a node.

reads $n[2]$ into t—but before it *inspects* the value of t—we perform a subtle maneuver in the logic: we "inspect" the value of t within the logic (using speculative choice), and in the case that t is **null** *we speculatively execute the spec*.[19] It is crucial to do so at this juncture, because by the time the code itself discovers that the value t is **null**, the value of $n[2]$ may have changed—and the contents of the queue on the specification side will have changed with it.

Assuming that t is not **null**, we can deduce that it points to a node n' that is at least in the Live state.[20] But the knowledge we have accumulated so far about n and n' gives only a lower bound on their abstract states. So in the next step, the code performs a **cas**, which atomically re-checks that n is *still* the sentinel, and at the same time updates head to point to n':

- If successful, n will be Dead, n' will be the new Sentinel, and we will have gained ownership of the first component of n. We then package up that first component in a **ref** island, and return it.
- Otherwise, we drop essentially all our knowledge on the floor, and retry.

If on the other hand t is **null**, there is little to show: we have already speculatively executed the specification, which must have returned **null**—the same value returned by the implementation.

[19] Using speculative choice $\oplus$ is not strictly necessary: we could have used vanilla disjunction instead, because the added conditions on t ensure that only one branch of the speculation is ever in play.

[20] This is yet another application of the lemma given in §6.5.2.

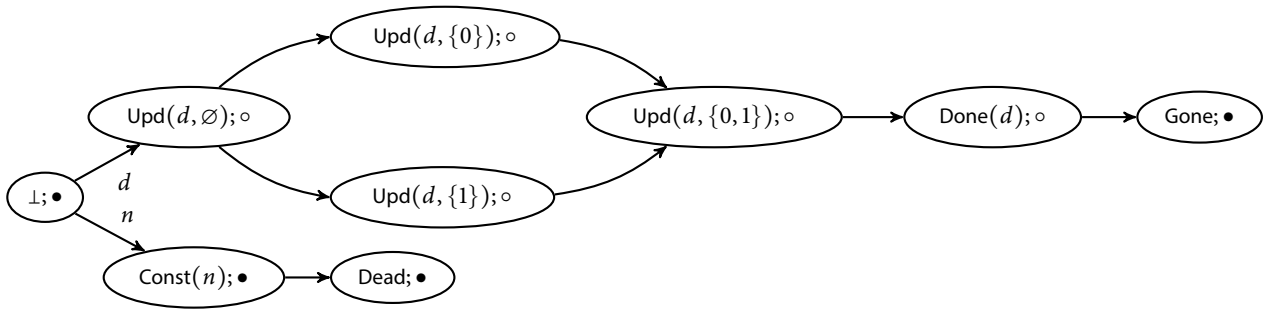
▸ CONDITIONAL COUNTER SPECIFICATION.

$$
\begin{array}{l}
\mathsf{counter_S} \triangleq \\
\quad \textbf{let } c = \textbf{new } 0, f = \textbf{new false} \\
\quad \textbf{let } \mathsf{setFlag}(b) = f := b \\
\quad \textbf{let } \mathsf{get}() \quad = \textbf{get } c \\
\quad \textbf{let } \mathsf{cinc}() \quad = c := \textbf{get } c + \textbf{if get } f \textbf{ then } 1 \textbf{ else } 0 \\
\quad \textbf{in } \mathsf{mkAtomic}(\mathsf{setFlag}, \mathsf{get}, \mathsf{cinc})
\end{array}
$$

▸ CONDITIONAL COUNTER IMPLEMENTATION.

$$
\begin{array}{l}
\mathsf{counter_I} \triangleq \\
\quad \textbf{let } c = \textbf{new inj}_1\ 0, f = \textbf{new false} \\
\quad \textbf{let } \mathsf{setFlag}(b) = f := b \\
\quad \textbf{let } \mathsf{complete}(x, n) = \\
\quad\quad \textbf{if get } f \textbf{ then cas}(c, x, \textbf{inj}_1\ (n+1)) \\
\quad\quad\quad\quad \textbf{else cas}(c, x, \textbf{inj}_1\ n) \\
\quad \textbf{let rec } \mathsf{get}() = \textbf{let } x = \textbf{get } c \textbf{ in case } x \textbf{ of} \\
\quad\quad \textbf{inj}_1\ n \Rightarrow n \\
\quad\quad \mid \textbf{inj}_2\ n \Rightarrow \mathsf{complete}(x, n); \mathsf{get}() \\
\quad \textbf{let rec } \mathsf{cinc}() = \textbf{let } x = \textbf{get } c \textbf{ in case } x \textbf{ of} \\
\quad\quad \textbf{inj}_1\ n \Rightarrow \textbf{let } y = \textbf{inj}_2\ n \textbf{ in} \\
\quad\quad\quad \textbf{if cas}(c, x, y) \textbf{ then } \mathsf{complete}(y, n); () \\
\quad\quad\quad\quad\quad\quad \textbf{else } \mathsf{cinc}() \\
\quad\quad \mid \textbf{inj}_2\ n \Rightarrow \mathsf{complete}(x, n); \mathsf{cinc}() \\
\quad \textbf{in } (\mathsf{setFlag}, \mathsf{get}, \mathsf{cinc})
\end{array}
$$

▸ PER-CELL PROTOCOL.



▸ GLOBAL PROTOCOL AND INTERPRETATION.

$$d ::= n, j, K \qquad B \subseteq \{0,1\} \qquad A \triangleq \mathsf{Loc} \qquad S_0 \triangleq \{\bot, \mathsf{Upd}(d,B), \mathsf{Done}(d), \mathsf{Gone}, \mathsf{Const}(n), \mathsf{Dead}\} \qquad S \triangleq \mathsf{Loc} \xrightarrow{\text{fin}} S_0$$

$$
\begin{array}{l}
I(s) \triangleq \exists b : \textbf{bool}.\ f_{\mathrm{I}} \mapsto_{\mathrm{I}} b * f_{\mathrm{S}} \mapsto_{\mathrm{S}} b * \mathsf{lock}_{\mathrm{S}} \mapsto_{\mathrm{S}} \textbf{false} \\
\quad * \exists! \ell_c.\ s(\ell) \in \{\mathsf{Const}(-), \mathsf{Upd}(-,-)\} \\
\quad * \begin{cases} \mathsf{linkUpd}(\ell_c, n, j, K, B) & s(\ell_c) = \mathsf{Upd}(n, j, K, B) \\ \mathsf{linkConst}(\ell_c, n) & s(\ell_c) = \mathsf{Const}(n) \end{cases} \\
\quad * \ \ast_{s(\ell)=\mathsf{Done}(n,j,K)} \ell \mapsto_{\mathrm{I}} \textbf{inj}_2\ n * j \twoheadrightarrow_{\mathrm{S}} K[()] \\
\quad * \ \ast_{s(\ell)=\mathsf{Gone}} \ell \mapsto_{\mathrm{I}} \textbf{inj}_2\ - \quad * \quad \ast_{s(\ell)=\mathsf{Dead}} \ell \mapsto_{\mathrm{I}} \textbf{inj}_1\ -
\end{array}
$$

$$\mathsf{linkConst}(\ell_c, n) \triangleq c_{\mathrm{I}} \mapsto_{\mathrm{I}} \ell_c * \ell_c \mapsto_{\mathrm{I}} \textbf{inj}_1\ n * c_{\mathrm{S}} \mapsto_{\mathrm{S}} n$$

$$
\begin{array}{l}
\mathsf{linkUpd}(\ell_c, n, j, K, B) \triangleq c_{\mathrm{I}} \mapsto_{\mathrm{I}} \ell_c * \ell_c \mapsto_{\mathrm{I}} \textbf{inj}_2\ n \\
\quad * \begin{pmatrix} c_{\mathrm{S}} \mapsto_{\mathrm{S}} n * j \twoheadrightarrow_{\mathrm{S}} K[\mathsf{cincBody_S}] & \\ \oplus\ c_{\mathrm{S}} \mapsto_{\mathrm{S}} n * j \twoheadrightarrow_{\mathrm{S}} K[()] & \text{if } 0 \in B \\ \oplus\ c_{\mathrm{S}} \mapsto_{\mathrm{S}} (n+1) * j \twoheadrightarrow_{\mathrm{S}} K[()] & \text{if } 1 \in B \end{pmatrix}
\end{array}
$$

Figure 6.9: Conditional increment, a simplification of CCAS

6.6 CONDITIONAL CAS

We are now in a position to tackle, in detail, a rather complex scalable concurrent algorithm: Harris *et al.*'s *conditional CAS*,[21] which performs a compare-and-set on one word of memory, but only succeeds when some other word (the *control flag*) is non-zero at the same instant. This data structure is the workhorse that enables Harris *et al.* to build their remarkable lock-free multi-word CAS from single-word CAS.

[21] Timo Harris *et al.* (2002), "A practical multi-word compare-and-swap operation", and Fraser and Tim Harris (2007), "Concurrent programming without locks"

As with the Michael-Scott queue, we have boiled down conditional CAS to its essence, retaining its key verification challenges while removing extraneous detail. Thus, we study lock-free conditional *increment* on a counter, with a fixed control flag per instance of the counter; see the specification $\mathsf{counter}_\mathsf{s}$ in Figure 6.9. These simplifications eliminate the need to track administrative information about the operation we are trying to perform but do not change the algorithm itself, so adapting our proof of conditional increment to full CCAS is a straightforward exercise.

6.6.1 *The protocol*

To explain our implementation, $\mathsf{counter}_\mathsf{i}$, we begin with its representation and the protocol that governs it. The control flag f is represented using a simple boolean reference; all of the action is in the counter c, which has type $\mathbf{ref}(\mathbf{nat} + \mathbf{nat})$. A value $\mathbf{inj}_1\ n$ represents an "inactive" counter with logical value n. A value $\mathbf{inj}_2\ n$, in contrast, means that the counter is undergoing a conditional increment, and had the logical value n when the increment began. Because $\mathbf{inj}_2\ n$ records the original value, a concurrent thread attempting another operation on the data structure can help finish the in-progress increment.[22] Helping is what makes the algorithm formally nonblocking (obstruction-free in particular; see §2.5.2): at any point in an execution, if a thread operating on the data structure is run in isolation, it will eventually finish its operation. This property precludes the use of locks, since a thread that does not hold the lock will, in isolation, forever wait by trying to acquire it.

[22] To help yourself, first help others (to get out of your way).

The question is how to perform a conditional increment without using any locks. Remarkably, the algorithm simply reads the flag f, and then—in a separate step—updates the counter c with a **cas**; see the complete function. It is possible, therefore, for one thread performing a conditional increment to read f as **true**, at which point another thread sets f to **false**; the original thread then proceeds with incrementing the counter, even though the control flag is **false**! Proving that $\mathsf{counter}_\mathsf{i}$ refines $\mathsf{counter}_\mathsf{s}$ despite this blatant race condition will require all the features of our model, working in concert.

▸ AN INITIAL IDEA is that when the *physical* value of the counter is $\mathbf{inj}_2\ n$, its *logical* value is ambiguous: it is either n or $n + 1$. This idea will only work if we can associate such logical values with feasible executions of the spec's cinc

code, since "logical" value really means the spec's value. The difficulty is in choosing *when* to take spec steps. If we wait to execute the spec code until a successful **cas** in complete, we may be too late: as the interleaving above shows, the flag may have changed by then. But we cannot execute the spec when we read the flag, either: the **cas** that follows it may fail, in which case some *other* thread must have executed the spec.

The way out of this conundrum is for threads to interact via a speculative protocol, shown in Figure 6.9. Recall that injections into sum types are heap-allocated, so every value c takes on has an identity: its location. The protocol gives the life story for every possible location in the heap as a potential value of c, with the usual constraint that all but finitely many locations are in the unborn ($\bot$) state. The first step of the protocol reflects the choice latent in the sum type:

- The location location may be a quiescent $\mathbf{inj}_1\ n$, represented initially by $\mathsf{Const}(n)$.
- Alternatively, the location may be an active increment operation $\mathbf{inj}_2\ n$, represented initially by $\mathsf{Upd}(d, \varnothing)$. The *logical descriptor* d gives the old value n of the counter, together with the thread id j and spec evaluation context K of the thread attempting the increment. The latter information is necessary because thread j temporarily donates its spec to the protocol, permitting helping threads to execute the spec on its behalf. Following the pattern laid out in Section 4.4, in return for donating its spec, thread j receives a token which will later permit it, and only it, to recover its spec. As usual, we depict the token with a bullet.

The life story for a quiescent $\mathbf{inj}_1\ n$ is quite mundane: either it is the current value pointed to by c, or it is Dead.

An active cell $\mathbf{inj}_2\ n$ leads a much more exciting life. In the first phase of life, $\mathsf{Upd}(d, B)$, the cell records which *branches* $B \subseteq \{0, 1\}$ of the complete code have been entered by a thread. Initially, no thread has executed complete, so the set is empty. If a thread subsequently reads that $f =$ **true** in the first step of executing complete, it moves to the set $\{1\}$, since it is now committed to the branch that adds 1 to the initial value n. Crucially, this step coincides with a *speculative* run of the specification; the un-run spec is also retained, in case some other thread commits to the 0 branch. The branch-accumulation process continues until some thread (perhaps not the original instigator of the increment) actually succeeds in performing its **cas** in complete. At that point, the increment is Done, and its $\mathbf{inj}_2\ n$ cell is effectively dead, but not yet Gone: in the end, the thread that instigated the original increment reclaims its spec, whose execution is guaranteed to be finished.

6.6.2 *The proof*

We now formally justify that $\mathsf{counter}_\mathrm{I}$ refines $\mathsf{counter}_\mathrm{s}$ by giving a concrete interpretation to the protocol and providing a Hoare-style proof outline for complete and cinc. The outline for get is then a straightforward exercise.

To formalize the protocol, we first give the set of states S_0 for an *individual* life story; see Figure 6.9. The states S for the data structure are then a product of individual STS states indexed by location, with all but finitely many locations required to be in state $\bot$. The set of tokens A for the product STS is just the set of locations, *i.e.,* there is one token per location (and hence per individual life story). The transition relation $\rightsquigarrow$ on the product STS lifts the one for individual life stories:

$$s \rightsquigarrow s' \triangleq \forall \ell.\ s(\ell) = s'(\ell) \vee s(\ell) \rightsquigarrow s'(\ell)$$

If F_0 is the free-token function for an individual STS, we can then define the product STS as follows:

$$\theta \quad \triangleq \quad (S, A, \rightsquigarrow, \lambda s.\{\ell \mid F_0(s(\ell)) = \{\bullet\}\})$$

The interpretation I for states of the product STS given in Figure 6.9 is fairly straightforward. The implementation and specification flag values must always match. There must exist a unique location ℓ_c ("$\exists!\ell_c$") in a "live" state of Const or Upd. This unique live location will be the one currently pointed to by c. In the Upd state, it also owns speculative spec resources according to the branch set B. Finally, Done nodes retain a finished spec, while Dead and Gone nodes are simply garbage $\mathbf{inj}_1\ (-)$ and $\mathbf{inj}_2\ (-)$ nodes, respectively.

▸ CONSTRUCTION As usual, the implementation $\mathsf{counter}_\mathrm{I}$ begins by allocating some shared, hidden data for its representation. We elide the straightforward proof outline for the construction phase, and just give its precondition,

$$j \rightarrowtail_\mathrm{s} K[\mathsf{counter}_\mathrm{s}]$$

and concrete postcondition,

$$\exists x.\ c_\mathrm{I} \mapsto_\mathrm{I} x * x \mapsto_\mathrm{I} \mathbf{inj}_1\ 0 * f_\mathrm{I} \mapsto_\mathrm{I} \mathbf{false}$$
$$* \ j \rightarrowtail_\mathrm{s} K[(\mathsf{setFlag}_\mathrm{s}, \mathsf{get}_\mathrm{s}, \mathsf{cinc}_\mathrm{s})] * c_\mathrm{s} \mapsto_\mathrm{s} 0 * f_\mathrm{s} \mapsto_\mathrm{s} \mathbf{false} * \mathsf{lock}_\mathrm{s} \mapsto_\mathrm{s} \mathbf{false}$$

To prepare to prove method refinement, we need to move these private resources into a new shared island governed by the protocol in Figure 6.9. We first use CONSEQUENCE to rewrite the postcondition in terms of the protocol with a single live cell x, *i.e.,* at state $[x \mapsto \mathsf{Const}(0)]$:

$$\exists x. I([x \mapsto \mathsf{Const}(0)]) * j \rightarrowtail_\mathrm{s} K[(\mathsf{setFlag}_\mathrm{s}, \mathsf{get}_\mathrm{s}, \mathsf{cinc}_\mathrm{s})]$$

We are thus in a position to apply the NEWISLAND rule to move these resources into an island:

$$\exists x.\ (\theta, I, [x \mapsto \mathsf{Const}(0)], \varnothing) * j \rightarrowtail_\mathrm{s} K[(\mathsf{setFlag}_\mathrm{s}, \mathsf{get}_\mathrm{s}, \mathsf{cinc}_\mathrm{s})]$$

▸ METHOD REFINEMENT We must then show, in the context of this extended island, that each of the implementation procedures refines the corresponding specification procedure. We give the detailed proof for cinc, *i.e.,*

$$\{j \rightarrowtail_{\mathsf{S}} K[\mathsf{cincBody}_{\mathsf{S}}] * (\theta, I, \varnothing, \varnothing)\}\ \mathsf{cincBody}_{\mathsf{I}}\ \{\mathsf{ret}.\ \mathsf{ret} = () \wedge j \rightarrowtail_{\mathsf{S}} K[()]\}$$

In the precondition, we weaken our knowledge about the island to simply saying that it is in a rely-future state of $\varnothing$ (where *every* location maps to $\bot$), since this is all we need to know.

The locality of the local life stories is manifested in our ability to make isolated, abstract assertions about a particular location governed by the data structure. Because every location is in some rely-future state of $\bot$, we can focus on a location x of interest by asserting that the product STS is in a rely-future state of $[x \mapsto s_0]$, where $s_0 \in S_0$.[23] For readability, we employ the following shorthand for making such local assertions about the island, with and without the token for the location in focus:

$$x \propto s_0 \triangleq (\theta, I, [x \mapsto s_0], \varnothing) \qquad x \propto_\bullet s_0 \triangleq (\theta, I, [x \mapsto s_0], \{x\})$$

Thus empowered, we can glean some additional insight about the algorithm: that the complete function satisfies the triple

$$\{x \propto \mathsf{Upd}(n, j, K, \varnothing)\}\ \mathsf{complete}(x, n)\ \{\mathsf{ret}.\ x \propto \mathsf{Done}(n, j, K)\}$$

In reading this triple, it is crucial to remember that assertions are closed under rely moves—so $x \propto \mathsf{Upd}(n, j, K, \varnothing)$ means that the location x *was once* a live, in-progress update. The interesting thing about the triple is that, regardless of the exact initial state of x, on exit we *know* that x is at least Done—and there's no going back.

The proof outline for complete is as follows:

$$\begin{array}{l}
\textbf{let}\ \mathsf{complete}(x, n) = \{x \propto \mathsf{Upd}(n, j, K, \varnothing)\} \\
\quad \textbf{if get}\ f_{\mathsf{I}}\ \textbf{then}\ \{x \propto \mathsf{Upd}(n, j, K, \{1\})\}\ \mathbf{cas}(c_{\mathsf{I}}, x, \mathbf{inj}_1\ (n+1))\ \{x \propto \mathsf{Done}(n, j, K)\} \\
\qquad\quad \textbf{else}\ \{x \propto \mathsf{Upd}(n, j, K, \{0\})\}\ \mathbf{cas}(c_{\mathsf{I}}, x, \mathbf{inj}_1\ n)\qquad \{x \propto \mathsf{Done}(n, j, K)\}
\end{array}$$

According to the proof outline, after reading the value of the flag, the location x is in an appropriately speculative state. To *prove* that fact, we must consider the rely-future states of $\mathsf{Upd}(n, j, K, \varnothing)$, and show that for each such state we can reach (via a guarantee move) a rely-future state of $\mathsf{Upd}(n, j, K, \{1\})$ or $\mathsf{Upd}(n, j, K, \{0\})$, depending on the value read from the flag. For example, if the initial state of the island is $s \uplus [x \mapsto s_0]$ and we read that the flag is **true**, we take a guarantee move to $s \uplus [x \mapsto s_0']$ as follows:[24]

If s_0 is	then s_0' is	If s_0 is	then s_0' is
$\mathsf{Upd}(d, \varnothing)$	$\mathsf{Upd}(d, \{1\})$	$\mathsf{Upd}(d, \{1\})$	$\mathsf{Upd}(d, \{1\})$
$\mathsf{Upd}(d, \{0\})$	$\mathsf{Upd}(d, \{0,1\})$	$\mathsf{Upd}(d, \{0,1\})$	$\mathsf{Upd}(d, \{0,1\})$
$\mathsf{Done}(d)$	$\mathsf{Done}(d)$	Gone	Gone

[23] And so every other location is in a rely-future state of $\bot$, *i.e.,* in an arbitrary state.

[24] This table is just a condensed version of the usual one for a use of the SHARED rule, although here we are only considering the case where the read returned **true**. The full table extends this one symmetrically for reading a **false** flag.

If the initial state for location x already included the necessary speculation (or was Done or Gone), there is nothing to show; otherwise, changing the state requires speculative execution of the spec using ASpecExec. The fact that the *rest* of the island's state s is treated as an unexamined frame here is the most direct reflection of protocol locality.

We perform a similar case analysis for both of the **cas** steps, but there we start with the knowledge that the appropriate speculation has already been performed—which is exactly what we need if the **cas** succeeds. If, on the other hand, the **cas** fails, it *must* be the case that x is at least Done: if it were still in an Upd state, the **cas** would have succeeded.

$\{j \rightarrowtail_s K[\mathsf{cincBody}_s] * (\theta, I, \varnothing, \varnothing)\}$
let x = **get** c_1 **in**
$\left\{j \rightarrowtail_s K[\mathsf{cincBody}_s] * \left(x \propto \mathsf{Const}(-) \vee x \propto \mathsf{Upd}(-,-)\right)\right\}$
case x
  of $\mathbf{inj}_1\ n \Rightarrow$
    $\{j \rightarrowtail_s K[\mathsf{cincBody}_s] * x \propto \mathsf{Const}(n)\}$
    let y = $\mathbf{inj}_2\ n$ **in**
    $\{j \rightarrowtail_s K[\mathsf{cincBody}_s] * x \propto \mathsf{Const}(n) * y \mapsto \mathbf{inj}_2\ n\}$
    if $\mathbf{cas}(c_1, x, y)$
    then
      $\{x \propto \mathsf{Dead}(n) \wedge y \propto_\bullet \mathsf{Upd}(n, j, K, \varnothing)\}$
      $\{y \propto_\bullet \mathsf{Upd}(n, j, K, \varnothing)\}$
      $\mathsf{complete}(y, n)$;
      $\{y \propto_\bullet \mathsf{Done}(n, j, K)\}$
      $()$
      $\{\mathsf{ret}.\ \mathsf{ret} = () \wedge j \rightarrowtail_s K[()] \wedge y \propto \mathsf{Gone}\}$
      $\{\mathsf{ret}.\ \mathsf{ret} = () \wedge j \rightarrowtail_s K[()]\}$
    else
      $\{j \rightarrowtail_s K[\mathsf{cincBody}_s] * (\theta, I, \varnothing, \varnothing)\}$
      $\mathsf{cinc}()$
      $\{\mathsf{ret}.\ \mathsf{ret} = () \wedge j \rightarrowtail_s K[()]\}$
  | $\mathbf{inj}_2\ n \Rightarrow$
    $\{j \rightarrowtail_s K[\mathsf{cincBody}_s] * x \propto \mathsf{Upd}(n, -, -, -)\}$
    $\mathsf{complete}(x, n)$;
    $\{j \rightarrowtail_s K[\mathsf{cincBody}_s] * x \propto \mathsf{Done}(n, -, -)\}$
    $\{j \rightarrowtail_s K[\mathsf{cincBody}_s] * (\theta, I, \varnothing, \varnothing)\}$
    $\mathsf{cinc}()$
    $\{\mathsf{ret}.\ \mathsf{ret} = () \wedge j \rightarrowtail_s K[()]\}$

Figure 6.10: Proof outline for cinc

▸ With complete out of the way, the proof of cinc is relatively easy. The proof outline is in Figure 6.10.[25] When entering the procedure, all that is known is that the island exists, and that the spec is owned. The thread first examines c_1 to see if the counter is quiescent, which is the interesting case.

[25] The steps labeled with ∴ indicate uses of the rule of consequence to weaken a post-condition.

If the subsequent **cas** succeeds in installing an active descriptor $\mathbf{inj}_2\ n$, that descriptor is the new live node (in state $\mathsf{Upd}(n, j, K, \varnothing)$)—and the thread, being responsible for this transition, gains ownership of the descriptor's token. The resulting assertion $y \propto_\bullet \mathsf{Upd}(n, j, K, \varnothing)$ is equivalent to

$$\exists i.\ i \mapsto \Big(y \propto \mathsf{Upd}(n, j, K, \varnothing)\Big) * i \mapsto \Big(y \propto_\bullet \mathsf{Upd}(n, j, K, \varnothing)\Big)$$

which means that we can use $i \mapsto y \propto_\bullet \mathsf{Upd}(n, j, K, \varnothing)$ as a *frame* in an application of the frame rule to the triple for $\mathsf{complete}(y, n)$. This gives us the framed postcondition

$$\exists i.\ i \mapsto \Big(y \propto \mathsf{Done}(n, j, K)\Big) * i \mapsto \Big(y \propto_\bullet \mathsf{Upd}(n, j, K, \varnothing)\Big)$$

which is equivalent to $y \propto_\bullet \mathsf{Done}(n, j, K)$. Since our thread still owns the token, we know the state is *exactly* $\mathsf{Done}(n, j, K)$, and in the next step (where we return the requisite unit value) we trade the token in return for our spec—which *some* thread has executed.

7

Related work: understanding concurrency

We have presented a model and logic for a **high-level language** with concurrency that enables **direct refinement proofs** for scalable concurrent algorithms, via a notion of **local protocol** that encompasses the fundamental phenomena of **role-playing**, **cooperation**, and **nondeterminism**. In this section, we survey the most closely related work along each of these axes.

"The whole thing that makes a mathematician's life worthwhile is that he gets the grudging admiration of three or four colleagues."

—Donald E. Knuth

7.1 HIGH-LEVEL LANGUAGE

There is an enormous literature on reasoning about programs in high-level languages, so we can only mention the most relevant precursors to our work. As far as we are aware, there are no prior proof methods that handle higher-order languages, shared-state concurrency, *and* local state.

7.1.1 *Representation independence and data abstraction*

John Reynolds famously asserted[1] that

> *Type structure is a syntactic discipline for enforcing levels of abstraction.*

and formalized this assertion through an "abstraction" theorem, which was later renamed to *representation independence* by John Mitchell.[2] The basic idea is very simple: a language enforces an abstraction if "benign" changes to the information hidden behind the abstraction cannot alter the behavior of its clients. In other words, client behavior is independent of the representation of the abstraction.

To formalize this idea, Reynolds adopted a *relational* view of semantics, which makes it easy to compare programs that differ only in the implementation of some abstraction. Representation independence is then a theorem about a language saying that if two implementations of an abstraction are appropriately related, the behavior of a program using one implementation is likewise related to the behavior using the other implementation. In particular, if the program returns some *concrete* output, *e.g.,* a natural number, it will return *the same* output in both cases. The formalization was an early instance of a (denotational) logical relation, and the central mantra of "related inputs produce related outputs" persists in current work in the area.

[1] Reynolds (1983), "Types, abstraction and parametric polymorphism"

[2] Mitchell (1986), "Representation independence and data abstraction"

In a language with the ability to introduce new abstractions—one with existential types, say, or with closures over mutable state—representation independence enables the reasoning technique of data abstraction, of which our approach to reasoning about concurrent data structures is one example. However, as we discussed in §3.3 and §3.4, we enable the client to reason not just in terms of simpler data, but also using a coarser grain of concurrent interaction.

A key takeaway point here is that representation independence and the data abstraction principle it enables are properties of high-level languages. In fact, enforcing abstraction boundaries could be taken as a necessary condition for *being* a high-level language. As we discussed in Chapter 3, such linguistic hiding mechanisms play an important role in real implementations of concurrent data structures, which motivated our semantic treatment in a high-level calculus like F^{μ}_{cas}.

7.1.2 *Local state*

The early treatment of linguistic hiding mechanisms focused primarily on existential types and on languages without mutable state. But of course many data structures in practice rely on *local* mutable state, *i.e.,* state that is hidden behind an abstraction boundary.

7.1.2.1 *Kripke logical relations*

Andrew Pitts pioneered an operational approach to logical relations,[3] and together with Ian Stark gave a logical relation for reasoning about functions with hidden state.[4] The language they studied included first-class functions and dynamically allocated references, but the references were limited to base type only (no "higher-order state"). Pitts and Stark use a *Kripke*-style logical relation, *i.e.,* one parameterized by a "possible world" and with an attendant notion of "future" worlds. In their model (a binary logical relation), a world is a relation on heaps, giving some invariant relationship between the heaps of two implementations of an abstraction—and thereby enabling data abstraction at the heap level. A world can be extended through a relational form of separating conjunction, *i.e.,* an additional relational invariant can be added so long as the new invariant governs disjoint pieces of the heaps. Thus, while worlds are flat relations, they are extended in an island-like fashion.

Many extensions of the Pitts-Stark model subsequently appeared. Ahmed *et al.* in particular showed how to scale the technique to a higher-order store and incorporated existential types.[5] The same paper also included an explicit notion of island—one in which the heap invariant can *evolve* over time.[6] Unfortunately, this island evolution was described in a somewhat complex way, by giving islands "populations" and "laws." Follow-up work[7] showed how the evolution of invariants can in fact be understood through simple

[3] Pitts (2002); Pitts (2005)

[4] Pitts and Stark (1998), "Operational reasoning for functions with local state"

[5] Ahmed *et al.* (2009), "State-dependent representation independence"

[6] It is folklore that this is equivalent to adding ghost state and so-called "history" invariants.

[7] Dreyer, Neis, and Birkedal (2010), "The impact of higher-order state and control effects on local relational reasoning"

state-transition systems, which were the impetus for our local protocols in Chapter 4.[8]

All of these logical relations provide semantic models through which one can reason about *e.g.,* contextual equivalence, but reasoning "in the model" is sometimes too low-level (involving arithmetic on step-indices) or too unstructured. Plotkin and Abadi (1993) gave a logic in which one can define and reason about logical relations proof-theoretically.[9] Subsequently, Dreyer *et al.* (2009) showed how to build a similar logic in the presence of step indexing with proof rules that largely eliminate the need for step arithmetic.[10] Follow-up work extended these ideas to a language with local state, proposing a logic called LADR in which one can define and reason about the logical relation of Ahmed *et al.* (2009).[11] These logics had a substantial influence on the logic presented in Chapter 5: we include a later modality and our island assertions resemble similar assertions in Dreyer, Neis, Rossberg, *et al.* (2010). There are some important differences, however. First, we kept our logic first-order (no quantification over predicates), which prevents us from *defining* the logical relation inside the logic; instead, we treat the logical relation as a particular assertion. Second, while Dreyer, Neis, Rossberg, *et al.* (2010) supports Hoare-style reasoning, we go further in defining our computation relation ($\preceq^{\mathcal{E}}$) in terms of Hoare triples, which significantly streamlines the logic.

[8] The work also showed how the presence or absence of various language features can be understood by the presence or absence of certain kinds of transitions in the STS.

[9] Plotkin and Abadi (1993), "A logic for parametric polymorphism"

[10] The logic incorporates the "later" modality first studied in Appel *et al.* (2007).

[11] Dreyer, Neis, Rossberg, *et al.* (2010), "A relational modal logic for higher-order stateful ADTs"

7.1.2.2 *Simulations with local state*

Although our work descends from research on logical relations, we would be remiss to not also mention the related work in the competing framework of (bi)simulations. Sumii and Pierce (2005) showed how to adapt simulation-style reasoning to existential types,[12] which Koutavas and Wand (2006) extended to an untyped language with general references.[13] These techniques were later generalized to "environmental bisimulations," in which a bisimulation between expressions is parameterized by the knowledge of the environment.[14] The use of the environment parameter in this last generalization often resembles the possible worlds technique on the logical relation side. See Dreyer *et al.* (2012) for a more detailed comparison.

Simulations, of course, also have a long history in reasoning about concurrency. However, as we said above, our model is the first to handle higher-order languages, shared-state concurrency, *and* local state. We will discuss the most closely-related simulation technique, RGSim, below.

[12] Sumii and Pierce (2005), "A bisimulation for type abstraction and recursion"

[13] Koutavas and Wand (2006), "Small bisimulations for reasoning about higher-order imperative programs"

[14] Sangiorgi *et al.* (2007), "Environmental Bisimulations for Higher-Order Languages"

7.1.3 *Shared-state concurrency*

Birkedal *et al.* recently developed the first logical-relations model for a higher-order concurrent language similar to the one we consider here.[15] Their aim was to show the soundness of a sophisticated Lucassen-and-Gifford-style[16] type-and-effect system, and in particular to prove the soundness of

[15] Birkedal *et al.* (2012), "A concurrent logical relation"

[16] Lucassen and Gifford (1988), "Polymorphic effect systems"

a Parallelization Theorem for disjoint concurrency expressed by the effect system (when the Bernstein conditions are satisfied). The worlds used in the logical relation capture the all-or-nothing approach to interference implied by the type-and-effect system. As a result, the model has rather limited support for reasoning about fine-grained data structures: it can only prove correctness of algorithms that can withstand arbitrary interference.

7.2 DIRECT REFINEMENT PROOFS

7.2.1 *Linearizability*

Herlihy and Wing's seminal notion of *linearizability*[17] has long been the gold standard of correctness for scalable concurrency, but as Filipović *et al.* argue,[18] what clients *really* want is a contextual refinement property. Filipović *et al.* go on to show that, under certain (strong) assumptions about a programming language, linearizability implies contextual refinement for that language.[19]

More recently, Gotsman and Yang generalized both linearizability and this result (the so-called *abstraction theorem*) to include potential ownership transfer of memory between data structures and their clients.[20] While it is possible to compose this abstraction theorem with a proof of linearizability to prove refinement, there are several advantages to our approach of proving refinement directly:

- First and foremost, it is a simpler approach: there is no need to take a detour through linearizability, or perform the (nontrivial!) proof that linearizability implies refinement. As it turns out, linearizability is neither the right proof technique (one would rather use something like protocols and thread-local reasoning) nor the right specification (clients really want refinement).
- We can treat refinement as an assertion in our logic, which means that we can *compose* proofs of refinement when reasoning about compound data structures, *and do so while working within a single logic*.
- Working with refinement makes it easier to leverage recent work for reasoning about hidden state, *e.g.,* Dreyer *et al.*'s STS-based logical relations Dreyer, Neis, and Birkedal 2010.
- Refinement seamlessly scales to the higher-order case, which would otherwise require extending the definition of linearizability to the higher-order case. We believe that this scalability is crucial for faithfully reasoning about algorithms that use higher-order features, *e.g.,* Herlihy's universal construction[21] or the recently proposed *flat combining* construction.[22]
- Finally, it should in principle allow us to combine reasoning about fine-grained concurrency with other kinds of relational reasoning, *e.g.,* relational parametricity.[23]

[17] Herlihy and Wing (1990), "Linearizability: a correctness condition for concurrent objects"

[18] Filipović *et al.* (2010), "Abstraction for Concurrent Objects"

[19] Under certain additional conditions, linearizability is also *complete* for contextual refinement.

[20] Gotsman and Yang (2012), "Linearizability with Ownership Transfer"

[21] Herlihy and Shavit (2008), "The Art of Multiprocessor Programming"

[22] Hendler *et al.* (2010), "Flat combining and the synchronization-parallelism tradeoff"

[23] Reynolds (1983), "Types, abstraction and parametric polymorphism"

7.2.2 *Denotational techniques*

Turon and Wand developed the first logic for reasoning directly about contextual refinement for scalable concurrent data structures.[24] Their model is based on ideas from rely-guarantee and separation logic and was developed for a simple first-order language, using an extension of Brookes's trace-based denotational semantics.[25] While it is capable of proving refinement for simple data structures like Treiber's stack, it does not easily scale to more sophisticated algorithms with hard-to-pinpoint linearization points (*e.g.,* those involving cooperation or temporally-dependent linearization).

[24] Turon and Wand (2011), "A separation logic for refining concurrent objects"

[25] Brookes (1996), "Full Abstraction for a Shared-Variable Parallel Language"

7.2.3 *RGSim*

More recently, Liang *et al.* proposed RGSim,[26] an inter-language simulation relation for verifying program transformations in a concurrent setting. The simulation relation is designed for compositional, concurrent reasoning: it is parameterized by rely and guarantee relations characterizing potential interference. Liang *et al.* use their method to prove that some simple, but realistic, data structures are simulated by their spec. While the original paper on RGSim did not relate simulation to refinement or linearizability, new (currently unpublished) work has done so.[27] We discuss this latter work, which also incorporates reasoning about cooperation, in §7.5.

[26] Liang *et al.* (2012), "A rely-guarantee-based simulation for verifying concurrent program transformations"

[27] Liang and Feng (2013), "Modular Verification of Linearizability with Non-Fixed Linearization Points"

7.3 LOCAL PROTOCOLS

7.3.1 *The hindsight approach*

O'Hearn *et al.*'s work on *Linearizability with hindsight*[28] clearly articulates the need for local protocols in reasoning about scalable concurrency, and demonstrates how a certain mixture of local and global constraints leads to insightful proofs about lock-free traversals. At the heart of the work is the remarkable *Hindsight Lemma*, which justifies conclusions about reachability in the past based on information in the present. Since O'Hearn *et al.* are focused on providing proofs for a particular class of algorithms, they do not formalize a general notion of protocol, but instead focus on a collection of invariants specific to the traversals they study. We have focused, in contrast, on giving a simple but general account of local protocols that suffices for *temporally-local* reasoning about a range of quite different data structures. It remains to be seen, however, whether our techniques yield a satisfying temporally-local correctness proof for the kinds of traversals O'Hearn *et al.* study, or whether (as O'Hearn *et al.* argue) these traversals are best understood non-locally.

[28] O'Hearn *et al.* (2010), "Verifying linearizability with hindsight"

7.3.2 *Concurrent abstract predicates*

The notion of protocol most closely related to ours is Dinsdale-Young *et al.*'s *Concurrent abstract predicates* (CAP).[29] CAP extends separation logic with shared, hidden regions similar to our islands. These regions are governed by a set of *abstract predicates*,[30] which can be used to make localized assertions about the state of the region. In addition, CAP provides a notion of named *actions* which characterize the possible changes to the region. Crucially, actions are treated as a kind of *resource* which can be gained, lost, or split up (in a fractional permissions style), and executing an action can result in a change to the available actions. It is incumbent upon users of the logic to show that their abstract predicates and actions cohere, by showing that every abstract predicate is "self-stable" (remains true after any available action is executed).

While CAP's notion of protocol is very expressive, it is also somewhat "low-level" compared to our STS-based protocols, which would require a somewhat unwieldy encoding to express in CAP. In addition, our protocols make a clear separation between *knowledge* bounding the state of the protocol (treated as a copyable assertion) and *rights* to change the state (treated as a linear resource: tokens), which are mixed in CAP. Another major difference is that CAP exposes the internal protocol of a data structure as part of the specification seen by a client—which means that the spec for a given data structure often depends on how the client is envisioned to use it. Additional specs (and additional correctness proofs) may be necessary for other clients. By contrast, we take a coarse-grained data structure as an all-purpose spec; if clients then want to use that data structure according to some sophisticated internal protocol, they are free to do so. Finally, our protocols support speculation and spec code as a resource, neither of which are supported by CAP.

Very recent work has sought to overcome some of the shortcomings of CAP by, in part, moving to a higher-order *logic*.[31] The key idea is to avoid overspecialization in specifications by *quantifying* over the pre- and post-conditions a client might want to use. Through a clever use of ghost state and fractional permissions, these client-side assertions are linked to the abstract state of the data structure being verified, and can thus track atomic changes to that data structure. The downside is that the model theory supporting this higher-order extension is quite tricky, and at present requires restrictions on instantiation to rule out certain kinds of self-reference. In addition, because the approach is not known to be sound for refinement, clients can make use of HOCAP specifications only if they also work within the HOCAP logic. With refinement, by contrast, the specification is given in a *logic-independent* way, *i.e.*, solely in terms of the operational semantics of a language, which means that clients are free to use any logic of their choice when reasoning about their code. Finally, it is as yet unclear whether the HOCAP approach

[29] Dinsdale-Young *et al.* (2010), "Concurrent Abstract Predicates"

[30] First introduced in Parkinson and Bierman (2005).

[31] Svendsen *et al.* (2013), "Modular Reasoning about Separation of Concurrent Data Structures"

can scale to handle cooperation and changes to the branching structure of nondeterminism.

7.3.3 *Views and other fictions of separation*

In the last year, there has been an explosion of interest in a new, highly abstract way to express the knowledge and rights of program components: through a (partial, commutative) monoid. The idea is that each element of the monoid captures a "piece" of abstract knowledge that a component (say, a thread) might have—and this piece of knowledge constrains changes that other components can make, since the knowledge *is not allowed to be violated*.[32] Formally, this is expressed by giving a concrete interpretation $\lfloor - \rfloor$ to the "global supply" of monoid elements (the product of each component's knowledge), and then insisting that all actions obey an abstract frame condition. Namely: if a command C claims to go from local knowledge m to m', it must satisfy

$$\forall m_F. [\![C]\!](\lfloor m \cdot m_F \rfloor) \subseteq \lfloor m' \cdot m_F \rfloor$$

That is, the new global state of the monoid $m' \cdot m_F$ must still contain the frame m_F from the original global state $m \cdot m_F$. A command cannot invalidate the abstract knowledge of its environment.

[32] Jensen and Birkedal (2012), "Fictional Separation Logic"

The result is *fictional separation logic*:[33] using monoid elements as assertions, we get an abstract notion of separation (via the monoid product) that may be fictional in that it does *not* coincide with physical separation. Put differently, the map $\lfloor - \rfloor$ that gives a physical interpretation to the global abstract state need not be a homomorphism, so in general

$$\lfloor m \cdot m' \rfloor \quad \neq \quad \lfloor m \rfloor * \lfloor m' \rfloor$$

[33] Jensen and Birkedal (2012), "Fictional Separation Logic"

As a simple example, a monotonic counter can be represented using a monoid of natural numbers with *max* as the product; if the counter is at location ℓ then the interpretation is just

$$\lfloor n \rfloor = \ell \mapsto n$$

Notice that

$$\lfloor n \cdot m \rfloor \quad = \quad \ell \mapsto \max(n, m) \quad \neq \quad \ell \mapsto n * \ell \mapsto m \quad = \quad \lfloor n \rfloor * \lfloor m \rfloor$$

By asserting the monoid element n, a component claims that the value of the counter is *at least* n; after all, the other components will contain some additional knowledge, say n_F, but $n \cdot n_F = \max(n, n_F) \geq n$. Similarly, the frame condition will ensure that the physical value of the counter monotonically increases.

While fictional separation logic began in a sequential setting, it has already been adapted to concurrent settings, both to give a compositional account of ghost state[34] and to provide an abstract framework ("Views") for concurrent program logics and type systems, with a single soundness proof that can be instantiated with arbitrary choices of monoids.[35] The Views framework has

[34] Ley-Wild and Nanevski (2013), "Subjective Auxiliary State for Coarse-Grained Concurrency"

[35] Dinsdale-Young *et al.* (2013), "Views: Compositional Reasoning for Concurrent Programs"

even been instantiated with CAP, which means that CAP's notion of protocol can be understood as a particular choice of monoid.

In joint work with Krishnaswami, Dreyer and Garg, we showed that the central ideas of fictional separation logic can be applied in the setting of a logical relation for a sequential language, where we use linguistic hiding mechanisms to introduce new monoids.[36] The logical relation uses a possible-worlds model in which each island consists of a different monoid equipped with an interpretation (like $\lfloor - \rfloor$ above). We thus expect that we could redo the work of Chapter 5 using monoids instead of STSs, by encoding our token-based STSs as monoids. It is an open question whether the converse is true—*i.e.,* whether STSs with tokens can express arbitrary monoids with frame conditions. In any case, for all of the algorithms we have examined, expressing the relevant protocol as an STS with tokens is invariably simpler and more intuitive than doing so with monoids, which is what led us to stick with our STS-based worlds. Other aspects of our model—direct refinement proofs, high-level languages, cooperation and speculation—have not yet been incorporated into the Views approach.

[36] Krishnaswami *et al.* (2012), "Superficially substructural types"

7.4 ROLE-PLAYING

The classic treatment of role-playing in shared-state concurrency is Jones's *rely-guarantee* reasoning,[37] in which threads *guarantee* to make only certain updates, so long as they can *rely* on their environment to make only certain (possibly different) updates. More recent work has combined rely-guarantee and separation logic (SAGL and RGSep[38]), in some cases even supporting a frame rule over the rely and guarantee constraints themselves (LRG[39]). This line of work culminated in Dodds *et al.*'s *deny-guarantee* reasoning[40]—the precursor to CAP—which was designed to facilitate a more dynamic form of rely-guarantee to account for non-well-bracketed thread lifetimes. In the deny-guarantee framework, actions are classified into those that both a thread and its environment can perform, those that neither can perform, and those that only one or the other can perform. The classification of an action is manifested in terms of two-dimensional fractional permissions (the dimensions being "deny" and "guarantee"), which can be split and combined dynamically. Our STSs express dynamic evolution of roles in an arguably more direct and visual way, through tokens.

[37] Jones (1983), "Tentative steps toward a development method for interfering programs"

[38] Feng *et al.* (2007); Vafeiadis and Parkinson (2007)

[39] Feng (2009), "Local rely-guarantee reasoning"

[40] Dodds *et al.* (2009), "Deny-guarantee reasoning"

7.5 COOPERATION

7.5.1 *RGSep*

Vafeiadis's thesis[41] set a high-water mark in verification of the most sophisticated concurrent data structures (such as CCAS). Building on his RGSep logic, Vafeiadis established an informal methodology for proving linearizabil-

[41] Vafeiadis (2008), "Modular fine-grained concurrency verification"

ity by employing several kinds of ghost state (including prophecy variables and "one-shot" resources, the latter representing linearization points). By cleverly storing and communicating this ghost state to another thread, one can perform thread-local verification and yet account for cooperation: the other thread "fires" the single shot of the one-shot ghost resource. While this account of cooperation seems intuitively reasonable, it lacks any formal metatheory justifying its use in linearizability or refinement proofs. Our computational resources generalize Vafeiadis's "one-shot" ghost state, since they can (and do) run computations for an arbitrary number of steps, and we have justified their use in refinement proofs—showing, in fact, that the technique of logical relations can be expressed in a "unary" (Hoare logic) style by using these computational resources.

7.5.2 *RGSim*

Concurrently with our work, Liang and Feng have extended their RGSim framework to account for cooperation.[42] The new simulation method is parameterized by a "pending thread inference map" Θ, which plays a role somewhat akin to our worlds. For us, worlds impose a relation between the current protocol state, the current implementation heap, and the current, speculative spec resources. By contrast, Θ imposes a relation between the current implementation heap and the current spec thread pool. To recover something like our protocols, one instead introduces *ghost state* into the implementation heap, much as Vafeiadis does; as a result, Θ can be used to do thread-local reasoning about cooperation. However, there are several important differences from our approach:

- There is no notion of *composition* on thread inference maps, which take the perspective of the global implementation heap and global pool of spec threads. Thus thread inference maps do not work as resources that can be owned, split up, transferred and recombined.
- The assertions that are used in pre- and post-conditions cannot talk directly about the thread inference map; they must control it indirectly, via ghost state.
- The simulation approach does not support speculation or high-level language features like higher-order functions or polymorphism.
- Finally, it requires encoding protocols via traditional ghost state and rely/guarantee, rather than through standalone, visual protocols.

[42] Liang and Feng (2013), "Modular Verification of Linearizability with Non-Fixed Linearization Points"

7.5.3 *Reduction techniques*

Groves and Colvin propose[43] a radically different approach for dealing with cooperation, based on Lipton's method[44] of *reduction*. Reduction, in a sense, "undoes" the effects of concurrency by showing that interleaved actions

[43] Groves and Colvin (2009), "Trace-based derivation of a scalable lock-free stack algorithm"

[44] Lipton (1975), "Reduction: a method of proving properties of parallel programs"

commute with one another: if a thread performs action a and then b and a is a "right-mover" ($a; c \sqsubseteq c; a$ for all environment actions c) then we can instead imagine the thread executes $\langle a; b \rangle$, *i.e.,* executes a and b together in one atomic step. Groves and Colvin are able to *derive* an elimination stack from its spec by a series of transformations including atomicity refinement and data refinement. The key to handling cooperation is working not just with individual actions, but with *traces*, so that a given refinement step can map a trace with a single action by one thread (say, accepting an offer to push) to a trace with two contiguous steps (say, a push and a pop) attributed to two different threads. Elmas *et al.* also developed a similar method[45] for proving linearizability using reduction and *abstraction* (the converse to refinement) and while they do not study cooperation explicitly, it is likely that their method could be adapted to cope with it too, if it was likewise reformulated using traces.

[45] Elmas *et al.* (2010), "Simplifying Linearizability Proofs with Reduction and Abstraction"

Groves and Colvin's approach is somewhat like reasoning directly about linearizability, since it is focused on proving the reorderability of steps within a trace with the aim of producing a sequential interleaving in the end. The downside is that the approach offers none of the kinds of locality we have emphasized:

- It lacks temporal locality because it is based on traces recording a complete method execution interleaved with arbitrary action sequences performed by other threads.
- It lacks thread locality because interleaved actions are drawn directly from the *code* of other executing threads, rather than an abstraction of their possible interference (say, a rely constraint or a protocol). This point is somewhat mitigated by the use of *abstraction*, especially for Elmas *et al.*'s calculus of atomic actions,[46] which allows code to be abstracted while delaying the proof that the abstraction is valid. It is unclear, however, how these abstraction techniques compare to rely/guarantee reasoning or local protocols.
- It lacks spatial locality in that the commutativity checks require considering "interference" from environment code even when that code is accessing a completely different part of the heap. Granted, such circumstances make commutativity easy to show, but with a spacially local account of interference the checks are unnecessary in the first place.

[46] Elmas *et al.* (2009), "A calculus of atomic actions"

Finally, in a reduction-based proof there is rarely an articulation of the protocol governing shared state. We believe that such an artifact is valuable in its own right as a way of understanding the basic mechanics of an algorithm separately from its implementation.

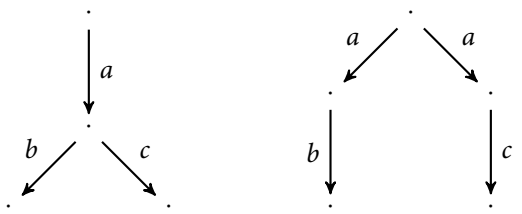
7.6 NONDETERMINISM

7.6.1 *The linear time/branching time spectrum*

Reasoning about time is a fraught issue, because in the presence of nondeterminism there is not necessarily a fixed notion of "the future." There is, in fact, a whole *spectrum* of possible ways to understand nondeterminism and temporal reasoning, elegantly summarized in Glabbeek (1990). The two extreme points in the spectrum are linear time and branching time:

- IN THE LINEAR-TIME VIEW, all nondeterminism is resolved in one shot, at the beginning of program execution—so after a program begins running, there *is* a coherent notion of "the future." This view of time works well with a trace semantics: by taking the meaning of a program to be a set of traces, one commits to the view that *all* nondeterminism is resolved through one choice, namely, the choice of trace. From the perspective of a particular point in a particular trace, "the future" is just the remaining suffix of the trace.

- IN THE BRANCHING-TIME VIEW, one distinguishes between a program that makes a nondeterministic choice *now* versus one that makes the choice *later*. These distinctions often go hand-in-hand with a distinction between *internal* nondeterminism (over which the environment has no control) or *external* nondeterminism (usually called *external choice*) in which the environment has some say. Thus, for example, the following two "programs" are distinguished according to a branching-time model:



The program on the left first interacts with the environment along channel a, and is then willing communicate along either b or c—whichever the environment chooses.[47] On the other hand, the program on the right communicates along a, but also makes an *internal* choice about whether to next attempt communication on b or on c. *These two programs have the same set of traces*, at least in a naive trace semantics. Semantically, the branching-time view is usually associated with (bi)simulation rather than traces; simulation requires that choices which are available at any point in an implementation's execution are also still available in a corresponding spec execution.

[47] In terms of reagents or CML events, this is just a choice between communication along two channels.

There are strong arguments that the branching-time view is too fine-grained: in many settings, it is *not possible* to write a program context which can

observe such differences.[48] This is typically true when, for example, defining contextual equivalence (or refinement) in terms of observing the final value produced, as we do. This is the essential reason why speculation—which destroys sensitivity to branching structure—is valid in our model: the definition of contextual refinement for F^{μ}_{cas} is similarly branching-insensitive, in part because F^{μ}_{cas} has no notion of external choice.

[48] See Bloom *et al.* (1995) and Nain and Vardi (2007).

7.6.2 *Forward, backward, and hybrid simulation*

Forward simulation (the usual kind of simulation) is well-known to be sensitive to branching, which in many cases means it distinguishes too many programs. On the other hand, forward simulation is appealingly *local*, since it considers only one step of a program at a time (as opposed to *e.g.*, trace semantics). To retain temporally-local reasoning but permit differences in nondeterminism (as in the late/early choice example), it suffices to use a combination of forward and *backward* simulation[49] or, equivalently, history and *prophecy* variables.[50] Lynch and Vaandrager showed that there are also *hybrids* of forward and backward simulations, which relate a *single* state in one system to a *set* of states in the other—much like our speculation. In fact, although it was not stated explicitly, Herlihy and Wing's original paper on linearizability proposed using something like hybrid simulation to deal with indeterminite linearization points.[51]

[49] Lynch and Vaandrager (1995), "Forward and Backward Simulations: Part I: Untimed Systems"

[50] Abadi and Lamport (1991), "The existence of refinement mappings"

[51] Herlihy and Wing (1990), "Linearizability: a correctness condition for concurrent objects"

Our technique goes further, though, in combining this *temporally*-local form of reasoning with *thread*-local reasoning: hybrid simulations work at the level of complete systems, whereas our threadpool simulations can be composed into larger threadpool simulations. Composability allows us to combine thread-private uses of speculation with shared uses of speculation in protocols, which is critical for proving soundness with respect to contextual refinement.

Part III

EXPRESSING SCALABLE CONCURRENCY

8

Join patterns

▸ Synopsis This chapter introduces join patterns and Russo (2007)'s joins API for C♯. It shows how join patterns can solve a wide range of synchronization problems, including many of the problems solved by JUC's primitives.[1] The full API is given in Appendix D.

"Programming is the art of writing essays in crystal clear prose and making them executable."

—Per Brinch Hansen

[1] Our versions lack some features of the real library, such as timeouts and cancellation, but these should be straightforward to add (§9.6).

8.1 Overview

The subtlety of scalable concurrent algorithms should, by now, be clear. In practice design and implementation is generally left up to the experts, who build extensive libraries of scalable primitives for application programmers to use. Inevitably, though, programmers are faced with new problems not directly addressed by the primitives. The primitives must then be composed into a solution—and doing so correctly and scalably can be as difficult as designing a new primitive.

Take the classic Dining Philosophers problem,[2] in which philosophers sitting around a table must coordinate their use of the chopstick sitting between each one; such competition over limited resources appears in many guises in real systems. The problem has been thoroughly studied, and there are solutions using primitives like semaphores that perform reasonably well. There are also many natural "solutions" that do not perform well—or do not perform *at all*. Naive solutions may suffer from deadlock, if for example each philosopher picks up the chopstick to their left, and then finds the one to their right taken. Correct solutions built on top of library primitives may still scale poorly with the number of philosophers (threads). For example, using a single global lock to coordinate philosophers is correct, but will force non-adjacent philosophers to take turns through the lock, adding unnecessary sequentialization. Avoiding these pitfalls takes experience and care.

[2] Dijkstra (1971), "Hierarchical ordering of sequential processes"

In this chapter, we demonstrate that Fournet and Gonthier's *join calculus*[3] can provide the basis for a declarative and scalable synchronization library. By *declarative*, we mean that the programmer needs only to write down the constraints for a synchronization problem,[4] and the library will automatically derive a correct solution. By *scalable*, we mean that the derived solutions deliver robust, competitive performance both as the number of processors or cores increase, and as the complexity of the synchronization problem grows.

[3] Fournet and Gonthier (1996), "The reflexive CHAM and the join-calculus"

[4] We will not attempt for formalize the precise class of "synchronizations" that can be solved using join patterns, but the examples in this chapter give some indication of the range of sharing (§2.2.2) and timing (§2.2.3) problems that can be declaratively addressed.

Figure 8.1 shows a solution to Dining Philosophers using our library, which is a drop-in replacement for Russo's C♯ Joins library.[5] The library is based on

[5] Russo (2007), "The Joins Concurrency Library"

```
var j = Join.Create();
Synchronous.Channel[]  hungry;    j.Init(out hungry, n);
Asynchronous.Channel[] chopstick; j.Init(out chopstick, n);

for (int i = 0; i < n; i++) {
   var left  = chopstick[i];
   var right = chopstick[(i+1) % n];

   // define one join pattern per philosopher
   j.When(hungry[i]).And(left).And(right).Do(() => {
      eat();
      left(); right(); // return the chopsticks
   });
}
```

Figure 8.1: Dining Philosophers, declaratively

the message-passing primitives of the join calculus. For Dining Philosophers, we use two *arrays* of channels (`hungry` and `chopstick`) to carry value-less messages; being empty, these messages represent unadorned events. The declarative aspect of this example is the *join pattern* starting with `j.When`. The declaration says that when events are available on the channels `hungry[i]`, `left`, and `right`, they may be *simultaneously and atomically* consumed. When the pattern fires, the philosopher, having obtained exclusive access to two chopsticks, eats and then returns the chopsticks. In neither the join pattern nor its body is the order of the chopsticks important. The remaining details of Russo's API are explained in §8.2.

Most implementations of join patterns, including Russo's, use coarse-grained locks to achieve atomicity, resulting in poor scalability (as we show experimentally in §9.8). Our contribution is a new implementation of the join calculus that uses ideas from fine-grained concurrency to achieve scalability on par with custom-built synchronization primitives. We present that implementation in Chapter 9.

This brief chapter provides a review of the join calculus, and of Russo's library API in particular. We recall how join patterns can be used to solve a wide range of coordination problems (§8.2), as is well-established in the literature.[6] The examples provide basic implementations of some of the JUC primitives mentioned in the introduction (Chapter 1). In each case, the Joins-based solution is as straightforward to write as the one for dining philosophers.

[6] Fournet and Gonthier 1996; Fournet and Gonthier 2002; Benton *et al.* 2004

8.2 THE JOIN CALCULUS AND RUSSO'S API

The join calculus takes a message-passing approach to concurrency where messages are sent over channels and channels are themselves first-class values that can be sent as messages. What makes the calculus interesting is the way messages are received. Programs do not actively request to receive messages from a channel. Instead, they employ *join patterns* (also called *chords*[7]) to declaratively specify reactions to message arrivals. The power of join patterns lies in their ability to respond *atomically* to messages arriving simultaneously on several different channels.

[7] Benton *et al.* (2004), "Modern concurrency abstractions for C#"

Suppose, for example, that we have two channels `Put` and `Get`, used by producers and consumers of data. When a producer and a consumer message are available, we would like to receive both simultaneously, and transfer the produced value to the consumer. Using Russo's API, we write:

```
class Buffer<T> {
  public readonly Asynchronous.Channel<T> Put; // T = message type
  public readonly Synchronous<T>.Channel Get;  // T = reply type
  public Buffer() {
    Join j = Join.Create();    // allocate a Join object
    j.Init(out Put);           // bind its channels
    j.Init(out Get);
    j.When(Get).And(Put).Do    // register chord
      (t ⇒ { return t; });
  }
}
```

This simple example introduces several aspects of the API.

First, there are multiple kinds of channels: `Put` is an asynchronous channel that carries messages of type `T`, while `Get` is a synchronous channel that yields replies of type `T` but takes no argument. A sender never blocks on an asynchronous channel, even if the message cannot immediately be received through a join pattern. For the `Buffer` class, that means that a single producer may send many `Put` messages, even if none of them are immediately consumed. Because `Get` is a synchronous channel, on the other hand, senders will block until or unless a pattern involving it is enabled. Synchronous channels also return a reply to message senders; the reply is given as the return value of join patterns.

Join patterns are declared using the `When` method. The single join pattern in `Buffer` stipulates that when one `Get` request and one `Put` message are available, they should both be consumed. After specifying the involved channels through `When` and `And`, the `Do` method is used to give the *body* of the join pattern. The body is a piece of code to be executed whenever the pattern is matched and relevant messages consumed. It is given as a delegate[8] taking as arguments the contents of the messages. In `Buffer`, the two channels `Get` and

[8] C♯'s first-class functions

`Put` yield only one argument, because `Get` messages take no argument. The body of the pattern simply returns the argument `t` (from `Put`), which then becomes the reply to the `Get` message. Altogether, each time the pattern is matched, one `Get` and one `Put` message are consumed, and the argument is transferred from `Put` to the sender of `Get` as a reply.

Channels are represented as delegates, so that messages are sent by simply invoking a channel as a function. From a client's point of view, `Put` and `Get` look just like methods of `Buffer`. If `buf` is an an instance of `Buffer`, a producer thread can post a value `t` by calling `buf.Put(t)`, and a consumer thread can request a value by calling `buf.Get()`.

Finally, each channel must be associated with an instance of the `Join` class.[9] Such instances are created using the static factory method `Join.Create`, which optionally takes the maximum number of required channels. Channels are bound using the `Init` method of the `Join` class, which initializes them using an out-parameter. These details are not important for the overall design, and are elided from subsequent examples. The full API—including the determination of types for join pattern bodies—is given in Appendix D.

[9] This requirement retains compatibility with Russo's original Joins library; we also use it for the stack allocation optimization described in §9.6.

8.3 SOLVING SYNCHRONIZATION PROBLEMS WITH JOINS

As we have seen, when a single pattern mentions several channels, it forces synchronization:

```
Asynchronous.Channel<A> Fst;
Asynchronous.Channel<B> Snd;
Synchronous<Pair<A,B>>.Channel Both;
// create j and init channels (elided)
j.When(Both).And(Fst).And(Snd).Do((a,b) ⇒ new Pair<A,B>(a,b));
```

The above pattern will consume messages `Fst(a)`, `Snd(b)` and `Both()` atomically, when all three are available. Only the first two messages carry arguments, so the body of the pattern takes two arguments. Its return value, a pair, becomes the return value of the call to `Both()`.

On the other hand, several patterns may mention the same channel, expressing choice:

```
Asynchronous.Channel<A> Fst;
Asynchronous.Channel<B> Snd;
Synchronous<Sum<A,B>>.Channel Either;
// create j and init channels (elided)
j.When(Either).And(Fst).Do(a ⇒ new Left<A,B>(a));
j.When(Either).And(Snd).Do(b ⇒ new Right<A,B>(b));
```

Each pattern can fulfill a request on `Either()`, by consuming a message `Fst(a)` or a message `Snd(b)`, and wrapping the value in a variant of a disjoint sum.

Using what we have seen, we can build a simple (non-recursive) `Lock`:

```
class Lock {
  public readonly Synchronous.Channel Acquire;
  public readonly Asynchronous.Channel Release;
  public Lock() {
    // create j and init channels (elided)
    j.When(Acquire).And(Release).Do(() ⇒ { });
    Release();  // initially free
  }
}
```

As in the dining philosophers example, we use void-argument, void-returning channels as *signals*. The `Release` messages are tokens that indicate that the lock is free to be acquired; it is initially free. Clients must follow the protocol of calling `Acquire()` followed by `Release()` to obtain and relinquish the lock. Protocol violations will not be detected by this simple implementation. However, when clients follow the protocol, the code will maintain the invariant that at most one `Release()` token is pending on the queues and thus at most one client can acquire the lock.

With a slight generalization, we obtain a *semaphore*:[10]

[10] Dijkstra (1965), "EWD123: Cooperating Sequential Processes"

```
class Semaphore {
  public readonly Synchronous.Channel Acquire;
  public readonly Asynchronous.Channel Release;
  public Semaphore(int n) {
    // create j and init channels (elided)
    j.When(Acquire).And(Release).Do(() ⇒ { });
    for (; n > 0; n--) Release(); // initially n free
  }
}
```

A semaphore allows at most n clients to `Acquire` the resource and proceed; further acquisitions must wait until another client calls `Release()`. We arrange this by priming the basic `Lock` implementation with n initial `Release()` tokens.

We can also generalize `Buffer` to a synchronous channel that exchanges data between threads:

```
class Exchanger<A, B> {
  readonly Synchronous<Pair<A, B>>.Channel<A> left;
  readonly Synchronous<Pair<A, B>>.Channel<B> right;
  public B Left(A a)  { return left(a).Snd; }
  public A Right(B b) { return right(b).Fst; }
  public Exchanger() {
    // create j and init channels (elided)
    j.When(left).And(right).Do((a,b) ⇒ new Pair<A,B>(a,b));
  }
}
```

Dropping message values, we can also implement an n-way *barrier* that causes n threads to wait until all have arrived:

```
class SymmetricBarrier {
  public readonly Synchronous.Channel Arrive;
  public SymmetricBarrier(int n) {
    // create j and init channels (elided)
    var pat = j.When(Arrive);
    for (int i = 1; i < n; i++) pat = pat.And(Arrive);
    pat.Do(() ⇒ { });
  }
}
```

This example is unusual in that its sole join pattern mentions a single channel `n` times: the pattern is *nonlinear*. This repetition means that the pattern will not be enabled until the requisite n threads have arrived at the barrier, and our use of a single channel means that the threads need not distinguish themselves by invoking distinct channels (hence "symmetric"). On the other hand, if the coordination problem did call for separating threads into groups,[11] it is easy to do so. We can construct a barrier requiring n threads of one kind and m threads of another, simply by using two channels.

We can also implement a tree-structured variant of an asymmetric barrier, which breaks a single potentially large n-way coordination problem into $O(n)$ two-way problems. Such tree-structured barriers (or more generally, combiners) have been studied in the literature;[12] the point here is just that adding tree-structured coordination is straightforward using join patterns. As we show in §9.8, the tree-structured variant performs substantially better than the flat barrier, although both variants easily outperform the .NET `Barrier` class (a standard sense-reversing barrier).

[11] *e.g.*, "gender" is useful in a parallel genetic algorithm (William N. Scherer, III *et al.* 2005)

[12] See Herlihy and Shavit 2008 for a survey

```
class TreeBarrier {
  public readonly Synchronous.Channel[] Arrive;
  private readonly Join j; // create j, init chans ...
  public TreeBarrier(int n) {Wire(0, n-1, () ⇒ {});}
  private void Wire(int low, int high, Action Done) {
    if (low == high) {
      j.When(Arrive[low]).Do(Done);
    } else if (low + 1 == high) {
      j.When(Arrive[low]).And(Arrive[high]).Do(Done);
    else { // low + 1 < high
      Synchronous.Channel Left, Right; // init chans
      j.When(Left).And(Right).Do(Done);
      int mid = (low + high) / 2;
      Wire(low, mid, () ⇒ Left());
      Wire(mid + 1, high, () ⇒ Right());
    }
  }
}
```

Finally, we can implement a simple reader-writer lock, using private asynchronous channels (`idle` and `shared`) to track the internal state of a synchronization primitive:[13]

[13] Benton *et al.* (2004), "Modern concurrency abstractions for C#"

```
class ReaderWriterLock {
  private readonly Asynchronous.Channel idle;
  private readonly Asynchronous.Channel<int> shared;
  public readonly Synchronous.Channel AcqR, AcqW,
                                      RelR, RelW;
  public ReaderWriterLock() {
    // create j and init channels (elided)
    j.When(AcqR).And(idle).Do(() ⇒ shared(1));
    j.When(AcqR).And(shared).Do(n ⇒ shared(n+1));
    j.When(RelR).And(shared).Do(n ⇒ {
      if (n == 1) idle(); else shared(n-1);
    });
    j.When(AcqW).And(idle).Do(() ⇒ { });
    j.When(RelW).Do(() ⇒ idle());
    idle(); // initially free
  }
}
```

While we have focused on the simplest synchronization primitives as a way of illustrating Joins, join patterns can be used to declaratively implement more complex concurrency patterns, from Larus and Parks-style *cohort-scheduling*,[14] to Erlang-style *agents* or *active objects*,[15] to stencil computations

[14] Benton *et al.* 2004

[15] Benton *et al.* 2004

with systolic synchronization,[16] as well as classic synchronization puzzles.[17]

[16] Russo 2008
[17] Benton 2003

9

Implementing join patterns

▸ Synopsis This chapter walks through the implementation of scalable join patterns, including excerpts of the core C♯ library code (§9.3 and §9.4) and optimizations (§9.5). It validates our scalability claims experimentally on seven different coordination problems (§9.8). For each coordination problem we evaluate a joins-based implementation running in both Russo's lock-based library and our new scalable library, and compare these results to the performance of direct, custom-built solutions. In all cases, the new library scales significantly better than Russo's, and competitively with—sometimes better than—the custom-built solutions, though it suffers from higher constant-time overheads in some cases.

"While formerly it had been the task of the programs to instruct our machines, it had now become the task of the machines to execute our programs."

—Edsger W. Dijkstra, "EWD952: Science fiction and science reality in computing"

9.1 Overview

In the previous chapter we saw, through a range of examples, how the join calculus allows programmers to solve synchronization problems by merely writing down the relevant constraints. Now we turn to our contribution: an implementation that solves these constraints in a scalable way.

9.1.1 *The problem*

The chief challenge in implementing the join calculus is providing atomicity when firing patterns: messages must be noticed and withdrawn from multiple collections simultaneously. A simple way to ensure atomicity, of course, is to use a lock (§2.4), and this is what most implementations do (see Chapter 12).[1] For example, Russo's original library associates a single lock with each `Join` object. Each sender must acquire the lock and, while holding it, enqueue their message and determine whether any patterns are thereby enabled.

[1] Some implementations use STM (Shavit and Touitou 1995), which we also discuss in Chapter 12.

But Russo's library goes further, putting significant effort into shortening the critical section: it uses bitmasks summarizing message availability to accelerate pattern matching,[2] represents `void` asynchronous channels as counters, and permits "message stealing" to increase throughput—all the tricks from Benton *et al.* (2004).

[2] Le Fessant and Maranget (1998), "Compiling Join Patterns"

Unfortunately, even with relatively short critical sections, coarse-grained locking inevitably limits scalability (§2.4). The scalability problems with locks are a major obstacle to using the Joins library to *implement* custom, low-level synchronization primitives. In addition to the general memory traffic

problems caused by locks, coarse-grained locking for joins unnecessarily serializes the process of matching and firing chords: at most one thread can be performing that work at a time. In cases like the exchanger and Dining Philosophers, a much greater degree of concurrency is both possible and desirable.

In short, for joins to be viable as a user-extensible synchronization library, we need an implementation that matches and fires chords in parallel while minimizing costly interprocessor communication, *i.e.,* we need *scalable join patterns*.

9.1.2 *Our approach*

In order to permit highly-concurrent access to the collection of messages available on a given channel, we use lock-free bags to represent channels.[3] The result is that, for example, two threads can be simultaneously adding separate messages to the same channel bag, while a third examines a message already stored in the bag—without any of the threads waiting on any other, and in many cases without any memory bus traffic. In choosing a bag rather than, say, a queue, we sacrifice message ordering guarantees to achieve greater concurrency: FIFO ordering imposes a sequential bottleneck on queues. The original join calculus did not provide any ordering guarantees, and relaxed ordering is typical in implementations.[4] The choice of ordering is not, however, fundamental to our algorithm; ordered channels are easy to provide (§9.6). None of our examples rely on ordering.

Lock-free bags allow messages to be added and inspected concurrently, but they do not solve the central problem of *atomically consuming* a pattern's worth of messages. To provide atomic matching, we equip messages with a `Status` field of the following type:

```
enum Stat { PENDING, CLAIMED, CONSUMED };
```

Statuses are determined according to the following protocol:

- Each message is `PENDING` to begin with, meaning that it is available for matching and firing.
- Matching consists of finding sufficiently many `PENDING` messages, then using CAS to try to change them one by one to from `PENDING` to `CLAIMED`.
- If matching is successful, each message can be marked `CONSUMED`. If it is unsuccessful, each `CLAIMED` message is reverted to `PENDING`.

Messages marked `CONSUMED` are *logically* deleted, but need not be *physically* removed from the bag until a later, more convenient moment.

Using the techniques of Chapter 4, we can visualize the per-message protocol as follows as shown in Figure 9.1. The local protocol uses a single token, denoted as usual with "•", which represents ownership of the message.

[3] The bag implementation we used for our measurements is, unfortunately, a closed-source implementation based on Microsoft intellectual property, but it is loosely based on the Michael-Scott queue (Michael and Scott 1998) and thus does not take full advantage of the orderless bag semantics. Subsequent to our implementation, several lock-free bag implementations have appeared in the literature (Sundell *et al.* 2011; David Dice and Otenko 2011).

[4] Fournet and Gonthier 1996; Russo 2007; Benton *et al.* 2004

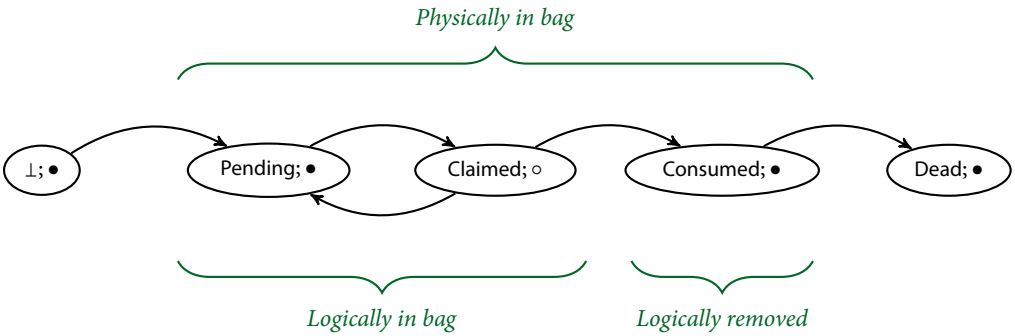


Figure 9.1: Per-message protocol

Part of this protocol should look familiar: the token-controlled loop between the Pending and Claimed states is isomorphic to the simple lock protocol in §4.3. Indeed, the status field does act as a kind of fine-grained lock, one tied to individual messages rather than an entire instance of `Join`. But if we fail to "acquire" a message, we do not immediately spinwait or block. Instead, we can continue looking through the relevant bag of messages for another message to claim—or, more generally, for another join pattern to match (§9.3).

There are three reasons the above is just an "overview" and not the full algorithm:

- Knowing when to terminate the protocol with the result of "no pattern matched" turns out to be rather subtle: because the message bags are not locked, new potential matches can occur at any time. Terminating the protocol is important for returning control to an asynchronous sender, or deciding to block a synchronous sender. But terminating too early can result in dropped (undetected, but enabled) matches, which can lead to deadlock. The full algorithm, including its termination condition, is given in §9.3.

- Patterns involving synchronous channels add further complexity: if an asynchronous message causes such a pattern to be fired, it must alert a synchronous waiter, which must in turn execute the pattern body. Likewise, if there are multiple synchronous senders in a given pattern, they must be coordinated so that only one executes the body and communicates the results to the others. We cover these details in §9.4.

- Two "optimizations" of the protocol turn out to be crucial for achieving scalability: lazy queueing and message stealing. The details of these optimizations are spelled out in §9.5, while their ramifications on scalability are examined empirically in §9.8.

9.2 REPRESENTATION

Before delving into the C♯ implementation of scalable join patterns, we briefly survey the interfaces to the key data structures it uses; see Figure 9.2.

```
// Msg implements:
  Chan Chan         { get; };
  Stat Status       { get; set; };
  bool TryClaim();  // CAS from PENDING to CLAIMED
  Signal Signal     { get; };
  Match ShouldFire  { get; set; };
  object Result     { get; set; };
```

```
// Chan<A> implements:
  Chord[] Chords    { get; };
  bool IsSync       { get; };
  Msg AddPending(A a);
  Msg FindPending(out bool sawClaims);
```

```
// Match implements:
  Chord Chord       { get; };
  Msg[] Claims      { get; };
```

```
// Chord implements:
  Chan[] Chans;     { get; };
  Msg[] TryClaim(Msg msg, ref bool retry);
```

Figure 9.2: Interfaces to the key data structures

Messages are represented as instances of the `Msg` class, which, in addition to carrying the message payload, includes a `Chan` property[5] tying it to the channel in which it was created, and the `Status` field discussed above.[6] The remaining `Msg` properties (`Signal`, `ShouldFire` and `Result`) are used for blocking on synchronous channels, and are discussed in §9.4.

The `Chan<A>` class implements a lock-free bag of messages of type `A`. Its `Chords` property gives, as an array, all of the chords that mention the channel, while the `IsSync` property records whether the channel is synchronous. The key operations on a channel are:

- `AddPending`, which takes a message payload and atomically adds a `Msg<A>` with `PENDING` status to the bag and returns it.
- `FindPending` attempts to locate and return—but not remove—*some* message with `PENDING` status. *Its precise semantics is subtle*, and is closely connected to the status protocol shown in Figure 9.1. There are three possible outcomes of a call to `FindPending`:

  "Yes": if `FindPending` returns a (non-`null`) message, that message was atomically observed to have status `PENDING`. Of course, by the time control is returned to the caller, the status may have been altered by

[5] The `get` and `set` keywords are used to specify the existence of "getters" and "setters" for properties in .NET interfaces. Properties externally look like fields, but they can be read-only, write-only, or read-write. Internally, setting or getting a property invokes the corresponding getter or setter method.

[6] The `Chan` property is just a convenience for the presentation in this paper. It is avoided in the real implementation for the sake of space.

a concurrent thread (see Figure 9.1, and notice that no thread owns the token in the `PENDING` status).

"Maybe": if `FindPending` returns **`null`** *and* its out-parameter[7] `sawClaims` is **`true`**, all that can be concluded is that some message was observed with a `CLAIMED` status (see Figure 9.1, and notice that a message with `CLAIMED` status might be reverted to `PENDING` by another thread at any time).

"No": if `FindPending` returns **`null`** *and* its out-parameter `sawClaims` is **`false`**, then there was some atomic instant during its execution at which all messages in the bag were marked `CONSUMED` (see Figure 9.1, and notice that a message with `CONSUMED` status can never be reverted to `PENDING`).

[7] A method parameter marked **`out`** in C♯ is passed as a reference that is considered uninitialized and can be updated by the method (with visible effect for the caller).

There is no explicit method for removing a message. As we mentioned earlier, a message can be logically removed from the bag by marking it as `CONSUMED` (leaving the bag implementation to physically delete it when convenient).

`Match` is a simple, immutable class used to record the data making up a matched pattern: a chord, and an array of `CLAIMED` messages[8] sufficient to fire it.

Finally, the `Chord` class represents a join pattern, which is simply a (heterogeneous) array of channels. The class includes a `TryClaim` method for trying to claim a given message (presumed to be on a channel that is part of the chord) together with enough other messages to satisfy the join pattern. Its implementation is given in the next section.

[8] This array is *heterogeneous*: it contains messages of varying types. In situations like this, we introduce an additional interface (here `Msg` without a type parameter) that represents, essentially, an existential package quantifying over the difference in types. We will gloss over this point from here on. See Kennedy and Russo (2005) for a detailed explanation.

9.3 THE CORE ALGORITHM: RESOLVING A MESSAGE

We have already discussed the key safety property for a Joins implementation: pattern matching and message consumption should be atomic. In addition, an implementation should ensure at least the following liveness property (assuming a fair scheduler):

If a chord can fire, eventually some *chord is fired.*[9]

[9] Notice that this property does not guarantee fairness; see §9.7.

Our strategy is to drive the firing of chords by the concurrent arrival of each message: each sender must "resolve" its own message. We consider a message *resolved* if it has been added to the appropriate channel bag, and *one* of the following holds:

1. It is marked `CLAIMED` by the sending thread, along with sufficiently many other messages to fire a chord; *or*
2. It is marked `CONSUMED` by another thread, and hence was used to fire a chord; *or*
3. No pattern can be matched using only the message and messages that arrived *prior* to it.

Ensuring that each message is eventually resolved is tricky, because message bags and statuses are constantly, concurrently in flux. In particular, just as one thread determines that its message `msg` does not enable any chord, another message from another thread may arrive that enables a chord involving `msg`.

▶ THE KEY IDEA is that each sender need only take responsibility for chords involving its messages and the messages that arrived prior to it; if a later sender enables a chord, that later sender is responsible for it. But given the highly concurrent nature of message bags, what does it mean for one message to arrive before another?

There is no need to provide a direct way of asking this question. Instead, we rely on the atomicity of the bag implementation (in the sense of §3.4). Atomicity means that we can think of calls to `AddPending` and `FindPending` (along with CASes to `Status`) as being executed atomically, in some global sequential order. In particular, all messages—even those added to distinct bags—can be semantically ordered by "arrival," *i.e.,* the time of their insertion. The bag interface does not provide a way to observe this ordering directly, but `FindPending` is guaranteed to respect it. For example, consider a thread that inserts a message into one bag, and then looks for a message in a different bag:

```
Msg m1 = bag1.AddPending(x);
bool sawClaims;
Msg m2 = bag2.FindPending(out sawClaims);
```

Suppose that `m2 = null` and `sawClaims = false`, in other words that the call to `FindPending` on `bag2` says that there were (atomically) no `PENDING` messages. By the time that call returns, `bag2` might in fact contain some `PENDING` messages—*but they can only be messages that arrived* after *`m1` did*. Thus, atomicity is the foundation for our idea of "message responsibility": the "instantaneous time" at which we insert a message to send is the pivot point determining which other messages we must consider when looking for a pattern that the message enables.

Figure 9.3 gives our implementation of message resolution. The `Resolve` method takes a message `msg` that has already been added to the appropriate channel's bag and loops until the message has been resolved. It first attempts to "claim" a chord involving `msg`, successively trying each chord in which `msg`'s channel is involved (lines 5–9). The `Chord` class's `TryClaim` method either returns an array of messages (which includes `msg`) that have all been `CLAIMED` by the current thread, or `null` if claiming failed. In the latter case, the `retry` by-reference[10] parameter is set to true if any of the involved message bags contained a message `CLAIMED` by another thread; otherwise, `retry` is unchanged by the chord's `TryClaim` method.

Cumulatively, the `retry` flag records whether an externally-`CLAIMED` message was seen in *any* failing chord. We must track such `CLAIMED` messages be-

[10] A by-reference parameter in C♯ must be initialized prior to method invocation; changes made to the parameter within the method are visible to the caller.

```
1  Match Resolve(Msg msg) {
2    var backoff = new Backoff();
3    while (true) {
4      bool retry = false;
5      foreach (var chord in msg.Chan.Chords) {
6        Msg[] claims = chord.TryClaim(msg, ref retry);
7        if (claims != null)
8          return new Match(chord, claims);
9      }
10     if (!retry || msg.Status == Stat.CONSUMED)
11       return null;
12     backoff.Once();
13   }
14 }
```

Figure 9.3: Resolving a message

cause they are unstable, in the sense that they may be reverted to `PENDING` (Figure 9.1), possibly enabling a chord for which the sender is still responsible.

The first way a message can be resolved—by claiming it and enough other messages to make up a chord—corresponds to the `return` on line 8. The second two ways correspond to the `return` on line 11. If none of the three resolution conditions hold, we must try again. We perform exponential backoff (line 12) in this case, because repeated retrying can only be caused by contention over messages. Resolution may fail to terminate, but only if the system as a whole is making progress (according to our liveness property above); see §9.7 for a proof sketch.

Figure 9.4 gives the implementation of the `TryClaim` for the `Chord` class,[11] which works in two phases:

- In the first phase (lines 8–17), `TryClaim` tries to *locate* sufficiently many `PENDING` messages to fire the chord. It is required to claim `msg` in particular. If it is unable to find enough messages, it exits (line 15) without having written anything to shared memory, which bodes well for its cache coherence behavior (§2.3.1). Channels are always listed in chords in a consistent, global order, which is needed to guarantee liveness (§9.7).

- Otherwise, the `TryClaim` enters the second phase (lines 20–28), wherein it attempts to claim each message. The message-level `TryClaim` method performs a CAS on the `Status` field, ensuring that only one thread will succeed in claiming a given message. If at any point we fail to claim a message, we roll back all of the messages claimed so far (lines 23–24). The implementation ensures that the `Chans` arrays for each chord are ordered consistently, so that in any race at least one thread entering the second phase will complete the phase successfully (§9.7).

[11] The `partial` keyword in C♯ provides a way of splitting a class definition up into several pieces.

```
partial class Chord {
  Chan[] Chans;  // the channels making up this chord

  Msg[] TryClaim(Msg msg, ref bool retry) {
    var msgs = new Msg[Chans.length];  // cached

    // locate enough pending messages to fire chord
    for (int i = 0; i < Chans.Length; i++) {
      if (Chans[i] == msg.Chan) {
        msgs[i] = msg;
      } else {
        bool sawClaims;
        msgs[i] = Chans[i].FindPending(out sawClaims);
        retry = retry || sawClaims;
        if (msgs[i] == null) return null;
      }
    }

    // try to claim the messages we found
    for (int i = 0; i < Chans.Length; i++) {
      if (!msgs[i].TryClaim()) {
        // another thread won the race; revert
        for (int j = 0; j < i; j++)
          msgs[j].Status = Stat.PENDING;
        retry = true;
        return null;
      }
    }

    return msgs; // success: each message CLAIMED
  }
}
```

Figure 9.4: Racing to claim a chord involving `msg`

The code in Figure 9.4 is a simplified version of our implementation that does not handle patterns with repeated channels, and does not stack-allocate or recycle message arrays. These differences are discussed in §9.6.

9.4 SENDING A MESSAGE: FIRING, BLOCKING AND RENDEZVOUS

Message resolution does not depend on the (a)synchrony of a channel, but the rest of the message-sending process does. In particular, when a message on an asynchronous channel is resolved with "no pattern matched," the sending

process is *finished*; but on a synchronous channel, the sender must *wait* until a pattern is matched and the message is consumed, so that it can calculate the reply to return.

To further complicate matters, chords can contain arbitrary mixtures of the two types of channel, so the protocols for sending on each type are intertwined. A key aspect of these protocols is determining which thread executed the body of a matched pattern:

- The body of an *asynchronous chord* (*i.e.,* one involving no synchronous channels) is executed by a newly-spawned thread; its return type must be `void`.
- The body of a *synchronous chord* (*i.e.,* one involving at least one synchronous channel) is executed by *exactly one* of the threads that sent a message on one of the involved synchronous channels.

These requirements are part of the semantics of the Joins library.

The code for sending messages is shown in Figure 9.5, with separate entry points `SyncSend` and `AsyncSend`. The actions taken while sending depend, in part, on the result of message resolution:

Send	We `CLAIMED`	They `CONSUMED`	No match
Sync	Fire (14)	Wait for result (6)	Block (6)
Async	(AC) Spawn (32) (SC) Wake (35–41)	Return (28)	Return (28)

where AC and SC stand for asynchronous chord and synchronous chord respectively.

▸ First we follow the path of a synchronous message, which begins by adding and resolving the message (lines 2–3). If the message was resolved by claiming it and enough additional messages to fire a chord, all relevant messages are immediately consumed (line 11). Otherwise, either another thread has `CONSUMED` the message, or no match was possible. In either case, the synchronous sender must wait (line 6).

Each synchronous message has a `Signal` associated with it. `Signals` provide methods `Block` and `Set`, allowing synchronous senders to block[12] and be woken. Calling `Set` triggers the signal:

[12] `Block` spinwaits a bit first; see §9.6

- If a thread has already called `Block`, it is then awoken and the signal is reset.
- Otherwise, the next call to `Block` will immediately return (instead of waiting), again resetting the signal.

We ensure that `Block` and `Set` are each called by at most one thread; the `Signal` implementation then ensures that waking only occurs as a result of triggering the signal (no "spurious wakeups").

```
R SyncSend<R, A>(Chan<A> chan, A a) {
  Msg msg = chan.AddPending(a);     // make our message visible
  Match mat = Resolve(msg);         //   and then resolve it

  if (mat == null) {                // msg CONSUMED, or no match
    msg.Signal.Block();             // wait for pattern match
    mat = msg.ShouldFire;           // non-null if woken by async
    if (mat == null)                // is this a rendezvous?
      return msg.Result;            //   return chord body's result
  } else {                          // we resolved msg by claiming,
    ConsumeAll(mat.Claims);         //   so consume the messages
  }

  var r = mat.Fire();               // execute the chord body
  // rendezvous with any other sync senders (they will be waiting)
  for (int i = 0; i < mat.Chord.Chans.Length; i++) {
    if (mat.Chord.Chans[i].IsSync && mat.Claims[i] != msg) {
      mat.Claims[i].Result = r;     // transfer result to sender
      mat.Claims[i].Signal.Set();   //   and wake it up
    }
  }
  return (R)r;
}
```

```
void AsyncSend<A>(Chan<A> chan, A a) {
  Msg msg = chan.AddPending(a);     // make our message visible
  Match mat = Resolve(msg);         //   and then resolve it

  if (mat == null) return;          // msg CONSUMED, or no match
  ConsumeAll(mat.Claims);           // resolved by CLAIMING

  if (mat.Chord.IsAsync) {          // asynchronous chord:
    new Thread(mat.Fire).Start();   //   fire in a new thread
  } else {                          // synchronous chord:
    // wake up the first synchronous caller
    for (int i = 0; i < mat.Chord.Chans.Length; i++) {
      if (mat.Chord.Chans[i].IsSync) {
        mat.Claims[i].ShouldFire = mat;  // tell it what to consume
        mat.Claims[i].Signal.Set();      // and wake it up
        return;
      }
    }
  }
}
```

Figure 9.5: Sending a message

There are two ways a blocked, synchronous sender can be woken: by an asynchronous sender or by another synchronous sender (which we call "rendezvous"). In the former case, the (initially `null`) `ShouldFire` field will contain a `Match` object whose body the synchronous caller is responsible for executing on behalf of the asynchronous sender (line 14). In the latter case, `ShouldFire` remains `null`, but the `Result` field will contain the result of a chord body as executed by another synchronous sender, which is immediately returned (line 9).

We regroup at line 14, in which the synchronous sender actually executes the chord body. It could have arrived at this line in two ways: either by matching a chord itself, or else by being woken by an *asynchronous* sender. In either case, after executing the body, it must then wake up any other synchronous senders involved in the chord and inform them of the result (the other side of rendezvous, lines 16–21). For simplicity, we ignore the possibility that the chord body raises an exception, but proper handling is easy to add and is addressed by the benchmarked implementation.

▸ Now we consider sending an asynchronous message. Just as before, we begin by adding and resolving the message to send (lines 25–26). If either the message was `CONSUMED` by another thread (in which case that thread is responsible for firing the chord) or no pattern is matchable (in which case the message is left for another thread to consume later), we immediately exit (line 28).

On the other hand, if we resolved the message by claiming it and enough other messages to fire a chord, we proceed by consuming all involved messages (line 29). If the chord is asynchronous (its pattern involves only asynchronous channels) we spawn a new thread to execute the chord body asynchronously (line 32). Otherwise at least one of the messages we just consumed belongs to a synchronous sender that is now blocked. Although multiple synchronous callers can be combined in a single chord, exactly one of them is chosen to execute the chord; afterwards it will share the result with (and wake up) all the others (rendezvous). After picking a synchronous sender to wake (lines 35–41), we tell it which chord to fire with which messages (line 37).

9.5 KEY OPTIMIZATIONS

While the implementation outlined above is already much more scalable than a lock-based implementation, it needs a bit more work to be competitive with hand-built synchronization constructs. In this section, we describe three key optimizations whose benefit is shown experimentally in §9.8.

9.5.1 *Lazy message creation*

It is not always necessary to allocate a message or add it to a bag in order to send it. For example, in the `Lock` class (Chapter 8), when sending an `Acquire` message we could first check to see whether a corresponding `Release` message is available, and if so, immediately claim and consume it without ever touching the `Acquire` bag. This shortcut saves both on allocation and potentially on interprocessor communication.

To implement such a shortcut in general, we add an optimistic "fast path" for sending a message that attempts to immediately find only the *other* messages needed to fire a chord. If no chord can be matched in this way, the code reverts to the "slow path," *i.e.,* the regular implementation of sending messages as described above. The implementation is straightforward, so we omit it.[13]

[13] Our reagents implementation in Chapter 11 does give the details of a similar optimization; see §11.5.2.

As an aside, this optimization appears closely connected to the idea of "dual data structures" described in §2.4.6. Consider sending on a synchronous channel with this optimization. We can view that action as a kind of operation that is "partial," in the sense that it can only be completed if one of the relevant join patterns is enabled. If so, the operation is carried out immediately by changing the state of some fine-grained concurrent data structure. Otherwise, the *request* to perform the operation is recorded (in the form of a message on the synchronous channel), just as it would be in a dual data structure. We discuss this point further in Chapter 12.

9.5.2 *Specialized channel representation*

Consider that a **`void`**, asynchronous channel (e.g. `Lock.Release`) is just a bag of indistinguishable tokens.[14] Sophisticated lock-based implementations of join patterns typically optimize the representation of such channels to a simple counter, neatly avoiding the cost of allocation for messages that are just used as signals.[15] We have implemented a similar optimization, adapted to suit our scalable protocol.

[14] A **`void`** *synchronous* channel, on the other hand, is a bag of distinct signals for waiting senders.

[15] Benton *et al.* (2004); Fournet *et al.* (2003)

The main challenge in employing the counter representation is that, in our protocol, it must be possible to *tentatively* decrement the counter (the analog of claiming a message), in such a way that other threads do not incorrectly assume the message has actually been consumed. Our approach is to represent **`void`**, asynchronous message bags as a word sized pair of half-words, separately counting claimed and pending messages. Implementations of, for example, `Chan.AddPending` and `Msg.TryClaim` are specialized to atomically update the shared-state word by CASing in a classic optimistic loop (§2.4.2). For example, we can claim a "message" as shown in Figure 9.6.

More importantly, `Chan.FindPending` no longer needs to traverse a data structure but can merely atomically read the bag's encoded state once, setting `sawClaimed` if the `claimed` count is non-zero.

```
bool TryClaimToken() {
  while (true) {
    uint startState = chan.state;  // shared channel state
    ushort claimed;
    ushort pending = Decode(startState, out claimed);
    if (pending > 0) {
      var nextState = Encode(claimed + 1, pending - 1);
      if CAS(ref chan.state, startState, nextState) return true;
    } else {
      return false;
    }
  }
}
```

Figure 9.6: Claiming a "PENDING" asynchronous message on a **void** channel represented using counters

While the counter representation avoids allocation, it does lead to more contention over the same shared state (compared with a proper bag). It also introduces the possibility of overflow, which we ignore here. Nevertheless, we have found it to be beneficial in practice (§9.8), especially for non-singleton resources like `Semaphore.Release` messages.

9.5.3 *Message stealing*

In the implementation described in §9.4, when an asynchronous sender matches a synchronous chord, it consumes all the relevant messages, and then wakes up one of the synchronous senders to execute the chord body. If the synchronous sender is actually blocked—so that waking requires a context switch—significant time may elapse before the chord is actually fired.

Since we do not provide a fairness guarantee, we can instead permit "stealing": we can wake up one synchronous caller, but roll back the rest of the messages to `PENDING` status, putting them back up for grabs by currently-active threads—including the thread that just sent the asynchronous message. In low-traffic cases, messages are unlikely to be stolen; in high-traffic cases, stealing is likely to lead to better throughput. This strategy is similar to the one taken in Polyphonic C♯,[16] as well as the "barging" allowed by the java.util.concurrent synchronizer framework.[17]

Some care must be taken to ensure our key liveness property still holds: when an asynchronous message wakes a synchronous sender, it moves from a safely resolved state (`CLAIMED` as part of a chord) to an unresolved state (`PENDING`). There is no guarantee that the woken synchronous sender will be able to fire a chord involving the original asynchronous message (see Benton *et al.* (2004) for an example). Yet `AsyncSend` simply returns to its caller. We must somehow ensure that the original asynchronous message is successfully

[16] Benton *et al.* (2004), "Modern concurrency abstractions for C#"

[17] Lea (2005), "The java.util.concurrent synchronizer framework"

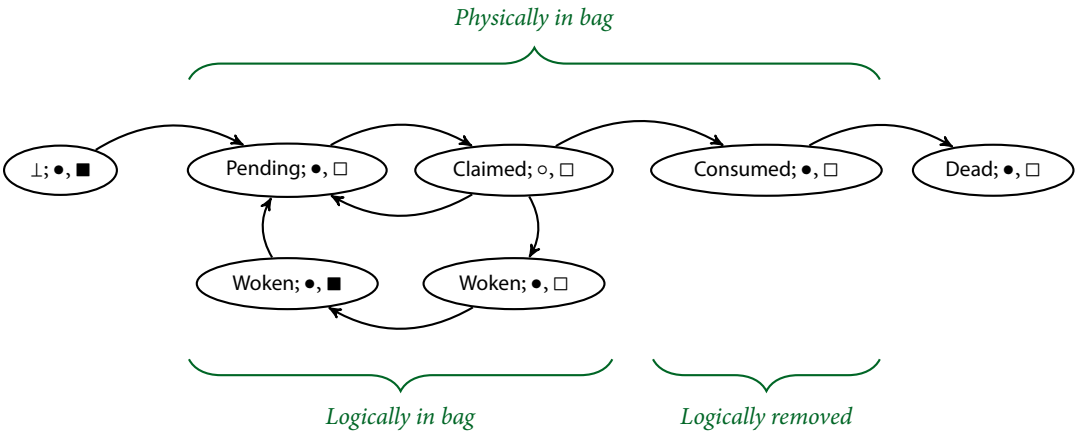


Figure 9.7: Per-message protocol, revised to support stealing

resolved. Thus, when a synchronous sender is woken, we record the asynchronous message that woke it, transferring responsibility for re-resolving the message.

To track message responsibility, we revise the message status protocol by introducing a new state, `WOKEN`; see Figure 9.7. A synchronous message is marked `WOKEN` if an asynchronous sender is transferring responsibility, and `CONSUMED` if a synchronous sender is going to fire a chord involving it. In both cases, the signal is set after the status is changed; in the latter case, it is set after the chord body has actually executed and the return value is available. `Resolve` is revised to return `null` at any point that the message is seen at status `WOKEN` (as well as `CONSUMED`).

As the protocol in Figure 9.7 shows, the `WOKEN` status ensures that a blocked synchronous caller is woken only once, which is important both to ensure correct use of the associated `Signal`, and to ensure that the synchronous sender will only be responsible for *one* waking asynchronous message. Only the thread that created a message (and is blocked waiting for it to enable a chord) is allowed to clear the `WOKEN` status: the additional token "■", which the sending thread gains ownership of when it originally creates the message, must be momentarily given up in order to move from `WOKEN` to `PENDING` in the protocol.

The new code for an asynchronous sender notifying a synchronous waiter is shown in Figure 9.8. Most of the paths through the code work as they did before. However, when the asynchronous message is `CLAIMED` as part of a *synchronous* chord (line 10), that chord is *not* immediately `CONSUMED`. Instead the asynchronous sender chooses one of the synchronous messages in the chord to wake up. And instead of informing that synchronous sender of the entire `CLAIMED` chord, it merely informs[18] the sender of the asynchronous message that woke it up (line 15), transferring responsibility for sending `msg`. While the synchronous sender's message is moved to the `WOKEN` status, all of the other `CLAIMED` messages—including the original `msg`—are reverted to

[18] Via a new field, `WakeMsg`, of type `Msg`.

```
1  void AsyncSendWithStealing<A>(Chan<A> chan, A a) {
2    Msg msg = chan.AddPending(a);     // make our message visible
3    Match mat = Resolve(msg);         //   and then resolve it
4
5    if (mat == null) {                // msg CONSUMED, or no match
6      return;                         //   so our work is done
7    } else if (mat.Chord.IsAsync) {   // CLAIMED asynchronous chord:
8      ConsumeAll(mat.Claims);         //   consume it, and
9      new Thread(mat.Fire).Start();   //   fire in a new thread
10   } else {                          // CLAIMED synchronous chord:
11     bool foundSleeper = false;
12     for (int i = 0; i < mat.Chord.Chans.Length; i++) {
13       if (m.Chord.Chans[i].IsSync && !foundSleeper) {
14         foundSleeper = true;              // the first sync sender:
15         m.Claims[i].WakeMsg = msg;        //   hand over msg
16         m.Claims[i].Status = Stat.WOKEN;  //   set wakeup type
17         m.Claims[i].Signal.Set();         //   wake it up
18       } else {
19         m.Claims[i].Status = Stat.PENDING; // relinquish other claims
20       }
21     }
22   }
23 }
```

Figure 9.8: Sending an asynchronous message, as revised to support stealing

`PENDING` (line 19), which allows them to be stolen before the synchronous sender wakes up.

Figure 9.9 gives the revised code for sending a synchronous message in the presence of stealing. A synchronous sender loops until its message is resolved by claiming a chord (exit on line 10), or by another thread consuming it (exit on line 17). In each iteration of the loop, the sender blocks (line 14); even if its message has already been `CONSUMED` as part of a synchronous rendezvous, it must wait for the signal to get its return value. In the case that the synchronous sender is woken by an asynchronous message (lines 19–20), it records the waking message and ultimately tries once more to resolve its own message. We perform exponential backoff every time this happens, since continually being awoken only to find messages stolen indicates high traffic.

After every resolution of the synchronous sender's message `msg`, the sender retries sending the last asynchronous message that woke it, if any (lines 11–12, 29–30). Doing so fulfills the liveness requirements outlined above: the synchronous sender takes responsibility for sending the asynchronous message that woke it. The `RetryAsync` method is similar to `AsyncSend`, but uses an already-added message rather than adding a new one. It is crucial to call `RetryAsync` only when holding *no claims* on messages—otherwise,

```
R SyncSendWithStealing<R, A>(Chan<A> chan, A a) {
  Msg wakeMsg = null;                  // last async msg to wake us
  Match mat = null;
  var backoff = new Backoff();         // accumulate exp. backoff

  Msg msg = chan.AddPending(a);        // make our message visible
  while (true) {                       // until CLAIMED or CONSUMED
    mat = Resolve(msg);                // (re)resolve msg

    if (mat != null) break;            // claimed a chord; exit
    if (wakeMsg != null)               // responsible for async msg?
      RetryAsync(wakeMsg);             //   retry sending it

    msg.Signal.Block();                // wait for pattern match

    if (msg.Status == Stat.CONSUMED) { // synchronous rendezvous:
      return msg.Result;               //   return body's result
    } else {                           // async wakeup (WOKEN):
      wakeMsg = msg.WakeMsg;           //   take responsibility
      msg.Status = Stat.PENDING;       //   get ready to retry
    }

    backoff.Once();                    // let others see PENDING msg
  }

  // we resolved msg by claiming it and the rest of a chord:
  ConsumeAll(mat.claims);              // so consume the chord

  if (wakeMsg != null)                 // retry last async waker,
    RetryAsync(wakeMsg);               // *after* consuming msg

  var r = mat.Fire();                  // execute the chord body
  // rendezvous with any other sync senders (they will be waiting)
  for (int i = 0; i < mat.Chord.Chans.Length; i++) {
    if (mat.Chord.Chans[i].IsSync && mat.Claims[i] != msg) {
      mat.Claims[i].Result = r;        // transfer result to sender
      mat.Claims[i].Signal.Set();      //   and wake it up
    }
  }
  return (R)r;
}
```

Figure 9.9: Sending a synchronous message while coping with stealing

`RetryAsync` might loop, forever waiting for those claims to be reverted or consumed. On the other hand, it is fine to retry the message even if it has already been successfully consumed as part of a chord;[19] `RetryAsync` will simply exit in this case.

[19] It will often be the case that the asynchronous message is `CONSUMED` on line 27, for example.

9.6 PRAGMATICS AND EXTENSIONS

There are a few smaller differences between the presented code and the actual implementation, which:

- Avoids boxing (allocation) and downcasts whenever possible.[20]
- Does not allocate a fresh message array every time `TryClaim` is called. The implementation stack-allocates an array[21] in `SyncSend` and `AsyncSend`, and reuses this array for every call to `TryClaim`.
- Handles nonlinear patterns, in which a single channel appears multiple times.

[20] On .NET, additional polymorphism (beyond what the code showed) can help avoid uses of `object`.

[21] Stack-allocated arrays are not directly provided by .NET, so we use a custom value type built by polymorphic recursion.

An important pragmatic point is that the `Signal` class first performs some spinwaiting before blocking. Spinning is performed on a memory location associated with the signal, so each spinning thread will wait on a distinct memory location whose value will only be changed when the thread should be woken, an implementation strategy long known to perform well on cache-coherent architectures (§2.4.6). The amount of spinning performed is determined adaptively on a per-thread, per-channel basis.

It should be straightforward to add timeouts and nonblocking attempts for synchronous sends[22] to our implementation, because we can always use `cas` to consume a message we have previously added to a bag to cancel an in-progress send—which will, of course, fail if the send has already succeeded.

[22] That is, a way to send a synchronous message only when it is immediately matches a join pattern.

Finally, to add channels with ordering constraints one needs only use a queue or stack rather than a bag for storing messages. Switching from bags to fair queues and disabling message stealing yields per-channel fairness for joins. In Dining Philosophers, for example, queues would guarantee that requests from waiting philosophers are fulfilled before those of philosophers that have just eaten. Such guarantees come at the cost of decreased parallelism, since they entail sequential matching of join patterns. At an extreme, programmers can enforce a round-robin scheme for matching patterns using an additional internal channel.[23]

[23] Russo (2008), "Join Patterns for Visual Basic"

9.7 CORRECTNESS

The most appropriate specification for our algorithm is something like the process-algebraic formulation of the join calculus,[24] perhaps treated as a canonical atomic spec (§3.4). In that specification, multiple messages are consumed—and a chord is fired—in a single step. We have not yet carried out a rigorous proof that our implementation satisfies this specification. We have,

[24] Fournet and Gonthier (1996), "The reflexive CHAM and the join-calculus"

however, identified what we believe are the key lemmas—one safety property, one liveness property—characterizing the `Resolve` method:

Lemma 4 (Resolution Safety)**.** Assume that `msg` has been inserted into a channel. If a subsequent call to `Resolve(msg)` returns, then `msg` is in a resolved state; moreover, the return value correctly reflects how the message was resolved.

Lemma 5 (Resolution Liveness)**.** Assume that threads are scheduled fairly. If a sender is attempting to resolve a message, eventually *some* message is resolved by its sender.

Recall that there are three ways a message can be resolved: it and a pattern's worth of messages can be marked `CLAIMED` by the calling thread; it can be marked `CONSUMED` by another thread; and it can be in an arbitrary status when it is determined that there are not enough messages sent *prior* to it to fire a chord.

Safety for the first two cases is fairly easy to show (Figure 9.1):

- Once a message is `CLAIMED` by a thread, the next change to its status is by that thread.
- Once a message is `CONSUMED`, its status never changes.

These facts mean that interference cannot "unresolve" a message that has been resolved in those two ways. The other fact we need to show is that the `retry` flag is only **`false`** if, indeed, no pattern is matched using only the message and messages that arrived before it. Here we use the assumptions about bags (§9.2), together with the facts about the status flags just given.

Now we turn to the liveness property. Notice that a call to `Resolve` fails to return only if `retry` is repeatedly **`true`**. This can only happen as a result of messages being `CLAIMED`. We can prove, using the consistent ordering of channels during the claiming process, that if any thread reaches the claiming process (lines 19–28 of Figure 9.4), *some* thread succeeds in claiming a pattern's worth of messages. The argument goes: claiming by one thread can fail only if claiming/consuming by another thread has succeeded, which means that the other thread has managed to claim a message on a higher-ranked channel. Since there are only finitely-many channels, some thread must eventually succeed in claiming the last message it needs to match a pattern.

Using both the safety and liveness property for `Resolve`, we expect the following overall liveness property to hold:

Conj 1. Assume that threads are scheduled fairly. If a chord can be fired, eventually *some* chord is fired.

The key point here is that if a chord can be fired, then in particular some message, together with its predecessors, *does* match a pattern, which rules out the possibility that the message is resolved with no pattern matchable.

9.8 PERFORMANCE

We close our discussion of join patterns with an experimental study of our implementation. The result is clear: the implementation scales well and, with the optimizations of §9.5, performs competitively with purpose-built synchronization constructs.

9.8.1 *Methodology*

Scalable concurrent algorithms are usually evaluated by targetted microbenchmarking, with focus on contention effects and fine-grained parallel speedup.[25] To evaluate our implementation, we constructed a series of microbenchmarks for seven classic coordination problems: dining philosophers, producers/-consumers, mutual exclusion, semaphores, barriers, rendezvous, and reader-writer locking.

[25] Mellor-Crummey and Scott (1991); Michael and Scott (1996); Herlihy, Luchangco, Moir, and W.N. N Scherer, III (2003); William N. Scherer, III and Scott (2004); Hendler *et al.* (2004); Fraser and Tim Harris (2007); Cederman and Tsigas (2010); Hendler *et al.* (2010)

Our solutions for these problems are fully described in Chapter 8. They cover a spectrum of shapes and sizes of join patterns. In some cases (producer/consumer, locks, semaphores, rendezvous) the size and number of join patterns stays fixed as we increase the number of processors, while in others a single pattern grows in size (barriers) or there are an increasing number of fixed-size patterns (philosophers).

Each benchmark follows standard practice for evaluating synchronization primitives: we repeatedly synchronize, for a total of k synchronizations between n threads. We use $k \geq 100{,}000$ and average over three trials for all benchmarks. To test interaction with thread scheduling and preemption, we let n range up to 96—twice the 48 cores in our benchmarking machine.

Each benchmark has two variants for measuring different aspects of synchronization:

PARALLEL SPEEDUP In the first variant, we simulate doing a *small* amount of work between synchronization episodes (and during the critical section, when appropriate). By performing some work, we can gauge to what extent a synchronization primitive inhibits or enables parallel speedup. By keeping the amount of work small, we gauge in particular speedup for *fine-grained* parallelism, which presents the most challenging case for scalable coordination.

PURE SYNCHRONIZATION In the second variant, we synchronize in a tight loop, yielding the cost of synchronization in the limiting case where the actual work is negligible. In addition to providing some data on constant-time overheads, this variant serves as a counterpoint to the previous one: it ensures that scalability problems were not hidden by doing too much work between synchronization episodes. Rather than looking for speedup, we are checking for slowdown.

To simulate work, we use .NET's `Thread.SpinWait` method, which spins in a tight loop for a specified number of times (and ensures that the spinning

Figure 9.10: Speedup on simulated fine-grained workloads

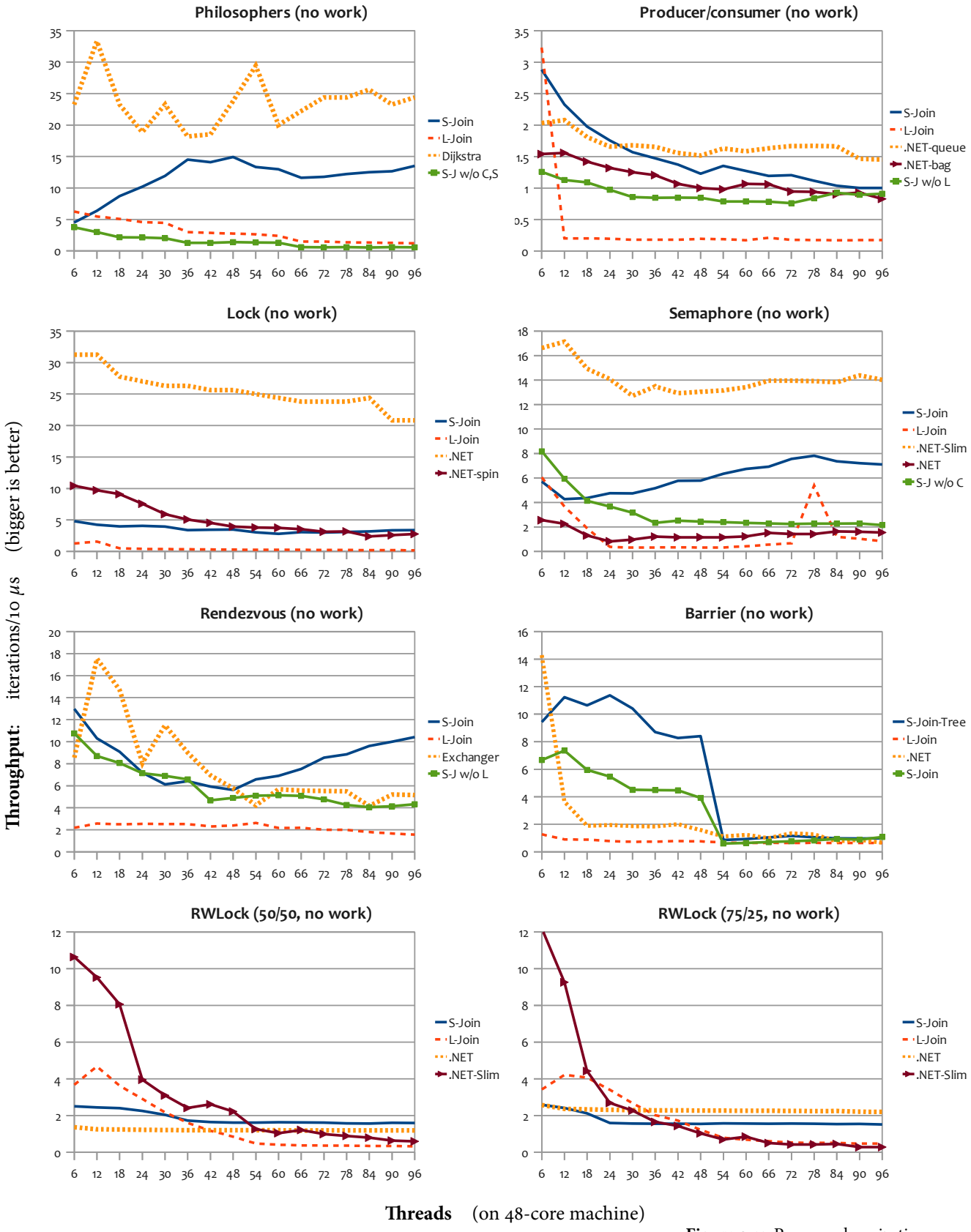


Figure 9.11: Pure synchronization performance

will not be optimized away). To make the workload a bit more realistic—and to avoid "lucky" schedules—we randomize the number of spins between each synchronization, which over 100,000 iterations will yield a normal distribution of total work with very small standard deviation. We ensure that the same random seeds are provided across trials and compared algorithms, so we always compare the same amount of total work. The mean spin counts are determined per-benchmark and given in the next section.

For each problem we compare performance between:

- a join-based solution using our fully-optimized implementation (S-Join, for "scalable joins"),
- a join-based solution using Russo's library (L-Join, for "lock-based joins"),
- at least one purpose-built solution from the literature or .NET libraries (label varies), and
- when relevant, our implementation with some or all optimizations removed to demonstrate the effect of the optimization (*e.g.*, S-J w/o S,C for dropping the Stealing and Counter optimizations; S-J w/o L for dropping the Lazy message creation optimization).

We detail the purpose-built solutions below.

Two benchmarks (rendezvous and barriers) required extending Russo's library to support multiple synchronous channels in a pattern; in these cases, and only in these cases, we use a modified version of the library.

Our benchmarking machine has four AMD "Magny-Cours" Opteron 6100 Series 1.7GHz processors, with 12 cores each (for a total of 48 cores), 32GB RAM, and runs Windows Server 2008 R2 Datacenter. All benchmarks were run under the 64-bit CLR.

9.8.2 *Benchmarks*

The results for all benchmarks appear in Figure 9.10 (for parallel speedup) and Figure 9.11 (for pure synchronization). The axes are consistent across all graphs: the x-axis measures the number of threads, and the y-axis measures throughput (as iterations performed every 10μs). Larger y values reflect better performance.

For measuring parallel speedup, we used the following mean spin counts for simulated work:

Benchmark	In crit. section	Out of crit. section	
Philosophers	25		5,000
Prod/Cons	N/A	producer	5,000
		consumer	500
Lock	50		200
Semaphore	25		100
Rendezvous	N/A		5,000
Barrier	N/A		10,000
RWLock	50		200

With too little simulated work, there is no hope of speedup; with too much, the parallelism becomes coarse-grained and thus insensitive to the performance of synchronization. These counts were chosen to be high enough that at least one implementation showed speedup, and low enough to yield significant performance differences.

The particulars of the benchmarks are as follows, where n is the number of threads and k the *total* number of iterations (so each thread performs k/n iterations):

- PHILOSOPHERS Each of the n threads is a philosopher; the threads are arranged around a table. An iteration consists of acquiring and then releasing the appropriate chopsticks. We compare against Dijkstra's original solution, using a lock per chopstick, acquiring these locks in a fixed order, and releasing them in the reverse order.

- PRODUCER/CONSUMER We let $n/2$ threads be producers and $n/2$ be consumers. Producers repeatedly generate trivial output and need not wait for consumers, while consumers repeatedly take and throw away that output. We compare against the .NET 4 `BlockignCollection` class, which transforms a nonblocking collection into one that blocks when attempting to extract an element from an empty collection. We wrap the `BlockingCollection` around the .NET 4 `ConcurrentQueue` class (a variant of Michael and Scott's classic lock-free queue) and `ConcurrentBag`.

- LOCK An iteration consists of acquiring and then releasing a single, global lock. We compare against both the built-in .NET lock (a highly-optimized part of the CLR implementation itself) and `System.Threading.SpinLock` (implemented in .NET).

- SEMAPHORE We let the initial semaphore count be $n/2$. An iteration consists of acquiring and then releasing the semaphore. We compare to two .NET semaphores: the `Semaphore` class, which wraps kernel semaphores, and `SemaphoreSlim`, a faster, pure .NET implementation of semaphores.

- RENDEZVOUS The n threads perform a total of k synchronous exchanges as quickly as possible. Unfortunately, .NET 4 does not provide a built-in

library for rendezvous, so we ported Scherer *et al.*'s *exchanger*[26] from Java; this is the exchanger included in java.util.concurrent.

[26] William N. Scherer, III *et al.* (2005), "A scalable elimination-based exchange channel"

- BARRIERS An iteration consists of passing through the barrier. We show results for both the tree and the flat versions of the join-based barrier. We compare against the .NET 4 `Barrier` class, a standard sense-reversing barrier.

- RWLOCK An iteration consists of (1) choosing at random whether to be a reader or writer and (2) acquiring, and then releasing, the appropriate lock. We give results for 50-50 and 75-25 splits between reader and writers. We compare against two .NET implementations: the `ReaderWriterLock` class, which wraps the kernel RWLocks, and the `ReaderWriterLock-Slim` class, which is a pure .NET implementation.

9.8.3 *Analysis*

The results of Figure 9.10 demonstrate that our scalable join patterns are competitive with—and can often beat—state of the art custom libraries. Application-programmers can solve coordination problems in the simple, declarative style we have presented here, and expect excellent scalability, even for fine-grained parallelism.

In evaluating benchmark performance, we are most interested in the slope of the throughput graph, which measures scalability with the number of cores (up to 48) and then scheduler robustness (from 48 to 96). In the parallel speedup benchmarks, both in terms of scalability (high slope) and absolute throughput, we see the following breakdown:

S-Join clear winner	Producer/consumer, Semaphore, Barrier
S-Join competitive	Philosophers, Lock, Rendezvous, RWLock 50/50
.NET clear winner	RWLock 75/25

The .NET concurrency library could benefit from replacing some of its primitives with ones based on the joins implementation we have shown—the main exception being locks. With some low-level optimization, it should be feasible to build an entire scalable synchronization library around joins.

The performance of our implementation is mostly robust as we oversubscribe the machine. The Barrier benchmark is a notable exception, but this is due to the structure of the problem: every involved thread must pass through the barrier at every iteration, so at $n > 48$ threads, a context switch is *required* for every iteration. Context switches are very expensive in comparison to the small amount of work we are simulating.

Not all is rosy, of course: the pure synchronization benchmarks show that scalable join patterns suffer from constant-time overheads in some cases, especially for locks. The table below approximates the overhead of pure

synchronization in our implementation compared to the best .NET solution, by dividing the scalable join pure synchronization time by the best .NET pure synchronization time:

Overhead compared to best custom .NET solution

n	Phil	Pr/Co	Lock	Sema	Rend	Barr	RWL
6	5.2	0.7	6.5	2.9	0.7	1.5	4.2
12	5.2	0.9	7.4	4.0	1.7	0.3	3.9
24	1.9	0.9	6.6	3.0	1.1	0.2	1.8
48	1.6	1.2	7.4	2.3	1.0	0.2	1.4

(*n* threads; smaller is better)

Overheads are most pronounced for benchmarks that use .NET's built-in locks (Philosophers, Lock). This is not surprising: .NET locks are mature and highly engineered, and are not themselves implemented as .NET code. Notice, too, that in Figure 9.11 the overhead of the spinlock (which *is* implemented within .NET) is much closer to that of scalable join patterns. In the philosophers benchmark, we are able to compensate for our higher constant factors by achieving better parallel speedup, even in the pure-synchronization version of the benchmark.

One way to decrease overhead, we conjecture, would be to provide compiler support for join patterns. Our library-based implementation spends some of its time traversing data structures representing user-written patterns. In a compiler-based implementation, these runtime traversals could be unrolled, eliminating a number of memory accesses and conditional control flow. Removing that overhead could put scalable joins within striking distance of the absolute performance of .NET locks. On the other hand, such an implementation would probably not allow the dynamic construction of patterns that we use to implement barriers.

In the end, constant overhead is trumped by scalability: for those benchmarks where the constant overhead is high, our implementation nevertheless shows strong parallel speedup when simulating work. The constant overheads are dwarfed by even the small amount of work we simulate. Finally, even for the pure synchronization benchmarks, our implementation provides competitive scalability, in some cases extracting speedup despite the lack of simulated work.

▸ EFFECT OF OPTIMIZATIONS Each of the three optimizations discussed in §9.5 is important for achieving competitive throughput. Stealing tends to be most helpful for those problems where threads compete for limited resources (Philosophers, Locks), because it minimizes the time between resource release and acquisition, favoring threads that are in the right place at the right time.[27] Lazy message creation improves constant factors across the board, in some cases (Producer/Consumer, Rendezvous) also aiding scalability. Finally, the counter representation provides a considerable boost for benchmarks, like Semaphore, in which the relevant channel often has multiple pending messages.

[27] This is essentially the same observation Doug Lea made about *barging* for abstract synchronizers (Lea 2005).

▸ PERFORMANCE OF LOCK-BASED JOINS Russo's lock-based implementation of joins is consistently—often dramatically—the poorest performer for both parallel speedup and pure synchronization. On the one hand, this result is not surprising: the overserialization induced by coarse-grained locking is a well-known problem (§2.4). On the other hand, Russo's implementation is quite sophisticated in the effort it makes to shorten critical sections. The implementation includes all the optimizations proposed for POLYPHONIC C♯[28], including a form of stealing, a counter representation for void-async channels (simpler than ours, since it is lock-protected), and bitmasks summarizing the state of messages queues for fast matching. Despite this sophistication, it is clear that lock-based joins do not scale.

We consider STM-based join implementations in §12.2.2.

[28] Benton *et al.* (2004), "Modern concurrency abstractions for C#"

10

Reagents

▸ Synopsis This chapter presents the design of reagents, both in terms of philosophical rationale (§10.1) and as motivated by a series of examples (§10.2, §10.5). The chapter shows in particular how to write all of the algorithms described in Chapter 2 concisely and at a higher-than-usual level of abstraction. It also demonstrates how the join calculus can be faithfully embedded into the reagent combinators (§10.3). The full API is given in Appendix E.

"Opposites are not contradictory but complementary."

—Niels Bohr

10.1 Overview

In the preceding two chapters, we saw that message-passing primitives—in the guise of join patterns—can be used to declaratively express scalable synchronization. In this chapter, we dig deeper and wider. We show how join patterns can themselves be built from more fundamental ingredients, and how, by supplementing those ingredients with shared state, we can support scalable *data structures* as well. The key is to embrace *both sides* of several apparent oppositions: isolation versus interaction, disjunction versus conjunction, and activity versus passivity. The result is a new abstraction—reagents—built up through combinators encompassing these dualities.

Reagents are a new instance of an old idea: representing computations as data. The computations being represented are scalable concurrent operations, so a value of type `Reagent[A,B]` represents a function from `A` to `B` that internally updates shared state, synchronizes with other threads, or both. Because the computations are data, however, they can be combined in ways that go beyond simple function composition. Each way of combining reagents corresponds to a way of combining their internal interactions with concurrent data structures. Existing reagents—for example, those built by a concurrency expert—can be composed by library users, without those users knowing their internal implementation. This way of balancing *abstraction* and *composition* was pioneered with Concurrent ML,[1] and is now associated with *monads*[2] and *arrows*.[3] Our contribution is giving a set of combinators appropriate for expressing and composing scalable concurrent algorithms, with a clear cost semantics and implementation story (given in Chapter 11).

[1] Reppy (1992), "Higher-order concurrency"

[2] Peyton Jones and Wadler (1993), "Imperative functional programming"

[3] Hughes (2000), "Generalising monads to arrows"

10.1.1 *Isolation versus interaction*

We begin by revisiting the "warring paradigms" of shared-state and message-passing concurrency. Chapter 2 argued that there is no *semantic* difference between the two approaches to concurrent programming, because each can easily be "programmed up" as syntactic sugar in the other. But there *are* differences in programming style and implementation pragmatics:

> *[The] choice of synchronization and communication mechanisms are the most important aspects of concurrent language design. . . . Shared-memory languages rely on the imperative features of the sequential sub-language for interprocess communication, and provide separate synchronization primitives to control access to the shared state. Message-passing languages provide a single unified mechanism for both synchronization and communication. Note that the distinction between shared-memory and message-passing languages has to do with the style of process interaction; many message-passing languages, including CML, are implemented in shared address spaces.*
>
> —John Reppy, "Concurrent programming in ML"

While there are some problems with taking the above quote as a *definition* of shared-state and message-passing concurrency,[4] it draws attention to an important difference in the typical programming style associated with the paradigms. Shared-state concurrent programming is often focused on *isolation*,[5] ensuring that updates to shared state appear to take place atomically (in a non-overlapping fashion; §2.2.4, §3.4). Synchronous message passing is just the opposite: it demands *interaction*, requiring that sends and receives *do* overlap. Both phenomena have a long history:

> *In 1965, Dijkstra demonstrated that mutual exclusion of events is a fundamental programming concept. In 1975, [Hoare] showed that the opposite idea, the coincidence of events, is just as important! This strikes me as the most profound idea incorporated in CSP.*
>
> —Per Brinch Hansen, "The invention of concurrent programming"

Chapter 2 also argued that the challenge of concurrent programming is managing sharing and timing. In our view, isolation and interaction are *both* fundamental tools for addressing this challenge—and (given expressivity §2.2.1) there is no reason to make only one of them available to the exclusion of the other. More than that: although they appear to be opposed, isolation and interaction can profitably be *combined*. For example, elimination backoff (§2.4.5) alternates between attempting an isolated update to a data structure (usually using **cas**) and interaction with a partner with which an operation can be eliminated.

The most primitive reagents provide pure isolated access to shared state or pure interaction through channels. As we show in §10.2.3, the elimination stack algorithm can then be elegantly expressed by *combining* two simpler reagents: one for performing an atomic update to the stack, and one for elimination against another thread. The combination is expressed as a *choice*, which we describe next.

[4] For example, as we mentioned in Chapter 2, traditionally shared-state constructs like Brinch Hansen-style monitors and STM tightly weld synchronization to communication. On the other hand, in practice channels often offer asynchronous interfaces even for receiving, by supporting a "tentative" receive operation or a callback interface.

[5] *i.e.,* mutual exclusion

10.1.2 *Disjunction versus conjunction*

"*Composable concurrency abstractions!*" has been the rallying cry of work on software transactional memory (STM),[6] Concurrent ML (CML),[7] and many others. As it happens, though, there is another duality latent in proposals for composition:

- With STM, programmers can *sequence* multiple reads/updates with shared state into a larger atomic block. STM-style sequencing is a kind of *conjunction*, because *all* of the underlying commands are executed by the resulting atomic step.
- With CML, programmers can take the *choice* of multiple "events," each encompassing potential interaction through synchronous channels. CML-style choice is a kind of *disjunction*, because *one* of the interactions is executed by the resulting CML event.

We have, in fact, already seen *both* forms of composition at work together with join patterns. Each chord represents a conjunction of interactions across a collection of channels, while the collection of chords mentioning a given channel represent a disjunction, since a message will participate in exactly one of them.[8] Some STM implementations, most notably Haskell's,[9] also provide a choice operation that can be used to combine atomic transactions.

Choice and sequencing are the basic ways of combining reagents, and arbitrary such combinations are permitted.[10]

[6] Shavit and Touitou (1995), "Software transactional memory"

[7] Reppy (1992), "Higher-order concurrency"

[8] Join patterns are, in a sense, written in disjunctive normal form (disjunctions of conjunctions). See §10.3.

[9] Tim Harris *et al.* (2005), "Composable memory transactions"

[10] There are some important caveats when using certain low-level reagent techniques. See §10.4 and §10.5.

10.1.3 *Activity versus passivity*

Reagents are much like functions: they are inert values that must be invoked to be useful. Reagents offer two means of invocation: active and passive.

In chemistry, a reagent is a participant in a reaction, and reagents are subdivided into *reactants*, which are consumed during reaction, and *catalysts*, which enable reactions but are not consumed by them. Similarly for us:[11]

- Invoking a reagent as a reactant is akin to calling it as a function: its internal operations are performed once, yielding a result or blocking until it is possible to do so. This is a single "reaction."
- Invoking a reagent as a catalyst instead makes it passively available as a participant in reactions. Because catalysts are not "used up," they can participate in many reactions in parallel.

The distinction is much like the one between sending and receiving messages in the join calculus: sending a message is an active process, while receiving messages is done through passive join patterns, which are permanently present.

[11] Chemical metaphors in concurrent programming go back to the Chemical Abstract Machine (Berry and Boudol 1992), a precursor to the join calculus.

10.2 THE HIGH-LEVEL COMBINATORS

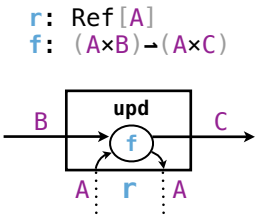
We now introduce the basic reagent building blocks as a Scala library, and use them to build a series of increasingly complex concurrent algorithms. By the end of the section, we will have seen how to implement all of the algorithms described in Chapter 2, and several more besides.

```
// Isolated updates on refs (shared state)
upd:  Ref[A] ⇒ (A × B ⇀ A × C) ⇒ Reagent[B,C]

// Interaction on channels (message passing)
swap: Endpoint[A,B] ⇒ Reagent[A,B]

// Composition
+  : Reagent[A,B] × Reagent[A,B] ⇒ Reagent[A,B]
>> : Reagent[A,B] × Reagent[B,C] ⇒ Reagent[A,C]
*  : Reagent[A,B] × Reagent[A,C] ⇒ Reagent[A, B × C]

// Lifting pure functions
lift: (A ⇀ B) ⇒ Reagent[A,B]

// Post-commit actions
postCommit: (A ⇒ Unit) ⇒ Reagent[A,A]

// Invoking a reagent:
dissolve: Reagent[Unit,Unit] ⇒ Unit // as a catalyst
react:    Reagent[A,B] ⇒ A ⇒ B     // as a reactant,
                                   //   same as the ! method
```

Figure 10.1: The high-level reagent API (in Scala)

10.2.1 *Atomic updates on `Ref`s*

Memory is shared between reagents using the type `Ref[A]` of atomically-updatable references. The `upd` combinator (Figure 10.1) builds a `Reagent[B,C]` that atomically updates a `Ref[A]`; it takes an *update function* (of type $A \times B \rightharpoonup A \times C$), which tells how to transform a snapshot of the reference cell and some input into an updated value for the cell and some output. Using `upd`, we can express Treiber's stack in a readable and concise way, as shown in Figure 10.2.



Scala has a notion of *partial* functions (denoted with the arrow ⇀) whose domain be queried via an `isDefinedAt` predicate. Anonymous partial functions can be written as a series of cases enclosed in braces; the domain is then

any value matching at least one of the cases. For `push` and `tryPop` the case analysis is exhaustive, so the update functions are in fact total.[12]

[12] `Unit` is akin to `void`: it is a type with a single member, written `()`.

```
class TreiberStack[A] {
  private val head = new Ref[List[A]](Nil)
  val push: Reagent[A, Unit] = upd(head) {
    case (xs, x) ⇒ (x::xs, ())
  }
  val tryPop: Reagent[Unit, Option[A]] = upd(head) {
    case (x::xs, ()) ⇒ (xs,  Some(x))
    case (Nil,   ()) ⇒ (Nil, None)
  }
}
```

Figure 10.2: Treiber's stack, using reagents

Being reagents, `push` and `tryPop` are inert values. They can be invoked as reactants using the `!` method, which is pronounced "react." For a `Reagent[A,B]` the `!` method takes an `A` and returns a `B`.[13] When we invoke these reagents, we are really executing an optimistic retry loop (as in the "hand-written" version in §2.4.3) with built-in exponential backoff (§2.4.4); reagents systematize and internalize common patterns of scalable concurrency. But by exposing `push` and `tryPop` as reagents rather than methods, we enable further composition and tailoring by clients (using *e.g.,* the combiners in §10.2.3, §10.2.4).

[13] Scala permits infix notation for methods, so we can use a `TreiberStack s` by writing `s.push ! 42`.

While `tryPop`'s update function is total—it handles both empty and nonempty stacks—we can write a variant that drops the empty case:

```
val pop: Reagent[Unit, A] = upd(head) { case (x::xs, ()) ⇒ (xs, x) }
```

Now our update function is partial. *Et voilà*: invoking `pop` will block the calling thread unless or until the stack is nonempty.

Along similar lines, it is easy to write a semaphore as a concurrent counter:[14]

[14] *Cf.* §2.4.2 and §8.3.

```
class Counter {
  private val c = new Ref[Int](0)
  val inc = upd(c) { case (i, ()) ⇒ (i+1, i) }
  val dec = upd(c) { case (i, ()) if (i > 0) ⇒ (i-1, i) }
  val tryDec = upd(c) {
     case (i, ()) if (i == 0) ⇒ (0,   None)
     case (i, ())             ⇒ (i-1, Some(i))
  }
}
```

The `inc` and `dec` reagents provide the usual mechanism for acquiring and releasing resources from a semaphore. The `tryDec` reagent, on the other hand, makes it possible to *tentatively* acquire a resource; the return value indicates

whether one was available. It's worth taking a moment to compare the `Counter` implementation to the `TreiberStack`—after all, a counter is isomorphic to a stack containing only `unit` values. Expressing these data structures at a high level, using reagents, makes the connection easy to see.

▸ REAGENTS CAN FAIL TO REACT in one of two ways: *transiently* or *persistently*.

- TRANSIENT FAILURES arise when a reagent loses a race to CAS a location; they can only be caused by *active interference* from another thread. A reagent that has failed transiently will internally retry, rather than block, following the concurrency patterns laid out in Chapter 2.
- PERSISTENT FAILURES arise when a reagent places requirements on its environment—such as the requirement, with `pop` above, that the `head` reference yield a nonempty list. Such failures are persistent in the sense that only activity by another thread can enable the reagent to proceed. When faced with a persistent failure, a reagent should block until signaled that the underlying state has changed.[15] Blocking and signaling are entirely handled by the reagent implementation; there is therefore no risk of lost wakeups.

[15] See §2.2.3.

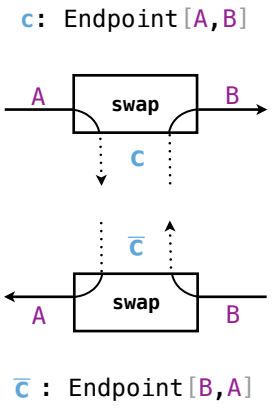
Any reagent that makes visible changes to state (by updating a reference or communicating on a channel) is subject to transient failures, which will silently cause a retry. The possibility and handling of persistent failures varies based on the combinator, so we describe the blocking behavior of each combinator as we encounter it.

The `upd(f)` reagent fails persistently only for those inputs on which `f` is undefined; a reaction involving such an update is blocked until the underlying reference changes.

10.2.2 *Synchronization: interaction within a reaction*

With reagents, updates to shared memory are isolated, so they cannot be used for interaction in which the parties are mutually aware. Reagents interact instead through *synchronous swap channels*, which consist of two complementary *endpoints*. The function `mkChan[A,B]` returns a pair of type

```
Endpoint[A,B] × Endpoint[B,A]
```



The combinator for communication is `swap` (see Figure 10.1), which lifts an `Endpoint[A,B]` to a `Reagent[A,B]`. When two reagents communicate on opposite endpoints, they provide messages of complementary type (`A` and `B`, for example) and receive each other's messages. On the other hand, if no complementary message is available, `swap` will block until a reaction can take place—a persistent failure.

There is no deep design principle behind the use of symmetric swap channels instead of the more common asymmetric channels. They are instead

motivated by the simple observation that an asymmetric—but synchronous—channel must already do all of the work that a swap channel does. In particular, threads blocked trying to receive from the channel must be queued; a swap channel just enables the queue to carry values as well as thread identities. Conversely, traditional channels can be recovered by choosing one end of a swap channel to carry `Unit` values:

```
val (sendEP, recvEP) = mkChan[A,Unit]
val send: Reagent[A, Unit] = swap(sendEP)
val recv: Reagent[Unit, A] = swap(recvEP)
```

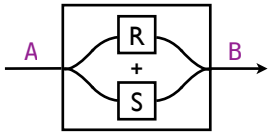
As with our version of join patterns, swap channels do not provide ordering guarantees: they are bags.[16] The motivation is the same as for join patterns: unordered channels provide greater potential for parallelism and less contention over centralized data.

Neither of these two choices are fundamental to the design of reagents.

[16] In this respect, our channels are two-sided exchangers (William N. Scherer, III *et al.* 2005).

10.2.3 *Disjunction of reagents: choice*

If `r` and `s` are two reagents of the same type, their *choice* `r + s` will behave like one of them, nondeterministically, when invoked. The choice is "mildly" left-biased: it will only attempt the right reagent on the left one failed, but unlike "truly" left-biased choice, the right-hand reagent is tried even when the left-hand one failed only *transiently*.[17] For the choice itself, failure depends on the underlying reagents. A choice fails persistently only when *both* of its underlying reagents have failed persistently. If either fails transiently, the choice reagent has failed transiently and should therefore retry.



[17] The `orElse` combinator in Haskell's STM is an example of "true" left-bias (Tim Harris *et al.* 2005).

The most straightforward use of choice is waiting on several signals simultaneously, but consuming only one of them once one is available. For example, if `c` and `d` are endpoints of the same type, then `swap(c) + swap(d)` is a reagent that will accept exactly one message, either from `c` or from `d`. If neither endpoint has a message available, the reagent will block until one of them does.

A more interesting use of choice is adding backoff strategies (§2.4.4, §2.4.5). For example, we can build an elimination backoff stack as follows:

```
class EliminationStack[A] {
  private val s = new TreiberStack[A]
  private val (elimPop, elimPush) = mkChan[Unit,A]
  val push: Reagent[A,Unit] = s.push + swap(elimPush)
  val pop:  Reagent[Unit,A] = s.pop  + swap(elimPop)
}
```

This simple pair of composite reagents give rise to a protocol of surprising complexity. Here is the chain of events when invoking `push`:[18]

[18] The `pop` protocol is nearly symmetric, except that `pop` can block; see below.

1. Because of the mild left-bias of choice, when `push` is invoked it will first attempt to push onto the underlying Treiber stack.

2. If the underlying push fails (transiently, due to a lost CAS race), `push` will then attempt to synchronize with a concurrent popper:

   (a) Following the strategy of lazy message creation (§9.5.1), `push` will first attempt to locate and consume a message waiting on the `elimPop` endpoint.

   (b) Failing that, `push` will create a message on the `elimPush` endpoint.

3. Because the underlying `s.push` only *transiently* failed, `push` will not block. It will instead *spinwait* briefly for another thread to accept its message along `elimPush`; the length of the wait grows exponentially, as part of the exponential backoff logic. Once the waiting time is up, the communication attempt is canceled, and the whole reagent is retried.

The protocol is a close facsimile of the elimination backoff strategy given in §2.4.5, but it *emerges naturally* from the reagent library implementation elaborated in Chapter 11.

Implementation details aside, we can reason about the blocking behavior of `EliminationStack` based on the failure semantics of choice. We deduce that `push` never blocks because the underlying `s.push` can only fail transiently, never persistently. On the other hand, `pop` can block because `s.pop` can fail persistently (on an empty stack) and `swap(elimPop)` can fail persistently (if there are no offers from pushers). Conversely, a blocked invocation of `pop` can be woken either by a normal push unto the underlying stack *or* through elimination.

▸ THE ELIMINATION STACK EMBODIES SEVERAL ASPECTS OF REAGENTS. First of all, it shows how reagents empower their clients through composition: as a client of `TreiberStack`, the `EliminationStack` is able to add a layer of additional functionality by using choice. Both semantic details (*e.g.,* blocking behavior) and implementation details (*e.g.,* backoff strategy) are seamlessly composed in the client. The other side of composability is abstraction: `EliminationStack` need not know or care about the precise implementation of `TreiberStack`. To make this abstraction more apparent (and also more useful), we can define a *generic* elimination stack, one that layers elimination as a mixin on top of any reagent-based stack:

```
trait ConcurrentStack[A] {
  val push: Reagent[A,Unit]
  val pop:  Reagent[Unit,A]
}

class WithElimination[A](s: ConcurrentStack[A])
extends ConcurrentStack[A] {
  private val (elimPop, elimPush) = mkChan[Unit,A]
  val push = s.push + swap(elimPush)
  val pop  = s.pop  + swap(elimPop)
}
```

Because the result of applying `WithElimination` is just another stack, it is possible to layer applications:

```
new WithElimination(new WithElimination(new TreiberStack[A]))
```

which approximates the array of elimination backoff channels.[19]

Elimination also demonstrates that isolation and interaction, far from being opposed, are in fact a potent combination.

[19] Layering elimination in this way is not terribly useful in conjunction with our representation of channels as concurrent bags, but it could be with other types of channels. See §13.2.

▸ CHOICE CAN BE EMPLOYED IN OTHER GENERIC WAYS AS WELL. For example, although we do not include it as an "official" combinator, it is easy to support a reagent for *timeout* that persistently fails until a certain amount of time has elapsed. A potentially-blocking reagent can then be (generically) embedded in a choice with a timeout, limiting the duration of blocking. Along similar lines, explicit *cancellation* of a potentially-blocking reagent can be programmed up by (roughly[20]) taking a choice with a `swap` reagent for a channel of cancellation events. Both timeouts and cancellations are important features for industrial-strength libraries like JUC and TBB, where they sometimes necessitate code duplication. With reagents, it is not even necessary for such support to be built into a library: a client can layer it on after the fact, tailoring the library to their own needs.

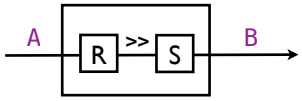
[20] Robust cancellation should inform the canceler whether the underlying reagent succeeded prior the cancellation attempt. This additional functionality can be programmed using the `postCommit` combinator described in §10.2.6.

10.2.4 *Conjunction of reagents: sequencing and pairing*

Choice offers a kind of disjunction on reagents. There are also two ways of *conjoining* two reagents, so that the composed reagent has the effect of *both* underlying reagents, atomically:

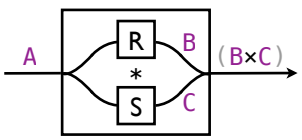
- End-to-end composition, via *sequencing*:

  if `r: Reagent[A,B]` and `s: Reagent[B,C]` then `r >> s: Reagent[A,C]`.



- Side-by-side composition, via *pairing*:

  if `r: Reagent[A,B]` and `s: Reagent[A,C]` then `r * s: Reagent[A,B × C]`.



These combinators differ only in information flow. Each guarantees that the atomic actions of both underlying reagents become a single atomic action for the composition. For example, if `s1` and `s2` are both stacks, then `s1.pop >> s2.push` is a reagent that will atomically transfer an element from the top of one to the top of the other. The reagent will block if `s1` is empty. Similarly, `s1.pop * s2.pop` will pop, in one atomic action, the top elements of both stacks, or block if either is empty.

Here we again see the benefits of the reagent abstraction. Both of the example combinations just mentioned work *regardless of how the underlying stacks are implemented*. If both stacks use elimination backoff, the conjoined operations will potentially use elimination on both simultaneously. This behavior is entirely emergent: it does not require any code on the part of the stack author, and it does not require the stack client to know anything about the implementation. Reagents can be composed in unanticipated ways.

Conjunctions provide a solution to the Dining Philosophers problem:[21] to consume two resources atomically, one simply conjoins two reagents that each consume a single resource. For example, if `c` and `d` are endpoints of type `Unit` to `A` and `B` respectively, then `swap(c) * swap(d)` is a reagent that receives messages on both endpoints simultaneously and atomically. There is no risk of introducing a deadlock through inconsistent acquisition ordering, because the reagents implementation is responsible for the ultimate acquisition order, and will ensure that this order is globally consistent.

[21] *Cf.* Chapter 8.

The failure behavior of conjunctions is dual to that of disjunctions: if *either* conjunct fails persistently, the entire conjunction fails persistently.

The implementation details for conjunctions are discussed later (Chapter 11), but a key point is that the performance cost is *pay as you go*. Single atomic reagents like `push` and `pop` execute a single CAS—just like the standard nonblocking algorithms they are meant to implement—even though these operations can be combined into larger atomic operations. The cost of conjunction is only incurred when a conjoined reagent is actually used. This is a crucial difference from STM, which generally incurs overheads regardless of the size of the atomic blocks.[22]

[22] See Chapter 12 for more discussion.

10.2.5 *Catalysts: passive reagents*

The `!` operator invokes a reagent as a reactant: the invocation lasts for a single reaction, and any messages the reagent sends are consumed by the reaction. But sometimes it is useful for invocations to persist beyond a single reaction, *i.e.*, to act as catalysts. For example, `zip` creates a catalyst that merges input from two endpoints and sends the resulting pairs to a third:[23]

[23] *Cf.* the join pattern for pairing in §8.3.

```
def zip(in1: Endpoint[Unit, A],
        in2: Endpoint[Unit, B],
        out: Endpoint[A × B, Unit]) =
  dissolve( (swap(in1) * swap(in2)) >> swap(out) )
```

The `dissolve` function takes a `Reagent[Unit, Unit]` and introduces it as a catalyst.[24] Operationally, in this example, that just means sending unit-carrying messages along `in1` and `in2` that are marked as "catalyzing," and hence are not consumed during reaction. The upshot is that senders along `in1` will see the catalyzing messages, look for messages along `in2` to pair with, and ultimately send messages along `out` (and symmetrically).

[24] For simplicity, we have not given a way to cancel catalysts after they have been introduced, but cancellation is easy to add.

Catalysts are, of course, not limited to message passing. The `zip` example above could be rephrased in terms of arbitrary reagents rather than just endpoints:

```
def zipR(in1: Reagent[Unit, A],
         in2: Reagent[Unit, B],
         out: Reagent[A × B, Unit])  =  dissolve((in1 * in2) >> out)
```

As these examples suggest, one important use of catalysts is directing (in this case, by "fusing" together) the flow of information through channels or other reagents.

Following the same progression we saw in Chapter 8, we can also define a "dual" to `zipR`, namely, an *arbiter*:

```
def arbiter(in1: Reagent[Unit, A],
            in2: Reagent[Unit, B],
            out: Reagent[A+B,  Unit]) =
  dissolve( ((in1 >> lift(inl)) + (in2 >> lift(inr))) >> out )
```

Here we employ the `lift` combinator to treat a Scala partial function as a kind of "pure" reagent that does not access shared state or interact on channels.[25] By using `>>` and `lift` together, we can wrap arbitrary transformations around the input or output of a reagent.

[25] Nothing guarantees that the lifted function actually *is* pure in this sense, which opens the door to side-effects that are invisible to the reagent library. We will take advantage of this fact later on (§10.5).

When a choice is consumed as a reactant, exactly one of its branches is used. But as a catalyst, a choice is *not* consumed, and so both of its branches may be used repeatedly in reactions with other reagents. Consequently, there is no semantic reason to dissolve a choice of reagents; dissolving them individually is equivalent:

```
def arbiter'(in1: Reagent[Unit, A],
             in2: Reagent[Unit, B],
             out: Reagent[A+B,  Unit]) = {
  dissolve(in1 >> lift(inl) >> out)
  dissolve(in2 >> lift(inr) >> out)
}
```

Catalysts could instead be expressed using an *active* thread that repeatedly invokes a reagent as a reactant. But allowing direct expression through `dissolve` is more efficient (since it does not tie up a thread) and allows greater parallelism (since, as with the `zip` example above, multiple reagents can react with it in parallel). The influence of the join calculus here is evident.

10.2.6 *Post-commit actions*

Reagents support "post commit actions", which comprise code to be run (for effect) after a reaction, *e.g.* code to signal or spawn another thread after an operation completes. The `postCommit` combinator (Figure 10.1) takes a function from `A` to `Unit` and produces a `Reagent[A,A]`. The post-commit action will be recorded along with the input of type `A`, which is passed along unchanged. Once the reaction completes, the action will be invoked on the stored input. The combinator is meant to be used in sequence with other combinators that will produce the appropriate input. For example, the reagent `pop >> postCommit(println)` will print the popped element from a stack after the pop has completed.

10.3 TRANSLATING JOIN PATTERNS

Throughout the discussion of reagents, we have emphasized connections to and inspiration from the join calculus (and, in particular, the implementation in Chapter 9). Now we are in a position to make the connection more clear by (loosely[26]) translating join patterns into reagents.

The basic idea is that a join pattern $c_1(x_1)\ \&\ \cdots\ \&\ c_n(x_n) \triangleright b$ can be interpreted directly as a catalyst.[27] The join operator & is interpreted as a conjunction $*$, the channel names c_i as `swap` instances, and the body b as a post-commit action. Altogether, the reagent corresponding to the pattern is

$$\begin{aligned} & [\![c_1(x_1)\ \&\ \cdots\ \&\ c_n(x_n) \triangleright b]\!] \\ \approx\ & (\texttt{swap}(c_1) * \cdots * \texttt{swap}(c_n))\ \texttt{>>}\ \texttt{postCommit}((x_1, \ldots, x_n) \Rightarrow b) \end{aligned}$$

A set of join patterns governing a set of channels can all be written in this way and dissolved as catalysts, which is equivalent to dissolving the choice of all the patterns.

While the above translation captures the general idea, it only works for the special case in which:

- all the channels in the join pattern are synchronous, and
- none of the channels expect a reply, *i.e.,* their reply types are all `Unit`.

Asynchronous channels can be coded up using reagents,[28] so to include one in a join pattern it is only necessary to use a reagent for receiving from such a channel[29] in place of `swap` on a synchronous channel. For a purely asynchronous chord, the body b can be wrapped with a thread spawn.

Handling replies on synchronous channels is, unfortunately, more difficult. Probably the best approach is the one outlined in Fournet and Gonthier (2002), in which a "reply continuation" is passed as an additional component of messages along synchronous channels. The body b must then be wrapped to send its return value along this channel. That said, many of the circumstances that would require a synchronous reply in the pure join calculus can be handled in a more direct way through reagents.

[26] There are a number of technical issues involved in a Russo-style API for join patterns—*e.g.,* not including parameters from `Unit`-carrying channels—which we gloss over here.

[27] We are using the original notation for join patterns, in which each pattern channels c_i with names x_i for the messages along those channels, and a body b in which the names x_i are bound (Fournet and Gonthier 1996).

[28] *e.g.,* `TreiberStack` (§10.2.1) or `MSQueue` (§10.6).

[29] *e.g.,* `pop` (§10.2.1).

The translation can be carried out in the setting of a library using implicits in Scala, or through advanced use of generics[30] in C♯. The broader point here is to show how join patterns can be understood in terms of reagent combinators. It should be noted, however, that not all aspects of the joins implementation are recovered through this encoding. For example, the message stealing optimization (§9.5.3) is not included in the reagent implementation, and would be difficult to add. On the other hand, lazy message creation (§9.5.1) is included for reagents (Chapter 11), and the specialization of asynchronous `Unit` channels to counters is directly expressible via reagents.

[30] See Russo (2007) for details.

10.4 ATOMICITY GUARANTEES

Because conjunction distributes over disjunction, every reagent built using the core combinators (Figure 10.1) can be viewed as a disjunction of conjunctions, where each conjunction contains some combination of updates and swaps. For such a reagent, reactions atomically execute all of the conjuncts within exactly one of the disjuncts. This STM-like guarantee works well for data structures that do not require traversal (§10.2.1) and for pure synchronization (*e.g.,* join patterns). It is, however, too strong for algorithms which need read (or traverse) shared memory without requiring those reads to be "visible," *i.e.,* , to participate in an atomic transaction. The next section will introduce *computed* reagents which allow invisible reads and writes, trading weaker guarantees for better performance. Nevertheless, the *visible* reads, writes, and synchronizations of these reagents are included in the above atomicity guarantee, and thus even computed reagents are composable.

When reagents attempt to interact through message passing, their atomicity becomes intertwined: they must react together in a single atomic step, or not. This requirement raises an important but subtle question: what should happen when isolation and interaction conflict? Consider two reagents that interact over a channel, but also each update the *same* shared reference. Atomicity demands that both reagents involved in the reaction commit together in one step, but isolation for references demands that the updates be performed in separate atomic steps!

This is an interesting semantic question for which, at present, we do not have a satisfying answer. One hopes that with sufficient application-programming experience using reagents, an answer will emerge.

In the meantime, for both simplicity and performance, we consider such situations to be *illegal*, and throw an exception when they arise; luckily, they are dynamically detectable. In practice, this rules out only compositions of certain operations within the *same* data structure, which are much less common than compositions across data structures. It is also straightforward to adopt an alternative approach, *e.g.* the one taken by Communicating Transactions (discussed in Chapter 12), which treats isolation/interaction conflicts as transient failures rather than as programming errors.

10.5 LOW-LEVEL AND COMPUTATIONAL COMBINATORS

```
// Low-level shared state combinators
read: Ref[A] ⇒ Reagent[Unit, A]
cas:  Ref[A] × A × A ⇒ Reagent[Unit, Unit]

// Computational combinators
computed: (A ⇀ Reagent[Unit, B]) ⇒ Reagent[A,B]
```

Figure 10.3: The low-level and computational combinators

10.5.1 *Computed reagents*

The combinators introduced in §10.2 are powerful, but they impose a strict phase separation: reagents are constructed prior to, and independently from, the data that flows through them. Phase separation is useful because it allows reagent execution to be optimized based on complete knowledge of the computation to be performed (see Chapter 11). But in many cases the choice of computation to be performed depends on the input or other dynamic data. The `computed` combinator (Figure 10.3) expresses such cases. It takes a partial function from `A` to `Reagent[Unit,B]` and yields a `Reagent[A,B]`. When the reagent `computed(f)` is invoked, it is given an argument value of type `A`, to which it applies the partial function `f`. If `f` is not defined for that input, the computed reagent issues a persistent (blocking) failure, similarly to the `upd` function. Otherwise, the application of `f` will yield another, dynamically-computed reagent, which is then invoked with `()`, the unit value.

In functional programming terms, the core reagent combinators of §10.2 can be viewed in terms of *arrows*,[31] which are abstract, composable computations whose structure is statically evident. With the addition of `computed`, reagents can also be viewed in terms of *monads*,[32] which extend arrows with dynamic determination of computational structure.

[31] Hughes (2000), "Generalising monads to arrows"

[32] Peyton Jones and Wadler (1993), "Imperative functional programming"

10.5.2 *Shared state: `read` and `cas`*

Although the `upd` combinator is convenient, it is sometimes necessary to work with shared state with a greater degree of control—especially when programming with computed reagents. To this end, we include two combinators, `read` and `cas` (see Figure 10.1), for working directly on `Ref` values. Together with the `computed` combinator described in §10.5.1, `read` and `cas` suffice to *build* `upd`:

- The `read` combinator is straightforward: if `r` has type `Ref[A]`, then `read(r)` has type `Reagent[Unit, A]` and, when invoked, returns a snapshot of `r`.

- The `cas` combinator takes a `Ref[A]` and two `A` arguments, giving the expected and updated values, respectively. Unlike its counterpart for F^{μ}_{cas}, the `cas` reagent does *not* yield a boolean result: a failure to CAS is transient failure of the whole reagent, and therefore results in a retry.

10.5.3 *Tentative reagents*

Because choice is left-biased, it can be used together with the remaining combinators to express *tentative* reagents: if `r` is a `Reagent[A,B]` then `r?` is a `Reagent[A,Option[B]]` defined by:

```
(r >> lift(Some)) + lift(_ ⇒ None)
```

The tentative `r?` first tries `r` (wrapping its output with `Some` if successful) and, *only on failure*, tries `lift(_ ⇒ None)`, which always succeeds. This allows a reaction to be attempted, without retrying or blocking it when it fails.

10.6 THE MICHAEL-SCOTT QUEUE

We close this chapter with a small case study: Michael and Scott (1996)'s lock-free queue (§4.2). Our implementation strategy readily scales to more complicated examples, such as concurrent skiplists or the lazy, lock-free set algorithm.[33] In all of these cases, we reap the usual benefits: a concise, composable and extensible exposition of the algorithm.

[33] Herlihy and Shavit (2008), "The Art of Multiprocessor Programming"

Unlike a stack, in which all activity focuses on the head, queues have two loci of updates. That means, in particular, that the `Ref`s used by its reagents may vary depending on the current state of the queue, which requires us to *compute* the necessary read, write, and update reagents. In addition, in order to enqueue a node we may traverse some of the data structure, looking for the tail—but the whole point of the algorithm is that this traversal *does not* need to be part of a large atomic transaction. Rather, once we believe we have located the tail, a single CAS will both perform our update and ensure that the node is *still* the tail.

A key question arises: what happens when reagents that perform invisible traversals are combined?

The answer is perhaps the most compelling aspect of reagent composition. Each invisible traversal produces, as its result, some small but *visible* reagent poised to perform an atomic operation—an operation that will both validate the traversal and enact an update. Sequencing two such traversals will compute (and sequence) two visible, validating reagents—a very small atomic transaction compared to, say, what STM would produce with even one traversal. Forthcoming commodity hardware, *e.g.,* Intel's Haswell, is designed to support small transactions directly and efficiently.

▶ Here is a brief review of the Michael-Scott algorithm.[34] The queue

[34] See §4.2 for a detailed account.

is represented as a *mutable* linked list, with a sentinel node at the head (front) of the queue. The head pointer always points to the current sentinel node; nodes are dequeued by a CAS to this pointer, just like Treiber stacks (but lagged by one node). The true tail of the queue is the unique node, reachable from the head pointer, with a null `next` pointer; thanks to the sentinel, such a node is guaranteed to exist. If the queue is empty, the same node will be the head (sentinel) and tail. Finally, as an optimization for enqueing, a "tail" pointer is maintained with the invariant that the true tail node is always reachable from it. The "tail" pointer may lag behind the true tail node, however, which allows the algorithm to work using only single-word CAS instructions.[35]

[35] Otherwise, it would have to link in a node *and* update the tail pointer in one step.

Figure 10.4: The Michael-Scott queue, using reagents

```
class MSQueue[A] {
  private case class Node(data: A, next: Ref[Node])
  private val initialSentinel = new Node(null)
  private val head = new Ref(initialSentinel)
  private val tail = new Ref(initialSentinel)

  val tryDeq: Reagent[Unit, Option[A]] = upd(head) {
    case (Node(_, Ref(n@Node(x, _))), ()) ⇒ (n, Some(x))
    case (emp,                        ()) ⇒ (emp, None)
  }
  private def findAndEnq(n: Node): Reagent[Unit,Unit] =
    read(tail) ! () match {
      case ov@Node(_, r@Ref(null)) ⇒    // found true tail
        cas(r, null, n) >> postCommit { cas(tail, ov, n)? ! () }
      case ov@Node(_, Ref(nv)) ⇒        // not the true tail
        cas(tail, ov, nv)? ! (); findAndEnq(n)
    }
  val enq: Reagent[A, Unit] = computed {
    (x: A) ⇒ findAndEnq(new Node(x, new Ref(null)))
  }
}
```

Our reagent-based implementation of the Michael-Scott queue is shown in Figure 10.4. The node representation is given as an inner *case* class. In Scala, case classes provide two features we take advantage of. First, the parameters to their constructors (here `data` and `next`) are automatically added as final fields to the class, which are initialized to the constructor argument values. Second, they extend pattern matching through **`case`** so that instances can be *deconstructed*. A pattern like **`case`** `Node(d, n)` matches any instance of the node class, binding `d` to its `data` field and `n` to its `next` field.

The `tryDeq` reagent is very similar to the `tryPop` reagent in `TreiberStack`, modulo the sentinel node. The reagent pattern matches on the sentinel node,

ignoring its `data` field by using `_`, the wildcard. The `next` field is then matched to a nested pattern, `Ref(n@Node(x, _))`. This pattern immediately reads the current value of the reference stored in `next`, binds that value to `n`, and then matches the pattern `Node(x,_)` against `n`. If the pattern matches—which it will any time the `next` field of the sentinel is non-null—the node `n` becomes the new `head` (and hence the new sentinel).

Since the location of the tail node is determined dynamically by the data in the queue, the `enq` reagent must itself be determined dynamically. For `enq`, we compute a dynamic reagent by first taking the given input `x`, creating a node with that data, and then calling a private function `findAndEnq` that will locate the tail of the queue and yield a reagent to update it to the new node. Since `findAndEnq` is private and tail-recursive, Scala will compile it to a loop.

The `findAndEnq` function searches for the *true* tail node (whose `next` field is `null`) starting from the `tail` pointer, which may lag. To perform the search, `findAndEnq` must read the `tail` pointer, which it does using the `read` combinator. *There is a subtle but important point here: this read occurs while the final reagent is being computed.* That means, in particular, that the read is *not* part of the computed reagent; it is a side-effect of computing the reagent. The distinction is important: such a read is effectively "invisible" to the outer reagent being computed, and thus is not guaranteed to happen atomically with it. As we explained above, invisible reads and writes are useful for avoiding compound atomic updates, but must be employed carefully to ensure that the computed reagent provides appropriate atomicity guarantees.

Once the tail pointer has been read, its value is pattern-matched to determine whether it points to the true tail. If it does, `findAndEnq` yields a `cas` reagent (§10.5.2) that will update the `next` field of the tail node from `null` to the new node. The attached post-commit action attempts to catch up the tail pointer through a `cas`, after the fact. Since the `cas` fails only if further nodes have been enqueued by other concurrent threads, we perform it tentatively (§10.5.3); it is not necessary or desirable to retry on failure.

If, on the other hand, the `tail` pointer is lagging, `findAndEnq` performs an *invisible* `cas` to update it. Since it may be racing with other enqueuers to catch up the tail, a failure to CAS is ignored here. Regardless of the outcome of the `cas`, the `findAndEnq` function will restart from a freshly-read `tail` pointer. Notice that in this case, an entire iteration of `findAndEnq` is executed with no visible impact or record on the final computed reagent—there is no extended redo log or compound atomic transaction. Only the final `cas` produced in the first case of `findAndEnq` is visible.

11

Implementing reagents

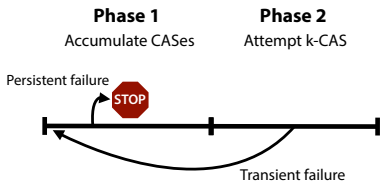
▸ Synopsis This chapter walks through the implementation of reagents (in Scala) in significant detail, which reveals the extent to which reagents turn patterns of scalable concurrency into a general algorithmic framework. It includes benchmarking results comparing multiple reagent-based collections to their hand-written counterparts, as well as to lock-based and STM-based implementations. Reagents perform universally better than the lock- and STM-based implementations, and are competitive with hand-written lock-free implementations.

"One of the joys of functional programming is the way in which apparently-exotic theory can have a direct and practical application, and the monadic story is a good example."

—Simon Peyton Jones, "Tackling the awkward squad"

11.1 Overview

When invoked, reagents attempt to *react*, which is *conceptually* a two-phase process: first, the desired reaction is built up; second, the reaction is atomically committed. We emphasize "conceptually" because reagents are designed to avoid this kind of overhead in the common case; it is crucial that reagents used to express scalable concurrent algorithms do not generate traffic to shared memory beyond what the algorithms require. We first discuss the general case (which imposes overhead) but return momentarily to the common (no overhead) case.



An attempt to react can fail during either phase. A failure during the first phase, *i.e.* a failure to build up the desired reaction, is always a persistent failure (§10.2.1). Persistent failures indicate that the reagent cannot proceed given current conditions, and should therefore block until another thread intervenes and causes conditions to change.[1] On the other hand, a failure during the second phase, *i.e.* a failure to commit, is always a transient failure (§10.2.1). Transient failures indicate that the reagent should retry, since the reaction was halted due to active interference from another thread. In general, an in-progress reaction is represented by an instance of the `Reaction` class, and contains three lists: the CASes to be performed, the messages to be consumed,[2] and the actions to be performed after committing. A `Reaction` thus resembles the redo log used in some STM implementations.[3]

In the common case that a reagent performs only one visible (§10.5) CAS or message swap, those components of the reaction are not necessary and hence are not used. Instead, the CAS or swap is performed immediately, compressing the two phases of reaction. Aside from avoiding extra allocations, this key optimization means that in the common case a `cas` or `upd` in a reagent

[1] *Cf.* §2.2.3.

[2] Message consumption ultimately boils down to additional CASes.

[3] Tim Harris *et al.* (2010), "Transactional Memory, 2nd edition"

leads to exactly one executed CAS during reaction, with no extra overhead.[4] When a reaction encompasses multiple visible CASes or message swaps, a costlier[5] *k*CAS protocol must be used to ensure atomicity. We discuss the *k*CAS protocol in §11.4, and the common case single CAS in §11.5.1.

▸ In the implementation, `Reagent[A,B]` is an abstract class all of whose subclasses are private to the library. These private subclasses roughly correspond to the public combinator functions, which are responsible for instantiating them; each subclass instance stores the arguments given to the combinator that created it.[6]

The one combinator that does *not* have a corresponding `Reagent` subclass is sequencing `>>`. Instead, the reagent subclasses internally employ *continuation-passing style* (CPS): each reagent knows and has control over the reagents that are sequenced after it, which is useful for implementing backtracking choice.[7] Thus, instead of representing the sequencing combinator `>>` with its own class, each reagent records its own *continuation* `k`, which is another reagent. For example, while the `cas` combinator produces a reagent of type `Reagent[Unit,Unit]`, the corresponding `CAS` class has a continuation parameter `k` of type `Reagent[Unit,R]`, and `CAS` extends `Reagent[Unit,R]` rather than `Reagent[Unit,Unit]`. The `R` stands for (final) result. The combinator functions are responsible for mapping from the user-facing API, which does not use continuations, to the internal reagent subclasses, which do. Each reagent initially begins with the trivial "halt" continuation, `Commit`, whose behavior is explained in §11.4.

Each subclass of `Reagent[A,B]` must implement its abstract methods:

```
abstract class Reagent[-A, +B] {   // the +/- are variance annotations
  def >>[C](next: Reagent[B,C]): Reagent[A,C]
  def canFail: Boolean // can this reagent fail?
  def canSync: Boolean // can this reagent send a message?
  def tryReact(a: A, rx: Reaction, offer: Offer[B]): Any
}
```

The `tryReact` method takes the input `a` (of type `A`) to the reagent and the reaction `rx` built up so far, and either:[8]

- completes the reaction, returning a result (type `B`), or
- fails, returning a failure (type `Failure`). The class `Failure` has exactly two singleton instances, `Block` and `Retry`, corresponding to persistent and transient failures respectively.

The remaining argument, `offer`, is used for synchronization and communication between reagents, which we explain next.

[4] An important implication is that the window of time between taking a snapshot of shared state and performing a CAS on it is kept small.

[5] Currently we implement *k*CAS in software, but upcoming commodity hardware is designed to support it primitively, at least for small *k*.

[6] Thus, reagents are an abstract data type whose instances are created using "smart constructors"—a very common idiom in functional programming.

[7] CPS is also needed for message passing: since a reagent will try to synchronize with any message it finds in a channel's bag, `swap` is also a form of backtracking choice.

[8] The `Any` type in Scala lies at the top of the subtyping hierarchy, akin to `Object` in Java. Here we are using `Any` to represent a union of the type `B` with the type `Failure`, to avoid extra allocation.

11.2 OFFERS

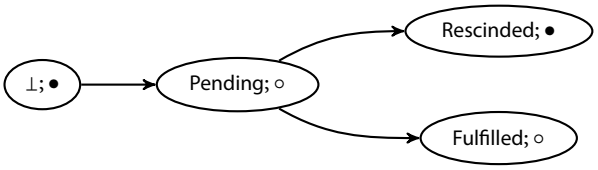
Message passing between reagents is synchronous, meaning that both reagents take part in a single, common reaction. In the implementation, this works by one reagent placing an *offer* to react in a location visible to the other.[9] The reagent making the offer either spinwaits or blocks until the offer is fulfilled;[10] if it spinwaits, it may later decide to withdraw the offer. The reagent accepting the offer sequences the accumulated `Reactions` of both reagents, and attempts to commit them together. Fulfilling the offer means, in particular, providing a final "answer" value that should be returned by the reagent that made the offer.[11] Each offer includes a status field, which is either `Pending`, `Rescinded`, or a final answer. Hence, the `Offer` class is parameterized by the answer type; a `Reagent[A,B]` will use `Offer[B]`. When fulfilling an offer, a reagent CASes its status from `Pending` to the desired final answer.

[9] *Cf.* messages in Chapter 9.

[10] It spinwaits iff it encountered a transient failure at any point during the reaction.

[11] The final answer will be the value passed the offer would have passed to its own `Commit` continuation.

Offers follow a very simple protocol (Chapter 4):



Due to the use of tokens, only the thread that originally created an offer can rescind it.

In addition to providing a basic means of synchronization, the offer data structure is used to resolve external choices. For example, the reagent `swap(ep1) + swap(ep2)` may resolve its choices internally by fulfilling an existing offer on `ep1` or `ep2`; but if no offers are available, the reagent will post a *single* offer to *both* endpoints, allowing the choice to be resolved externally. Reagents attempting to consume that offer will race to change a single, shared status field, thereby ensuring that such choices are resolved atomically.

Offers are made as part of the same `tryReact` process that builds and commits reactions. The `offer` argument to `tryReact` is `null` when the reaction is first attempted; reagents make offers lazily, just as join patterns create messages lazily (§9.5.1), as we will see below.

11.3 THE ENTRY POINT: REACTING

The code for performing a reaction is given in the `!` method definition for `Reagent[A,B]`, shown in Figure 11.1. This method provides two generalized versions of the optimistic retry loops we described in Chapter 2. The retry loops are written as local, tail-recursive functions, which Scala compiles down to loops.

The first retry loop, `withoutOffer`, attempts to perform the reaction without making visible offers to other reagents. It may, however, find and consume offers from other reagents as necessary for message passing.[12] To initiate the reaction, `withoutOffer` calls the abstract `tryReact` method with

[12] *Cf.* §9.5.1.

```
def !(a: A): B = {
  val backoff = new Backoff
  def withoutOffer(): B =
    tryReact(a, empty, null) match {
      case Block ⇒ withOffer()
      case Retry ⇒
        backoff.once()
        if (canSync) withOffer() else withoutOffer()
      case ans   ⇒ ans.asInstanceOf[B]
    }
  def withOffer(): B = {
    val offer = new Offer[B]
    tryReact(a, empty, offer) match {
      case (f: Failure) ⇒
        if (f == Block) park() else backoff.once(offer)
        if (offer.rescind) withOffer() else offer.answer
      case ans ⇒ ans.asInstanceOf[B]
    }
  }
  withoutOffer()
}
```

Figure 11.1: The `!` method, defined in `Reagent[A,B]`

the input `a`, an empty reaction to start with, and no offer. If the reaction fails in the first phase (a persistent failure, represented by `Block`), the next attempt must be made with an offer, to set up the blocking/signaling protocol. If the reaction fails in the second phase (a transient failure, represented by `Retry`), there is likely contention over shared data. To reduce the contention, `withoutOffer` performs one cycle of exponential backoff before retrying. If the reagent includes communication attempts, the retry is performed with an offer, since doing so increases chances of elimination (§10.2.3) without further contention. Finally, if both phases of the reaction succeed, the final answer `ans` is returned.

The second retry loop, `withOffer`, is similar, but begins by allocating an `Offer` object to make visible to other reagents. Once the offer has been made, the reagent can actually block when faced with a persistent failure; the offer will ensure that the attempted reaction is visible to other reagents, which may complete it, fulfilling the offer and waking up the blocked reagent. Blocking is performed by the `park` method provided by Java's `LockSupport` class.[13] On a transient failure, the reagent spinwaits, checking the offer's status. In either case, once the reagent has finished waiting it attempts to rescind the offer, which will fail if another reagent has fulfilled the offer.[14]

Initially, the reaction is attempted using `withoutOffer`, representing optimism that the reaction can be completed without making a visible offer.

[13] The `park`/`unpark` methods work similarly to the signals we used in Chapter 9, but they are associated with each thread and may suffer from spurious wakeups.

[14] Even if the reagent had blocked, it is still necessary to check the status of its offer, because `park` allows spurious wakeups.

11.4 THE EXIT POINT: COMMITTING

As mentioned in §11.2, the initial (outermost) continuation for reagents is an instance of `Commit`, which represents an "empty" reagent:

```
class Commit[A] extends Reagent[A,A] {
  def >>[B](next: Reagent[A,B]) = next
  def canFail = false
  def canSync = false
  def tryReact(a: A, rx: Reaction, offer: Offer[A]) =
    if (offer != null && !offer.rescind) offer.answer
    else if (rx.commit) a
    else Retry
}
```

The emptiness of `Commit` is reflected in the first three methods it defines: it is an identity for sequencing, and it does not *introduce* any failures or synchronizations. Any failure or synchronization must be due to some reagent sequenced prior to the `Commit` reagent, which always comes last.

The `tryReact` method of `Commit` makes the phase-transition from building up a `Reaction` object to actually committing it. If the reagent has made an offer, but has also completed the first phase of reaction, the offer must be rescinded before the commit phase is attempted—otherwise, the reaction could complete twice. As with the `!` method, the attempt to rescind the offer is in a race with other reagents that may be completing the offer. If `Commit` loses the race, it returns the answer provided by the offer. Otherwise, it attempts to commit the reaction, and if successful simply returns its input, which is the final answer for the reaction.

Committing a reaction requires a *k*CAS operation: *k* compare and sets must be performed atomically. This operation, which forms the basis of STM, is in general expensive and not available in most hardware.[15] There are several software implementations that provide nonblocking progress guarantees.[16] Reagents that perform a multiword CAS will inherit the progress properties of the chosen implementation.

For our prototype implementation, we have opted to use an extremely simple implementation that replaces each location to be CASed with a sentinel value, essentially locking the location. As the `Reaction` object is assembled, locations are kept in a consistent global order and hence avoids dead- and live-lock within the *k*CAS implementation. The advantage of this implementation, other than its simplicity, is that is has no impact on the performance of single-word CASes to references, which we expect to be the common case; such CASes can be performed directly, without any awareness of the *k*CAS protocol. Our experimental results in §2.4 indicate that even this simple *k*CAS implementation provides reasonable performance—much

[15] Though, as we have noted several times, hardware support is coming and may eventually be commonplace.

[16] Fraser and Tim Harris (2007); Luchangco *et al.* (2003); Attiya and Hillel (2008)

better than STM or coarse-grained locking—but a more sophisticated *k*CAS would likely do even better.

11.5 THE COMBINATORS

11.5.1 *Shared state*

Reads are implemented by the nearly trivial `Read` class:

```
class Read[A,R](ref: Ref[A], k: Reagent[A,R])
extends Reagent[Unit,R] {
  def >>[S](next: Reagent[R,S]) = new Read[A,S](ref, k >> next)
  def canFail = k.canFail
  def canSync = k.canSync

  def tryReact(u: Unit, rx: Reaction, offer: Offer[R]) = {
    if (offer != null) ref.addOffer(offer)
    k.tryReact(ref.get(), rx, offer)
  }
}
```

A read introduces neither failures nor synchronization, but its continuation might, so `canFail` and `canSync` defer to the values in `k`. The role of reading in `tryReact` is fairly straightforward: absent an offer, we simply perform the read and pass its result to the continuation `k`, with an unchanged reaction argument `rx`. However, if an offer is present, it is recorded in a bag of offers associated with the reference (via `addOffer`). Although the read itself cannot block, the value it reads could be the proximal cause of blocking in the continuation `k`. Thus, if the continuation is preparing to block (as evidenced by the non-`null` offer), logging the offer with the read reference will ensure that the entire reagent is woken up if the reference changes. Once the offer is rescinded or fulfilled, it is considered "logically removed" from the bag of offers stored with `ref`, and will be physically removed when convenient.[17]

[17] This is essentially the same approach we used to remove messages in Chapter 9.

While the `Read` class is private to the reagent library, the corresponding `read` combinator is exported:

```
def read[A](ref: Ref[A]): Reagent[Unit, A] =
  new Read[A,A](ref, new Commit[A])
```

All of the primitive reagent combinators are defined in this style, using the `Commit` reagent as the (empty) continuation. The result type `R` of the `Read` reagent is thus initially set at `A` when reading a `Ref[A]`.

The implementation of the `cas` combinator is given by the `CAS` class, shown in Figure 11.2. Its `tryReact` method is fairly simple, but it illustrates a key optimization we have mentioned several times: if the reaction so far has no CASes, and the continuation is guaranteed to succeed, then the entire reagent

```
class CAS[A,R](ref: Ref[A], ov: A, nv: A, k: Reagent[Unit,R])
extends Reagent[Unit,R] {
  def >>[S](next: Reagent[R,S]) = new CAS[A,S](ref, ov, nv, k >> next)
  def canFail = true
  def canSync = k.canSync

  def tryReact(u: Unit, rx: Reaction, offer: Offer[R]) =
    if (!rx.hasCAS && !k.canFail)  // can we commit immediately?
      if (ref.cas(ov, nv))         // try to commit
        k.tryReact((), rx, offer)  // successful; k MUST succeed
      else Retry
    else // otherwise must record CAS to reaction log, commit in k
      k.tryReact((), rx.withCAS(ref, ov, nv), offer)
}
```

Figure 11.2: The `CAS` class

is performing a single CAS and can thus attempt the CAS immediately. This optimization eliminates the overhead of creating a new `Reaction` object and employing the *k*CAS protocol, and it means that lock-free algorithms like `TreiberStack` and `MSQueue` behave just like their hand-written counterparts. If, on the other hand, the reagent may perform a *k*CAS, then the current `cas` is recorded into a new `Reaction` object,[18] which is passed to the continuation `k`. In either case, the continuation is invoked with the unit value as its argument.

11.5.2 *Message passing*

We represent each endpoint of a channel as a lock-free bag.[19] The lock-freedom allows multiple reagents to interact with the bag in parallel; the fact that it is a bag rather than a queue trades a weaker ordering guarantee for increased parallelism, but any lock-free collection would suffice.[20]

The endpoint bags store messages, which contain offers along with additional data from the sender:

```
case class Message[A,B,R](
  payload: A,           // sender's actual message
  senderRx: Reaction,   // sender's checkpointed reaction
  senderK: Reagent[B,R], // sender's continuation
  offer: Offer[R]       // sender's offer
)
```

Each message is essentially a checkpoint of a reaction in progress, where the reaction is blocked until the payload (of type `A`) can be swapped for a dual payload (of type `B`). Hence the stored sender continuation takes a `B` for input;

[18] The `withCAS` method performs a *functional update*, *i.e.*, returns a new `Reaction` object. It is important not to mutate the reaction objects: reagents use backtracking choice (§11.5.3), and at various points in the branches of such a choice reaction objects may be used to advertise synchronizations (§11.5.2).

[19] It is possible to build the bag itself using non-blocking reagents, thereby bootstrapping the library.

[20] The tradeoffs here are essentially the same as in Chapter 9.

it returns a value of type `R`, which matches the final answer type of the sender's offer.

The core implementation of `swap` is shown in the `Swap` class in Figure 11.3. If an offer is being made, it must be posted in a new message on the endpoint before any attempt is made to react with existing offers. This ordering guarantees that there are no lost wakeups: each reagent is responsible only for those messages posted prior to it posting its own message.[21] On the other hand, if there is no offer, `Swap` attempts to complete by consuming a message on the dual endpoint without ever creating (or publishing) its own message—exactly like the lazy message creation of §9.5.1.

[21] The design rationale and key safety/liveness properties here are exactly the same as those in Chapter 9.

Once the offer (if any) is posted, `tryReact` peruses messages on the dual endpoint using the tail-recursive loop, `tryFrom`. The loop navigates through the dual endpoint's bag using a simple cursor, which will reveal at least those messages present prior to the reagent's own message being posted to its endpoint. If a dual message is found, `tryFrom` attempts to complete a reaction involving it. To do this, it must *merge* the in-progress reaction of the dual message with its own in-progress reaction:

- The `++` operation on a pair of `Reaction` produces a new reaction with all of their CASes and post-commit actions.
- The `SwapK` inner class is used to construct a new continuation for the dual message. This new continuation uses the `withFulfill` method of `Reaction` to record a fulfillment[22] of the dual message's offer with the final result of the reagent in which that dual message was embedded.
- When `SwapK` is invoked as part of a reaction, it invokes the original continuation `k` with the payload of the dual message.
- If the reaction is successful, the final result is returned (and the result for the other reagent is separately written to its offer status). Recall that, in this case, the `Commit` reagent will first *rescind* the `offer` of `Swap.tryReact`, if any. Thus, if `Swap` had earlier advertised itself through its own message, it removes that advertisement before instead consuming an advertised message on the dual endpoint.[23] Just as in Chapter 9 consuming a message logically removed it, here rescinding or fulfilling the offer associated with a message logically removes the message from the bag.
- If the reaction fails, `tryFrom` continues to look for other messages. If no messages remain, `swap` behaves as if it were a disjunction: it fails persistently only if *all* messages it encountered led to persistent failures. The failure logic here closely resembles that of the `retry` flag in §9.3.

[22] Fulfillment includes waking the reagent if it is blocked on the offer.

[23] `Swap` may fail to rescind its message, but only if some other thread has fulfilled its offer; in this case, `Commit` aborts the attempt to consume a message on the dual channel and simply returns the result from the fulfilling thread.

```
class Swap[A,B,R](ep: Endpoint[A,B], k: Reagent[B, R])
extends Reagent[A,R] {
  def >>[S](next: Reagent[R,S]) = new Swap[A,S](ep, k >> next)
  def canFail = true
  def canSync = true

  // NB: this code glosses over some important details
  //     discussed in the text
  def tryReact(a: A, rx: Reaction, offer: Offer[R]) = {
    if (offer != null) // send message if so requested
      ep.put(new Message(a, rx, k, offer))
    def tryFrom(cur: Cursor, failMode: Failure): Any = {
      cur.getNext match {
        case Some(msg, next) =>
          val merged =
            msg.senderK               // complete sender's continuation
            >> new SwapK(msg.payload, // then complete our continuation
                         msg.offer)
          merged.tryReact(a, rx ++ msg.senderRx, offer) match {
            case Retry => tryFrom(next, Retry)
            case Block => tryFrom(next, failMode)
            case ans   => ans
          }
        case None => failMode
      }
    }
    tryFrom(ep.dual.cursor, Block) // attempt reaction
  }

  // lift our continuation to a continuation for the dual sender
  class SwapK[S](dualPayload: B, dualOffer: Offer[S])
  extends Reagent[S,R] {
    def >>[T](next: Reagent[R,T]) = throw Impossible // unreachable
    def canFail = true
    def canSync = k.canSync

    def tryReact(s: S, rx: Reaction, myOffer: Offer[S]) = {
      k.tryReact(dualPayload, rx.withFulfill(dualOffer, s), myOffer)
    }
  }
}
```

Figure 11.3: The `Swap` class

The code we have given for `Swap` glosses over some corner cases that a full implementation must deal with. For example, it is possible for a reagent to attempt to `swap` on both sides of a channel, but it should avoid fulfilling its own offer in this case. Similarly, if a reagent `swaps` on the same channel multiple times, the implementation should avoid trying to consume the same message on that channel multiple times.

11.5.3 *Disjunction: choice*

The implementation of choice is pleasantly simple:

```
class Choice[A,B](r1: Reagent[A,B], r2: Reagent[A,B])
  extends Reagent[A,B] {
  def >>[C](next: Reagent[B,C]) =
    new Choice[A,C](r1 >> next, r2 >> next)
  def canFail = r1.canFail || r2.canFail
  def canSync = r1.canSync || r2.canSync

  def tryReact(a: A, rx: Reaction, offer: Offer[B]) =
    r1.tryReact(a, rx, offer) match {
      case Retry ⇒ r2.tryReact(a, rx, offer) match {
        case (_: Failure) ⇒ Retry // must retry r1
        case ans          ⇒ ans
      }
      case Block ⇒ r2.tryReact(a, rx, offer)
      case ans   ⇒ ans
    }
}
```

Choice attempts a reaction with either of its arms, trying them in left to right order. As explained in §10.2.3, a persistent failure of choice can only result from a persistent failure of *both* arms.[24] The right arm is tried even if the left arm has only failed transiently.

[24] The accumulation of the "`Retry`" signal here is reminiscent of the `retry` flag in §9.3.

11.5.4 *Conjunction: pairing and sequencing*

To implement the pairing combinator $*$, we first implement combinators `first` and `second` that lift reagents into product types; see Figure 11.4. These combinators are associated with *arrows*[25] in Haskell, and are useful for building up complex wiring diagrams.

[25] Hughes (2000), "Generalising monads to arrows"

```
def first[A,B,C] (r: Reagent[A,B]): Reagent[A × C, B × C] =
  new First(r, new Commit[B × C])
def second[A,B,C](r: Reagent[A,B]): Reagent[C × A, C × B] =
  new Second(r, new Commit[C × B])
```

```
class First[A,B,C,R](r: Reagent[A,B], k: Reagent[B × C,R])
extends Reagent[A × C,R] {
  def >>[S](next: Reagent[R,S]) = new First[A,B,C,S](r, k >> next)
  def canFail = r.canFail || k.canFail
  def canSync = r.canSync || k.canSync
  def tryReact(both: A × C, rx: Reaction, offer: Offer[R]) =
    (r >> Lift(b ⇒ (b, both._2)) >> k).tryReact(both._1, rx, offer)
}
// Second is defined symmetrically
```

Figure 11.4: Arrow-style lifting into product types

With them in hand, we can define a `pair` combinator[26] quite easily. The $*$ method on reagents is just an alias for the `pair` combinator, to support infix syntax.

[26] This combinator would be called &&&, or "fanout", in Haskell's arrow terminology.

```
def pair[A,B,C](r1: Reagent[A,B], r2: Reagent[A,C]): Reagent[A,B × C] =
  lift(a ⇒ (a, a)) >> first(r1) >> second(r2)
```

11.5.5 *Computational reagents*

The `lift` combinator, defined in Figure 11.5 by the `Lift` class, is the simplest reagent: it blocks when the function to lift is undefined, and otherwise applies the function and passes the result to its continuation.

```
class Lift[A,B,R](f: A ⇀ B, k: Reagent[B,R])
  extends Reagent[A,R] {
  def >>[S](next: Reagent[R,S]) = new Lift[A,B,S](c, k >> next)
  def canFail = k.canFail
  def canSync = k.canSync
  def tryReact(a: A, rx: Reaction, offer: Offer[R]) =
    if (f.isDefinedAt(a)) k.tryReact(f(a), rx, offer)
    else Block
}
```

Figure 11.5: The `Lift` class

The implementation of computed reagents (Figure 11.6) is exactly as described in §10.5: attempt to execute the stored computation `c` on the argument `a` to the reagent, and invoke the resulting reagent with a unit value. If `c` is not defined at `a`, the computed reagent issues a persistent failure. The implementation makes clear that the reads and writes performed within the computation `c` are invisible: they do not even have access to the `Reaction` object, and so they cannot enlarge the atomic update performed when it is committed.

```
class Computed[A,B,R](c: A ⇀ Reagent[Unit,B], k: Reagent[B,R])
  extends Reagent[A,R] {
  def >>[S](next: Reagent[R,S]) = new Computed[A,B,S](c, k >> next)
  def canFail = true  // must be conservative
  def canSync = true  // must be conservative
  def tryReact(a: A, rx: Reaction, offer: Offer[R]) =
    if (c.isDefinedAt(a)) (c(a) >> k).tryReact((), rx, offer)
    else Block
}
```

Figure 11.6: The `Computed` class

11.6 CATALYSIS

Thus far, our survey of the reagent implementation has focused wholly on reactants. What about catalysts?

It turns out that very little needs to be done to add support for the `dissolve` operation. Catalysts are introduced by invoking `tryReact` with an instance of a special subclass of `Offer`. This "catalyzing" subclass treats "fulfillment" as a no-op—and because it is never considered fulfilled, the catalyst is never used up. Because fulfillment is a no-op, multiple threads can react with the catalyst in parallel.

11.7 PERFORMANCE

11.7.1 *Methodology and benchmarks*

As we mentioned in Chapter 9, scalable concurrent data structures are usually evaluated by targetted microbenchmarking, with focus on contention effects and fine-grained parallel speedup.[27] In addition to those basic aims, we wish to evaluate (1) the extent to which reagent-based algorithms can compete with their hand-built counterparts and (2) whether reagent composition is a plausible approach for scalable atomic transfers.

To this end, we designed a series of benchmarks focusing on simple lock-free collections, where overhead from reagents is easy to gauge. Each benchmark consists of n threads running a loop, where in each iteration they apply one or more atomic operations on a shared data structure and then simulate a workload by spinning for a short time. For a high contention simulation, the spinning lasts for 0.25μs on average, while for a low contention simulation, we spin for 2.5μs.

In the "PushPop" benchmark, all of the threads alternate pushing and popping data to a single, shared stack. In the "StackTransfer" benchmark, there are two shared stacks, and each thread pushes to one stack, atomically transfers an element from that stack to the other stack, and then pops

[27] Mellor-Crummey and Scott 1991; Michael and Scott 1996; Herlihy, Luchangco, Moir, and W.N. N Scherer, III 2003; William N. Scherer, III and Scott 2004; Hendler *et al.* 2004; Fraser and Tim Harris 2007; Cederman and Tsigas 2010; Hendler *et al.* 2010

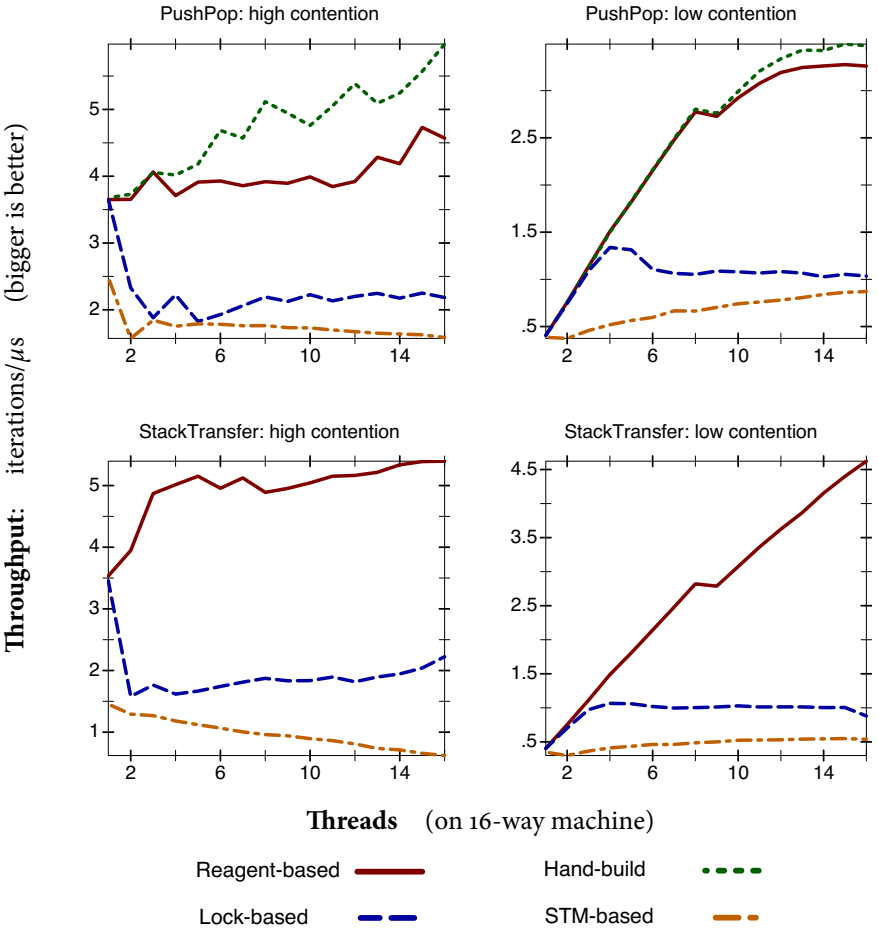


Figure 11.7: Benchmark results for stacks

an element from the second stack; the direction of movement is chosen randomly. The stack benchmarks compare our reagent-based `TreiberStack` to (1) a hand-built Treiber stack, (2) a mutable stack protected by a single lock, and (3) a stack using STM.

The "EnqDeq" and "QueueTransfer" benchmarks are analogous, but work with queues instead. The queue benchmarks compare our reagent-based `MSQueue` to (1) a hand-built Michael-Scott queue, (2) a mutable queue protected by a lock, and (3) a queue using STM.

For the transfer benchmarks, the hand-built data structures are dropped, since they do not support atomic transfer; for the lock-based data structures, we acquire both locks in a fixed order before performing the transfer.

We used the Multiverse STM, a sophisticated open-source implementation of Transaction Locking II[28] which is distributed as part of the Akka package for Scala. Our benchmarks were run on a 3.46Ghz Intel Xeon X5677 (Westmere) with 32GB RAM and 12MB of shared L3 cache. The machine has two physical processors with four hyperthreaded cores each, for a total of 16 hardware threads. L1 and L2 caches are per-core. The software environment includes Ubuntu 10.04.3 and the Hotspot JVM 6u27.

[28] Dave Dice *et al.* (2006), "Transactional locking II"

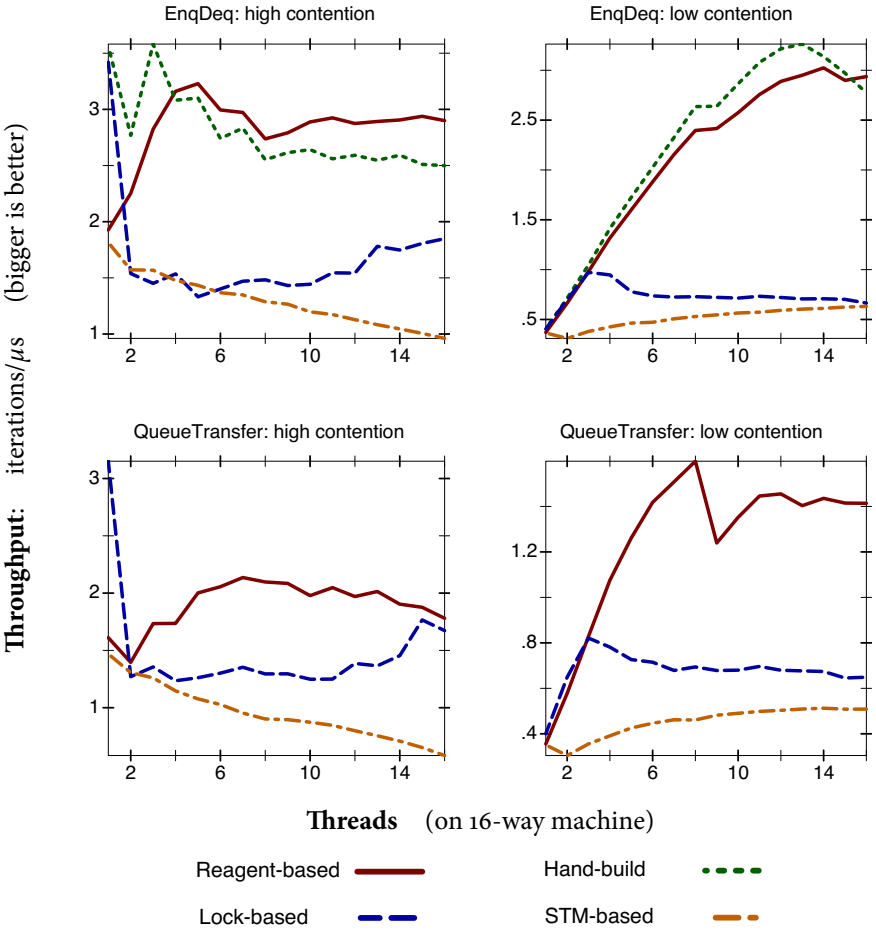


Figure 11.8: Benchmark results for queues

11.7.2 *Analysis*

The results are shown in Figures 11.7 and 11.8; the x-axes show thread counts, while the y-axes show throughput (so larger numbers are better). The results show that reagents can plausibly compete with hand-built concurrent data structures, while providing scalable composed operations that are rarely provided for such data structures.

Just as with our join pattern benchmarks (§9.8), we are mainly interested in the slopes of these graphs, which provide an indicator of scalability. One immediately noticeable feature of the slope on most of the graphs is a smaller incline (or even momentary plateau) at 8 threads, visible for the faster implementations but not so for the slower ones. The likely culprit is hyperthreading: the machine only has 8 actual cores, each of which attempts to share processing resources between two hardware threads. In addition, at 8 or more threads the pigeonhole principle kicks in: at least one thread is competing to be preemptively scheduled onto a single core with either another worker thread, or with OS services.

Aside from the changes at 8 threads, the benchmarks tell a consistent story: the reagent-based data structures perform universally better than the lock-

or STM-based data structures. They scale comparably with hand-built fine-grained data structures at low contention. For high contention, scalability is comparable for the queue, but for the stack reagents do not scale as well as the hand-written algorithm. Since the algorithm being used is identical for the reagent-based and hand-built stack, the reason for the difference must be due to the interpretive overhead in executing the reagent. Interpretive overhead occurs in our Scala implementation in the use of dynamic dispatch during reagent execution. While the HotSpot compiler is generally good at inlining or caching dynamic dispatched in hot loops, the functional programming idioms we use in our Scala code can sometimes thwart these optimizations.[29] On the other hand, as soon as the algorithm becomes more complex (*e.g.,* for the queue), these overheads are dwarfed by other costs.

[29] With the forthcoming introduction of lambda expressions for Java, the HotSpot JIT's ability to optimize functional programming idioms is likely to improve.

12

Related work: expressing concurrency

Join patterns and reagents are just two points in a large design space, both in terms of design and implementation strategy. This chapter briefly surveys the closest neighbors along three axes: composable concurrency constructs (§12.1), implementations of join patterns (§12.2), and scalable synchronization (§12.3).

"Don't worry about people stealing your ideas. If your ideas are any good, you'll have to ram them down people's throats."

—Howard Aiken

12.1 COMPOSABLE CONCURRENCY

12.1.1 *Concurrent ML*

12.1.1.1 *Design*

Concurrent ML[1] (CML) was designed to resolve an apparent tension between abstraction and choice: a synchronization protocol can be encapsulated as a function, but doing so makes it impossible to *e.g.,* take the external choice between two such protocols. The solution is *higher-order concurrency*, a code-as-data approach in which synchronous message-passing protocols are represented as *events*—an abstract data type. CML's events are built up from combinators, including a choice combinator, communication combinators, and combinators for arbitrary computations not involving communication.

[1] Reppy (1991), "CML: A higher concurrent language"

Reagents are clearly influenced by the design of CML's events, and include variants of CML's core event combinators (communication and choice). But where CML is aimed squarely at capturing synchronous communication protocols, reagents are designed for writing and tailoring fine-grained concurrent data structures and synchronization primitives. This difference in motivation led us to include a number of additional combinators for dealing with shared state and expressing join patterns.

12.1.1.2 *Implementation*

Our implementation of join patterns and reagents both draw some inspiration from Reppy, Russo and Xiao's Parallel CML (PCML), a scalable implementation of CML.[2] The difficulty in implementing CML is, in a sense, dual to that of the join calculus: disjunctions of events (rather than conjunctions of messages) must be resolved atomically. PCML implements disjunction (choice) by adding a single, *shared* event to the queue of each involved channel. Events have a "state" similar to our message statuses; event "resolution" is performed by an optimistic protocol that uses CAS to claim events.

[2] Reppy *et al.* (2009), "Parallel concurrent ML"

The PCML protocol is, however, much simpler than the protocol we have presented for join patterns: in PCML, events are resolved while holding a channel lock. In particular, if an event is offering a choice between sending on channel *A* and receiving on channel *B*, the resolution code will first lock channel *A* while looking for partners, then (if no partners are found) unlock *A* and lock channel *B*. These channel locks prevent concurrent changes to channel queues, allowing the implementation to avoid subtle questions about when it is safe to stop running the protocol—exactly the questions we address in Chapter 9. The tradeoff for this simplicity is, in principle, reduced scalability under high contention due to reduced concurrency. Although we have not performed a head-to-head comparison of PCML against a finer-grained implementation, our experimental results for the highly optimized lock-based joins suggest that locking will be a scalability bottleneck for contended channels in PCML.

12.1.2 *Software transactional memory*

Software transactional memory (STM) was originally intended "to provide a general highly concurrent method for translating sequential object implementations into non-blocking ones".[3] This ambitious goal has led to a remarkable research literature, which has been summarized in textbook form.[4] Much of the research is devoted to achieving scalability on multiprocessors or multicores, sometimes by relaxing consistency guarantees or only providing obstruction-freedom rather than lock-freedom.[5]

[3] Shavit and Touitou (1997), "Software transactional memory"

[4] Tim Harris *et al.* (2010), "Transactional Memory, 2nd edition"

[5] Herlihy, Luchangco, Moir, and W.N. N Scherer, III (2003), "Software transactional memory for dynamic-sized data structures"

12.1.2.1 *Join patterns and STM*

In some ways, the join calculus resembles STM: it allows a programmer to write a kind of "atomic section" of unbounded complexity. But where STM allows arbitrary shared-state computation to be declared atomic, the join calculus only permits highly-structured join patterns. By reducing expressiveness relative to STM, our joins library admits a relatively simple implementation with robust performance and scalability. It is not too hard to see the vestiges of an STM implementation in our joins algorithm, *e.g.,* in its use of a kind of undo log. But there are many aspects of the algorithm that take specific advantage of its restricted scope to go beyond a generic STM, *e.g.,* the retry/search strategy it uses, or its lazy message creation.

12.1.2.2 *Reagents and STM*

Both join patterns and STM provide fully "declarative" atomicity: the programmer simply asks that (respectively) certain channel interactions or certain state interactions be performed atomically—full stop. Reagents, by contrast, are aimed at a less ambitious goal: enabling the concise expression, user tailoring, and composition of scalable concurrent algorithms. That is, unlike STM, reagents do not attempt to provide a *universal* algorithm. In-

stead, they assist in writing and combining *specific* algorithms, carving out a middle ground between completely hand-written algorithms and completely automatic `atomic` blocks.

Consider, for example, implementing a concurrent queue:

- Using STM, one would simply wrap a sequential queue implementation in an `atomic` block, which requires no algorithmic insight or concurrency expertise. To implement a transfer from one queue to another, it again suffices to write some sequential code and wrap with `atomic`. Unfortunately, even with a very clever STM, such an implementation is unlikely to scale as well as *e.g.,* the Michael-Scott queue (§11.7).

- Using reagents, a concurrency expert could instead directly express an implementation like the Michael-Scott queue, which requires algorithmic insight but in return provides much greater scalability. Reagents provide a higher-than-usual level of abstraction for writing such implementations, but their main benefit is that nonexpert users can combine and tailor such implementations using additional reagent combinators. Thus reagents provide *some* of the composability and declarative nature of STM, while leaving room for experts to write specialized algorithms.

A key point is that, when used in isolation, reagents are *guaranteed* to perform only the CASes that the corresponding hand-written algorithm would.[6] Such a clear cost model is essential for maximizing scalability, and we know of no STM that provides similar guarantees. But there is an inherent tradeoff: the cost model depends on giving experts an "escape hatch" by which they can perform "invisible" read or write operations, but such invisible operations can render certain reagent compositions *unsafe*. The technical details are covered in §10.5, but the takeaway is that reagents trade some safety in return for the ability to write expert-level algorithms.

[6] See Chapter 11 for more detail.

Haskell's STM[7] demonstrated that transactions can be represented via monads,[8] explicitly composed, and combined with blocking and choice operators; its approach is in many respects reminiscent of CML. Reagents also form a monad, but we have chosen an interface closer to *arrows*,Hughes 2000 to encourage static reagent layout wherever possible (§10.5.1). Like `orElse` in Haskell's STM, our choice operator is left-biased. But unlike `orElse`, our choice operator will attempt the right-hand side even when the left-hand side has only failed transiently (rather than permanently).[9] While the distinction appears technical, it is crucial for examples like the elimination backoff stack (§10.2.3).

[7] Tim Harris *et al.* (2005), "Composable memory transactions"

[8] Peyton Jones and Wadler (1993), "Imperative functional programming"

[9] Note that `retry` in Haskell's STM signals a *permanent* failure, rather than an optimistic retry.

12.1.3 *Transactions that communicate*

A central tenet of transactions is *isolation*: transactions should not be aware of the concurrent execution of other transactions. But sometimes it is desirable for concurrent transactions to coordinate or otherwise communicate while

executing. Recent papers have proposed mechanisms for incorporating communication with STM, in the form of Communicating Transactions,[10] Transaction Communicators,[11] and Transactions with Isolation and Cooperation (TIC).[12] A key question in this line of work is how the expected isolation of shared memory can safely coexist with concurrent communication:

- Communicating Transactions use explicit, asynchronous message passing to communicate; the mechanism is entirely separate from shared memory, which retains isolation. When there is a conflict between isolation and interaction, the transaction is aborted and retried.
- On the other hand, Transaction Communicators and TIC allow shared memory isolation to be weakened in a controlled way.

[10] Lesani and Palsberg (2011), "Communicating memory transactions"

[11] Luchangco and Marathe (2011), "Transaction communicators: enabling cooperation among concurrent transactions"

[12] Smaragdakis *et al.* (2007), "Transactions with isolation and cooperation"

Our mixture of message-passing and shared-state combinators most closely resembles Communicating Transactions. Of course, the most important difference is that we do not build on top of an STM, but rather provide a lower-level programming interface as described above. We also believe that *synchronous* communication is better for expressing patterns like elimination (§10.2.3), which rely on mutual awareness between participants.

There has also been work treating pure message-passing in a transactional way. Transactional Events[13] combines CML with an atomic sequencing operator. Previously, Transactional Events were implemented on top of Haskell's STM, relied on search threads[14] for matching communications, and used an STM-based representation of channels. However, Transactional Events are expressible using reagents, through the combination of `swap` and the conjunction combinators. Doing so yields a new implementation that does not require search threads, performs parallel matching for communication, and represents channels as lock-free bags. We are not in a position to do a head-to-head comparison, but based on the results in §11.7, we expect the reagent-based implementation to scale better on fine-grained workloads.

[13] Donnelly and Fluet (2008), "Transactional events"

[14] The implementation can be made to work with a single search thread at the cost of lost parallelism.

Of course, the join calculus is another example of message passing with a transactional flavor.

12.1.4 *Composing scalable concurrent data structures*

Most of the literature on scalable concurrent data structures is focused on "within-object" atomicity, for example developing algorithms for inserting or removing elements into a collection atomically. Recently, though, Cederman and Tsigas (2010) proposed a method for deriving atomic transfer operations between lock-free data structures. The basic approach relies on a *k*CAS operation in much the same way that reagent sequencing does. However, the transfer methods must be written manually, in advance, and with access to the internals of the relevant data structures. Reagents, by contrast, allow *clients* to define arbitrary new compositions, without manually implementing them, and without access to the code or internals of the involved data structures. Nevertheless, reagent sequencing yields an algorithm very similar to the

manually-written transfer methods. On the other hand, Cederman and Tsigas (2010) provide a generic proof of correctness for their transfer methodology, while we have provided no such proof for reagent composition. Chapter 13 discusses possible avenues for doing so.

It is also possible to go in the other direction: start from STM, which provides composition, and add an "escape hatch" for writing arbitrary scalable concurrent algorithms within the scope of a transaction. The escape hatch can be provided through unlogged reads and/or writes to memory locations being used by transactions, as in early release[15] or elastic transactions.[16] As we discussed above (§12.1.2.2), we favor an approach where the focus is foremost on writing scalable algorithms, with *guarantees* about the performance and shared-memory semantics of those algorithms. Providing such guarantees via an escape hatch mechanism may be difficult or impossible, depending on the details of the STM implementation. As we showed in §10.2, it is also very useful to have combinators for choice, message-passing, and blocking, if one wishes to capture the full range of scalable concurrent algorithms.

[15] Herlihy, Luchangco, Moir, and W.N. N Scherer, III (2003), "Software transactional memory for dynamic-sized data structures"

[16] Felber *et al.* (2009), "Elastic transactions"

12.2 JOIN CALCULUS IMPLEMENTATIONS

Fournet and Gonthier originally proposed the join calculus as an asynchronous process algebra designed for efficient implementation in a distributed setting.[17] It was positioned as a more practical alternative to Milner's π-calculus.

[17] Fournet and Gonthier (1996); Fournet and Gonthier (2002)

12.2.1 *Lock-based implementations*

The join calculus has been implemented many times, and in many contexts. The earliest implementations include Fournet *et al.*'s JoCaml[18] and Odersky's Funnel[19] (the precursor to Scala), which are both functional languages supporting declarative join patterns. JoCaml's runtime is single-threaded so the constructs were promoted for concurrency control, not parallelism. Though it is possible to run several communicating JoCaml *processes* in parallel, pattern matching will always be sequential. Funnel targeted the Java VM, which can exploit parallelism, but we could find no evaluation of its performance on parallel hardware. Benton *et al.* (2004) proposed an object-oriented version of join patterns for C♯ called Polyphonic C♯; around the same time, Von Itzstein and Kearney (2001) independently described JoinJava, a similar extension of Java. The advent of generics in C♯ 2.0 led Russo to encapsulate join pattern constructs in the Joins library,[20] which served as the basis for our library. There are also implementations for Erlang, C++, and VB.[21]

All of the above implementations use coarse-grained locking to achieve the atomicity present in the join calculus semantics. In some cases (*e.g.* Polyphonic C♯, Russo's library) significant effort is made to minimize the

[18] Fournet *et al.* (2003), "JoCaml: A Language for Concurrent Distributed and Mobile Programming"

[19] Odersky (2002), "An Overview of Functional Nets"

[20] Russo (2007), "The Joins Concurrency Library"

[21] Plociniczak and Eisenbach (2010); Liu (2009); Russo (2008), respectively.

critical section, but as we have shown (§9.8) coarse-grained locking remains an impediment to scalability.

12.2.2 *STM-based implementations*

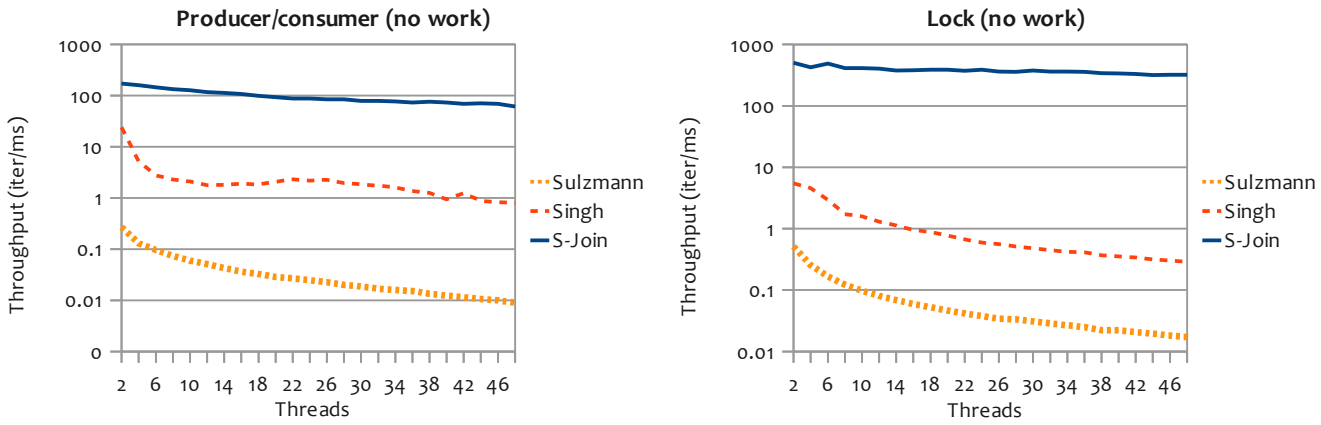


Figure 12.1: Comparison with Haskell-STM implementations on 48-core machine. **Note log scale**.

We are aware of two join-calculus implementations that do not employ a coarse-grained locking strategy, instead using Haskell's STM. Singh's implementation builds directly on the STM library, using transacted channels and atomic blocks to provide atomicity;[22] the goal is to provide a simple implementation, and no performance results are given. In unpublished work, Sulzmann and Lam suggest a *hybrid* approach, saying that "an entirely STM-based implementation suffers from poor performance".[23] Their hybrid approach uses a nonblocking collection to store messages, and then relies on STM for the analog to our message resolution process. In addition to basic join patterns, Sulzmann and Lam allow *guards* and *propagated clauses* in patterns, and to handle these features they spawn a thread *per message*; Haskell threads are lightweight enough to make such an approach viable. The manuscript provides some performance data, but only on a four core machine, and does not provide comparisons against direct solutions to the problems they consider.

The simplest—but perhaps most important—advantage of our implementation over STM-based implementations is that we do not require STM, making our approach more portable. STM is an active area of research, and state-of-the-art implementations require significant effort.

The other advantage over STM is the specificity of our algorithm. An STM implementation must provide a general mechanism for declarative atomicity, conflict-resolution and contention management. Since we are attacking a more constrained problem, we can employ a more specialized (and likely more efficient and scalable) solution. For example, in our implementation one thread can be traversing a bag looking for `PENDING` messages, determine

[22] Singh (2006), "Higher-Order combinators for join patterns using STM"

[23] Sulzmann and Lam (2009), "Parallel Join Patterns with Guards and Propagation"

that none are available, and exit message resolution all while another thread is adding a new `PENDING` message to the bag. Or worse: two threads might both add messages to the same bag. It is not clear how to achieve the same degree of concurrency with STM: depending on the implementation, such transactions would probably be considered conflicting, and one aborted and retried. While such spurious retries might be avoidable by making STM aware of the semantics of bags, or by carefully designing the data structures to play well with the STM implementation, the effort involved is likely to exceed that of the relatively simple algorithm we have presented.

To test our suspicions about the STM-based join implementations, we replicated the pure synchronization benchmarks for producer/consumer and locks, on top of both Singh and Sulzmann's implementations. Figure 12.1 gives the results on the same 48-core benchmarking machine we used for §9.8. Note that, to avoid losing too much detail, we plot the throughput in these graphs using a log scale. *The comparison is, of course, a loose one*: we are comparing across two very different languages and runtime systems, and Sulzmann's implementation provides more than just join patterns. However, it seems clear that the STM implementations suffer from both drastically increased constant overheads, as well as much poorer scalability. Surprisingly, of the two STM implementations, Singh's much simpler implementation was the better performer.

Given these results, and the earlier results for lock-based implementations, our Joins implementation is the only one we know to scale when used for fine-grained parallelism.

12.2.3 *Languages versus libraries*

By implementing joins as a library, we forgo some expressivity. JoCaml, for example, supports restricted polymorphic sends: the type of a channel can be generalized in those type variables that do not appear in the types of other, conjoined channels.[24] Since our channels are monomorphic C♯ delegates, we are, unfortunately, unable to provide that level of polymorphism. Nevertheless, one can still express a wide range of useful generic abstractions (e.g. `Buffer<T>`, `Swap<A,B>`). Another difference is that our rendezvous patterns are more restrictive than JoCaml's. Our implementation only allows us to return a single value to all synchronous channels, instead of returning separately typed values to each synchronous channel. In effect, we strike a compromise between the power of JoCaml and limitations of Polyphonic C♯ (which allowed at most one synchronous channel per pattern). As a consequence, our coding of swap channels is clumsier than JoCaml's, requiring wrapper methods to extract the relevant half of the common return value. JoCaml instead supports (the equivalent of) selective return statements, allowing one to write, *e.g.*,

[24] Fournet *et al.* (1997), "Implicit Typing à la ML for the Join-Calculus"

```
return b to Left; return a to Right;
```

within the same chord. The static semantics of selective returns are difficult to capture in a library, so we have avoided them. Note that forcing all channels to wait on a single return statement, as we do, also sacrifices some concurrency.

12.3 SCALABLE SYNCHRONIZATION

12.3.1 *Coordination in java.util.concurrent*

The java.util.concurrent library contains a class called AbstractQueuedSynchronizer that provides basic functionality for queue-based, blocking synchronizers.[25] Internally, it represents the state of the synchronizer as a single 32-bit integer, and requires subclasses to implement `tryAcquire` and `tryRelease` methods in terms of atomic operations on that integer. It is used as the base class for at least six synchronizers in the java.util.concurrent package, thereby avoiding substantial code duplication. In a sense, our Joins library is a generalization of the abstract synchronizer framework: we support arbitrary internal state (represented by asynchronous messages), n-way rendezvous, and the exchange of messages at the time of synchronization. Reagents then generalize further by incorporating updates to shared memory.

[25] Lea (2005), "The java.util.concurrent synchronizer framework"

12.3.2 *Dual data structures*

Another interesting aspect of java.util.concurrent is its use of *dual data structures*,[26] in which blocking calls to a data structure (such as `Pop` on an empty stack) insert a "reservation" in a nonblocking manner; they can then spinwait to see whether that reservation is quickly fulfilled, and otherwise block. Reservations provide an analog to the *conditions* used in monitors, but apply to nonblocking data structures.

[26] William N. Scherer, III and Scott (2004), "Nonblocking Concurrent Data Structures with Condition Synchronization"

Both join patterns and reagents offer alternative perspectives on reservations:

- WITH JOIN PATTERNS, one generally treats methods (like `Pop`) as synchronous channels. Calling a method is then tantamount to sending a message. But with lazy message creation (§9.5.1), the caller will first attempt to find some current enabled pattern and immediately fire it—just as, with a dual data structure, a caller first attempts to perform the operation normally. Only if no pattern is enabled (*i.e.,* the method call should block) is a message actually created and added to a bag—just as, with a dual data structure, a reservation would be created and enqueued. So something like dual data structures "fall out" as a natural consequence of the joins implementation, including not just reservations but the spinwaiting and blocking strategies as well.

- REAGENTS LIKEWISE naturally support dual data structures as coded through join patterns, since the reagent implementation includes lazy message creation. But the shared state combinators provide an alternative

avenue for blocking. When a reagent fails permanently in a way that could depend on the value read from a reference, the blocking protocol will automatically add an appropriate continuation to a bag of waiting reagents associated with the reference.

Neither the joins- nor reagents-based blocking reservations work *exactly* the same way as those in hand-built dual data structures, which take advantage of specific representation details to store either real data or reservations, but never both. On the other hand, joins and reagents provide a more general and automatic technique for adding blocking to a scalable data structure.

Part IV

EPILOGUE

13

Conclusion

13.1 LOOKING BACK

"I may not have gone where I intended to go, but I think I have ended up where I needed to be."

—Douglas Adams

This dissertation demonstrates two claims:

A. Scalable algorithms can be *understood* through linked protocols governing each part of their state, which enables verification that is local in space, time, and thread execution.

B. Scalable algorithms can be *expressed* through a mixture of shared-state and message-passing combinators, which enables extension by clients without imposing prohibitive overhead.

The support for these claims takes the form of three distinct research artifacts:

- A LOGIC for local reasoning about scalable concurrency using standalone, visual protocols. The approach enables direct refinement proofs via Hoare-style reasoning, and scales to high-level languages (with higher-order functions and recursive and abstract types) and sophisticated algorithms (employing role-playing, cooperation and internal nondeterminism).
- A LIBRARY for declarative and scalable synchronization based on the join calculus. Users of the library can write down arbitrary synchronization constraints as join patterns (roughly, disjunctions of conjunctions), and the library will automatically derive a reasonably scalable solution to them.
- AN ABSTRACTION—reagents—for expressing and composing scalable concurrent algorithms and data structures through a blend of message-passing and shared-state primitives. Reagents serve the needs of two distinct groups: concurrency experts and concurrency users. Using reagents, experts can write libraries more easily, because common patterns are expressible as abstractions and many are built-in. Users can then extend, tailor and compose the resulting library without detailed knowledge of the algorithms involved.

Taken together, these contributions make a significant step forward in our ability to understand and express scalable concurrency. But much remains to be done.

13.2 LOOKING AHEAD

13.2.1 *Understanding scalable concurrency*

For the sake of simplicity, the logic of Chapter 5 only goes part of the way to a full-fledged "logic for logical relations": it is not powerful enough to *define* our logical relation, and so instead treats it as a built-in assertion. In particular, the assertions of the logic do not allow second-order quantification or recursion. By adding these constructions, we could treat both value- and expression-level refinement as *sugar* for more basic logical formulas, in the spirit of Plotkin and Abadi (1993).

Likewise, while we have given a substantial sketch of a proof theory for the logic, there are several gaps—most importantly in our treatment of spec resources and their interaction with speculation, which we reason about entirely "in the model." There is a duality between implementation reasoning (in which one must consider *all* executions) and specification reasoning (in which one must discover *some* execution), elegantly expressed in the two modalities of *dynamic logic*.[1] It would be interesting to reformulate our atomic Hoare triples along the lines of dynamic logic, allowing them to be used for spec reasoning as well—and to study the interaction with speculation as well.

[1] Harel *et al.* (2000), "Dynamic logic"

One somewhat irritating aspect of the refinement approach, at least as we have presented it, is that specifications tend to be overly concrete. Our "canonical atomic specs" (Chapter 3) use a physical lock to guarantee mutual exclusion, and our specs for data structures like queues (Chapter **??**) must concretely represent the current value of the queue. Ideally, specs would be given even more abstractly, in terms of some "atomic" keyword and arbitrary "mathematical" data types. However, as we discussed in §3.4, an "atomic" keyword is problematic: it gives too much power to the context. So we leave as an open question how to write more abstract atomic specs in a higher-order refinement framework.

There are at least two interesting questions about the *completeness* of our logic. First, is it complete for the traditional scope of linearizability, *i.e.,* first-order modules that share no state with their clients? Second, are local protocols "complete" for Views,[2] *i.e.,* can one express any monoid instantiation of the Views framework in terms of an STS with tokens?

[2] Dinsdale-Young *et al.* (2013), "Views: Compositional Reasoning for Concurrent Programs"

Despite the fact that our logic scales to higher-order programs, we have only explored very simple uses of this capability, namely, in reasoning about simple "modules" which export a tuple of functions. It remains to be seen how suitable the logic is for reasoning about more fundamentally higher-order code like Herlihy's universal construction[3] or the flat combining algorithm.[4] Likewise, we have focused on atomic specifications, but many data structures provide more complex forms of interaction (*e.g.,* iteration in a concurrent collection) or weaker guarantees (*e.g.,* RCU-based algorithms in the Linux kernel). Can we give these data structures clean specs and correctness proofs?

[3] Herlihy and Shavit (2008), "The Art of Multiprocessor Programming"

[4] Hendler *et al.* (2010), "Flat combining and the synchronization-parallelism tradeoff"

Finally, there are a number of extensions of our logic to explore:

- Can the logic be extended to reason about liveness as well as safety?[5] It is unclear to what extent a step-indexed model such as ours can capture liveness properties; presumably, at the very least, one would need to work with a more complex ordinal than ω.
- Is it possible, moreover, to reason *thread-locally* about liveness? Traditionally liveness properties are proved using a global, well-founded measure, but nonblocking progress properties (which do not rely on fair scheduling) are likely amenable to protocol-based, thread-local reasoning.
- We have assumed sequential consistency, which is not unreasonable (see §3.2), but in the long run will prevent application of the logic to algorithms that use reference cells with weaker guarantees. A major open question is whether something like our local protocols—which rely on a *global* notion of "current state"—can be adapted to work in the context of a weak memory model.

[5] See §2.5.

13.2.2 *Expressing scalable concurrency*

There is significant remaining work for elucidating both the theory and practice of reagents:

- On the theoretical side, developing a formal operational semantics would help to clarify the interactions possible between shared state and message passing, as well as the atomicity guarantees that reagents provide. A reasonable starting point would be the operational semantics for Haskell's STM.[6]
- On the practical side, developing a serious concurrency library using reagents would go a long way toward demonstrating their usability. We have begun work along these lines by building an implementation of reagents, called Caper,[7] for the Racket programming language. Racket's runtime system is undergoing an incremental parallelization process,[8] and currently the language is equipped with very few scalable concurrent data structures. Since Racket also includes a sophisticated macro system, we plan to explore compiler support for reagents in Caper.

[6] Tim Harris *et al.* (2005), "Composable memory transactions"

[7] Concurrent and parallel extensions to Racket.

[8] Swaine *et al.* (2010), "Back to the futures: incremental parallelization of existing sequential runtime systems"

Beyond these immediate steps, a major open question for reagents (and similar composable concurrency abstractions) is how to integrate *lock-based* algorithms. Many scalable concurrent algorithms use fine-grained locking, and are thereby outside of the current scope of reagents. The key problem is that, in a monadic setting, it is not possible for the reagent library to know or enforce lock ordering up front. Ideally, reagents would allow free composition of lock-based and lock-free algorithms, thereby enabling a gradual transition from one to the other in the spirit of "script to program evolution."[9]

[9] Tobin-Hochstadt (2010), "Typed Scheme: From Scripts to Programs"

13.2.3 *Crossing the streams*

Ultimately, the two parts of this dissertation should have more to say to each other:

A. It should be possible to use our logic to prove the correctness of our join pattern and reagents implementations.

B. It should also be possible to use something like our logic of local protocols to reason about reagent-based algorithms. This reasoning should be abstract, in that it should use only some high-level specification of the reagent API without divulging its implementation details.

Item A is already feasible for the joins library, although giving a modular proof would require doing some of the work outlined above, *e.g.,* cleanly specifying concurrent iterators. Applying it to reagents would entail, at the very least, giving an operational semantics for the reagent API.

Item B would likely require some new ideas in the logic, in order to specify and perform client-side reasoning about monadic APIs at an appropriate level of abstraction.

In the long run, of course, items A and B should plug together: it should be possible to compose a correctness proof of the reagents library implementation with a correctness proof of a reagent-based algorithm.

References

Abadi, Martín and Leslie Lamport (1991).
The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284 (cited on pages 60, 75, 138).

Abadi, Martín and Leslie Lamport (1993).
Composing specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(1):73–132 (cited on page 82).

Abadi, Martín and Leslie Lamport (1995).
Conjoining specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(3):507–535 (cited on page 82).

Abelson, Harold and Gerald Jay Sussman (1996).
Structure and Interpretation of Computer Programs. MIT Press. URL: http://mitpress.mit.edu/sicp/ (cited on pages 17, 21).

Abramsky, Samson (1990).
The lazy lambda calculus. *Research topics in functional programming*, pages 65–116. URL: http://moscova.inria.fr/~levy/courses/X/M1/lambda/bib/90abramskylazy.pdf (cited on page 57).

Adve, Sarita V. and Kourosh Gharachorloo (1996).
Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76 (cited on page 26).

Agarwal, A and M Cherian (1989).
Adaptive backoff synchronization techniques. *In proceedings of the International Symposium on Computer Architecture (ISCA)*. New York, New York, USA: ACM Press, pages 396–406 (cited on page 36).

Ahmed, Amal (2004).
Semantics of Types for Mutable State. PhD thesis. Princeton University. URL: http://www.cs.indiana.edu/~amal/ahmedsthesis.pdf (cited on page 82).

Ahmed, Amal (2006).
Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. *In proceedings of the European Symposium on Programming (ESOP)*. Springer, pages 69–83 (cited on pages 47, 57).

Ahmed, Amal, Derek Dreyer, and Andreas Rossberg (2009).
State-dependent representation independence. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press (cited on pages 57, 128, 129).

Alpern, Bowen and Fred B. Schneider (1985).
Defining liveness. *Information Processing Letters*, 21(4):181–185 (cited on page 39).

Amdahl, Gene M. (1967).
Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*. New York, New York, USA: ACM Press, pages 483–485 (cited on pages 32, 33).

Appel, Andrew W. and David McAllester (2001).
An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5):657–683 (cited on page 82).

Appel, Andrew W., Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon (2007).
A very modal model of a modern, major, general type system. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. New York, New York, USA: ACM Press, pages 109–122 (cited on pages 90, 129).

Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali (1989).
I-structures: data structures for parallel computing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):598–632 (cited on page 18).

Attiya, Hagit, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev (2011).
Laws of order. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. New York, New York, USA: ACM Press, pages 487–498 (cited on page 13).

Attiya, Hagit and Eshcar Hillel (2008).
Highly-concurrent multi-word synchronization. *In proceedings of Distributed Computing and Networking (ICDCN)*. Springer Berlin Heidelberg, pages 112–123 (cited on page 197).

Batty, Mark, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber (2011).
Mathematizing C++ concurrency. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. New York, New York, USA: ACM Press, pages 55–66 (cited on page 12).

Benton, Nick (2003).
Jingle Bells: Solving the Santa Claus Problem in Polyphonic C#. URL: `http://research.microsoft.com/~nick/santa.pdf` (cited on page 148).

Benton, Nick, Luca Cardelli, and Cédric Fournet (2004).
Modern concurrency abstractions for C#. *ACM Transactions on Program-*

ming Languages and Systems (TOPLAS), 26(5):769–804 (cited on pages 142, 143, 147, 149, 150, 160, 161, 174, 213).

Berry, Gérard and Gérard Boudol (1992).
The chemical abstract machine. *Theoretical computer science*, 96(1):217–248 (cited on page 177).

Birkedal, Lars, Filip Sieczkowski, and Jacob Thamsborg (2012).
A concurrent logical relation. *In proceedings of Computer Science Logic (CSL)*. URL: http://itu.dk/people/fisi/pub/relconc_conf.pdf (cited on pages 55, 129).

Blelloch, Guy E. (1996).
Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97 (cited on page 12).

Blelloch, Guy E., Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun (2012).
Internally deterministic parallel algorithms can be fast. *In proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. New York, New York, USA: ACM Press, pages 181–192 (cited on page 13).

Bloom, Bard, Sorin Istrail, and Albert R Meyer (1995).
Bisimulation can't be traced. *Journal of the ACM*, 42(1):232–268 (cited on page 138).

Blumofe, Robert D., Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou (1996).
Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69 (cited on page 13).

Blundell, Colin, E. Christopher Lewis, and Milo M. K. Martin (2006).
Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters*, 5(2):17 (cited on page 55).

Boehm, Hans J. and Sarita V. Adve (2008).
Foundations of the C++ concurrency memory model. *In proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. New York, New York, USA: ACM Press, pages 68–78 (cited on page 12).

Brinch Hansen, Per (1973).
Operating system principles. Prentice Hall. URL: http://dl.acm.org/citation.cfm?id=540365 (cited on pages 11, 23, 31).

Brinch Hansen, Per (2001).
The invention of concurrent programming. *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*. Springer-Verlag.

URL: http://oberon2005.oberoncore.ru/paper/bh2002.pdf (cited on page 176).

Brookes, Stephen (1996).
Full Abstraction for a Shared-Variable Parallel Language. *Information and Computation*, 127(2):145–163 (cited on page 131).

Brookes, Stephen (2002).
Traces, Pomsets, Fairness and Full Abstraction for Communicating Processes. *In proceedings of Concurrency Theory (CONCUR)*, pages 466–482 (cited on page 15).

Brooks, Jr., Frederick P. (1987).
No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19 (cited on page 15).

Calcagno, Cristiano, Peter W. O'Hearn, and Hongseok Yang (2007).
Local Action and Abstract Separation Logic. *In proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, pages 366–378 (cited on page 114).

Cederman, Daniel and Philippas Tsigas (2010).
Supporting lock-free composition of concurrent data objects. *In proceedings of the ACM International Conference on Computing Frontiers (CF)*, pages 53–62 (cited on pages 167, 204, 212, 213).

Dice, Dave, Ori Shalev, and Nir Shavit (2006).
Transactional locking II. *In proceedings of Distributed Computing (DISC)*. Springer, pages 194–208 (cited on page 205).

Dice, David and Oleksandr Otenko (2011).
Brief announcement: MultiLane - a concurrent blocking multiset. *In proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. New York, New York, USA: ACM Press, pages 313–314 (cited on page 150).

Dijkstra, Edsger W.
EWD952: Science fiction and science reality in computing. URL: http://www.cs.utexas.edu/~EWD/transcriptions/EWD09xx/EWD952.html (cited on page 149).

Dijkstra, Edsger W. (1965).
EWD123: Cooperating Sequential Processes. Technical report. URL: http://www.cs.utexas.edu/~EWD/transcriptions/EWD01xx/EWD123.html (cited on pages 23, 30, 145).

Dijkstra, Edsger W. (1971).
Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138 (cited on page 141).

Dijkstra, Edsger W. (1976).
A Discipline of Programming. Prentice Hall (cited on page 101).

Dijkstra, Edsger W. (2000).
EWD 1303: My recollections of operating system design. URL: http://www.cs.utexas.edu/~EWD/transcriptions/EWD13xx/EWD1303.html (cited on page 11).

Dinsdale-Young, Thomas, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang (2013).
Views: Compositional Reasoning for Concurrent Programs. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (cited on pages 133, 222).

Dinsdale-Young, Thomas, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis (2010).
Concurrent Abstract Predicates. *ECOOP*. Springer, pages 504–528. URL: http://www.springerlink.com/index/184241T463712776.pdf (cited on pages 62, 132).

Dodds, Mike, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis (2009).
Deny-guarantee reasoning. *In proceedings of the European Symposium on Programming (ESOP)*. 736. Springer, pages 363–377 (cited on page 134).

Donnelly, Kevin and Matthew Fluet (2008).
Transactional events. *Journal of Functional Programming (JFP)*, 18(5 & 6):649–706 (cited on page 212).

Drepper, Ulrich (2007).
What every programmer should know about memory. URL: http://ftp.linux.org.ua/pub/docs/developer/general/cpumemory.pdf (cited on pages 26, 29).

Dreyer, Derek, Amal Ahmed, and Lars Birkedal (2009).
Logical Step-Indexed Logical Relations. *Logical Methods in Computer Science*, 7(2:16):71–80 (cited on pages 90, 129).

Dreyer, Derek, Georg Neis, and Lars Birkedal (2010).
The impact of higher-order state and control effects on local relational reasoning. *In proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*. New York, New York, USA: ACM Press, pages 143–156 (cited on pages 57, 60–62, 66, 82, 85, 128, 130).

Dreyer, Derek, Georg Neis, and Lars Birkedal (2012).
The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming (JFP)*, 22(4-5):477–528 (cited on page 129).

Dreyer, Derek, Georg Neis, Andreas Rossberg, and Lars Birkedal (2010). A relational modal logic for higher-order stateful ADTs. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. New York, New York, USA: ACM Press (cited on pages 79, 82, 90, 129).

Elmas, Tayfun, Shaz Qadeer, Ali Sezgin, Omer Subasi, and Serdar Tasiran (2010). Simplifying Linearizability Proofs with Reduction and Abstraction. *TACAS*. Springer, pages 296–311 (cited on page 136).

Elmas, Tayfun, Shaz Qadeer, and Serdar Tasiran (2009). A calculus of atomic actions. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (cited on page 136).

Felber, Pascal, Vincent Gramoli, and R. Guerraoui (2009). Elastic transactions. *In proceedings of Distributed Computing (DISC)* (cited on page 213).

Felleisen, Matthias (1991). On the expressive power of programming languages. *Science of Computer Programming*, 17(1-3):35–75 (cited on page 14).

Feng, Xinyu (2009). Local rely-guarantee reasoning. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, pages 315–327 (cited on page 134).

Feng, Xinyu, Rodrigo Ferreira, and Zhong Shao (2007). On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. *In proceedings of the European Symposium on Programming (ESOP)* (cited on page 134).

Filipović, Ivana, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang (2010). Abstraction for Concurrent Objects. *Theoretical Computer Science*, 411(51-52):4379–4398 (cited on pages 56, 130).

Fournet, Cédric and Georges Gonthier (1996). The reflexive CHAM and the join-calculus. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 372–385 (cited on pages 6, 141, 142, 150, 165, 186, 213).

Fournet, Cédric and Georges Gonthier (2002). The Join Calculus: A Language for Distributed Mobile Programming. *International Summer School on Applied Semantics (APPSEM)*. LNCS. Springer-Verlag, pages 268–332 (cited on pages 142, 186, 213).

Fournet, Cédric, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt (2003). JoCaml: A Language for Concurrent Distributed and Mobile Programming. *International School on Advanced Functional Programming (AFP)*. LNCS (cited on pages 160, 213).

Fournet, Cédric, Luc Maranget, Cosimo Laneve, and Didier Rémy (1997). Implicit Typing à la ML for the Join-Calculus. *In proceedings of Concurrency Theory (CONCUR)* (cited on page 215).

Fraser, Keir and Tim Harris (2007). Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2) (cited on pages 63, 120, 167, 197, 204).

Glabbeek, R J Van (1990). The linear time-branching time spectrum. *CONCUR'90 Theories of Concurrency: Unification and Extension*():278–297 (cited on page 137).

Gotsman, Alexey and Hongseok Yang (2012). Linearizability with Ownership Transfer. *In proceedings of Concurrency Theory (CONCUR)* (cited on page 130).

Groves, Lindsay and Robert Colvin (2009). Trace-based derivation of a scalable lock-free stack algorithm. *Formal Aspects of Computing*, 21(1-2):187–223 (cited on page 135).

Harel, David, Dexter Kozen, and Jerzy Tiuryn (2000). *Dynamic logic*. MIT Press. URL: http://mitpress.mit.edu/books/dynamic-logic (cited on page 222).

Harper, Robert (2011). Parallelism is not concurrency. URL: http://existentialtype.wordpress.com/2011/03/17/parallelism-is-not-concurrency/ (cited on pages 11, 12, 18).

Harris, Tim, James Larus, and Ravi Rajwar (2010). *Transactional Memory, 2nd edition*. Morgan and Claypool (cited on pages 193, 210).

Harris, Tim, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy (2005). Composable memory transactions. *In proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, pages 48–60 (cited on pages 177, 181, 211, 223).

Harris, Timo, Keir Fraser, and Ian A. Pratt (2002). A practical multi-word compare-and-swap operation. *In proceedings of Distributed Computing (DISC)*, pages 265–279. URL: http://www.cl.cam.ac.uk/research/srg/netos/papers/2002-casn.pdf (cited on page 120).

Heller, Steve, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, III, and Nir Shavit (2006).
A lazy concurrent list-based set algorithm. *In proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. Springer, pages 3–16 (cited on page 36).

Hendler, Danny, Itai Incze, Nir Shavit, and Moran Tzafrir (2010).
Flat combining and the synchronization-parallelism tradeoff. *In proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364 (cited on pages 42, 45, 130, 167, 204, 222).

Hendler, Danny, Nir Shavit, and Lena Yerushalmi (2004).
A scalable lock-free stack algorithm. *In proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. New York, New York, USA: ACM Press, pages 206–215 (cited on pages 37, 71, 167, 204).

Herlihy, Maurice (1991).
Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149 (cited on pages 30, 45).

Herlihy, Maurice, Victor Luchangco, and Mark Moir (2003).
Obstruction-free synchronization: double-ended queues as an example. *In proceedings of the International Conference on Distributed Computing Systems (ICDCS)*. IEEE, pages 522–529 (cited on page 42).

Herlihy, Maurice, Victor Luchangco, Mark Moir, and W.N. N Scherer, III (2003).
Software transactional memory for dynamic-sized data structures. *In proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)* (cited on pages 167, 204, 210, 213).

Herlihy, Maurice and Nir Shavit (2008).
The Art of Multiprocessor Programming. Morgan Kaufmann (cited on pages 13, 24, 32, 40, 41, 130, 146, 189, 222).

Herlihy, Maurice and Jeannette M. Wing (1990).
Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492 (cited on pages 40, 75, 130, 138).

Hoare, C.A.R. (1972).
Proof of correctness of data representations. *Acta {I}nformatica*, 1(4):271–281. URL: http://www.springerlink.com/index/W7446683830J348H.pdf (cited on page 60).

Hughes, John (2000).
Generalising monads to arrows. *Science of computer programming*, 37(1-3):67–111 (cited on pages 175, 188, 202, 211).

Hur, Chung-Kil, Derek Dreyer, Georg Neis, and Viktor Vafeiadis (2012).
The marriage of bisimulations and Kripke logical relations. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. New York, New York, USA: ACM Press, pages 59–72 (cited on page 86).

Jensen, Jonas Braband and Lars Birkedal (2012).
Fictional Separation Logic. *In proceedings of the European Symposium on Programming (ESOP)* (cited on page 133).

Jerger, Natalie D. Enright (2008).
Chip Multiprocessor Coherence and Interconnect System Design. PhD thesis. University of Wisconsin-Madison (cited on page 26).

Jones, Cliff B. (1983).
Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619 (cited on pages 61, 134).

Kahn, Gilles (1974).
The semantics of a simple language for parallel programming. *Information processing*, pages 471–475. URL: http://www.citeulike.org/group/872/article/349829 (cited on page 18).

Kennedy, Andrew and Claudio V. Russo (2005).
Generalized algebraic data types and object-oriented programming. *In proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*. New York, New York, USA: ACM Press, pages 21–40 (cited on page 153).

Knuth, Donald E. (1977).
Notes on the van Emde Boas construction of priority deques: An instructive use of recursion (cited on page 101).

Knuth, Donald E. (1997).
The Art of Computer Programming, Volume 1: Fundamental Algorithms. Addison Wesley (cited on page 31).

Knuth, Donald E. (2003).
Bottom-up education. *Proceedings of the 8th annual conference on Innovation and technology in computer science education (ITiCSE)*. New York, New York, USA: ACM Press (cited on page 25).

Koutavas, Vasileios and Mitchell Wand (2006).
Small bisimulations for reasoning about higher-order imperative programs. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 141–152 (cited on pages 57, 129).

Krishnaswami, Neelakantan R., Aaron Turon, Derek Dreyer, and Deepak Garg (2012).
Superficially substructural types. *In proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)* (cited on page 134).

Kuper, Lindsey and Ryan R. Newton (2012).
A Lattice-Based Approach to Deterministic Parallelism with Shared State (cited on page 18).

Lauer, Hugh C. and Roger M. Needham (1979).
On the duality of operating system structures. *ACM SIGOPS Operating Systems Review*, 13(2):3–19 (cited on page 14).

Le Fessant, Fabrice and Luc Maranget (1998).
Compiling Join Patterns. *In proceedings of the International Workshop on High-Level Concurrent Languages (HLCL)* (cited on page 149).

Lea, Doug.
Concurrency JSR-166 Interest Site. URL: http://gee.cs.oswego.edu/dl/concurrency-interest/ (cited on page 4).

Lea, Doug (2000).
A Java fork/join framework. *In proceedings of the ACM 2000 Conference on Java Grande (JAVA)*. New York, New York, USA: ACM Press, pages 36–43 (cited on page 4).

Lea, Doug (2005).
The java.util.concurrent synchronizer framework. *Science of Computer Programming*, 58(3):293–309 (cited on pages 4, 161, 173, 216).

Leitner, Felix von (2009).
Source Code Optimization. URL: http://www.linux-kongress.org/2009/slides/compiler_survey_felix_von_leitner.pdf (cited on page 5).

Lesani, Mohsen and Jens Palsberg (2011).
Communicating memory transactions. *In proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (cited on page 212).

Ley-Wild, Ruy and Aleksandar Nanevski (2013).
Subjective Auxiliary State for Coarse-Grained Concurrency. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (cited on pages 75, 133).

Liang, Hongjin and Xinyu Feng (2013).
Modular Verification of Linearizability with Non-Fixed Linearization Points (cited on pages 131, 135).

Liang, Hongjin, Xinyu Feng, and Ming Fu (2012).
A rely-guarantee-based simulation for verifying concurrent program transformations. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (cited on page 131).

Lipton, Richard J (1975).
Reduction: a method of proving properties of parallel programs. *Communications of the ACM (CACM)*, 18(12):717–721 (cited on page 135).

Liu, Yigong (2009).
Asynchronous Message Coordination and Concurrency Library. URL: http://channel.sourceforge.net/ (cited on page 213).

Lucassen, J M and D K Gifford (1988).
Polymorphic effect systems. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 47–57 (cited on page 129).

Luchangco, Victor and V.J. J Marathe (2011).
Transaction communicators: enabling cooperation among concurrent transactions. *In proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (cited on page 212).

Luchangco, Victor, Mark Moir, and Nir Shavit (2003).
Nonblocking k-compare-single-swap. *In proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (cited on page 197).

Lynch, Nancy and Frits Vaandrager (1995).
Forward and Backward Simulations: Part I: Untimed Systems. *Information and Computation*, 121(2):214–233. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.12.3241&rep=rep1&type=pdf (cited on pages 74, 138).

Manolios, Panagiotis (2003).
A compositional theory of refinement for branching time. *In proceedings of the Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 304–318 (cited on page 75).

Manolios, Panagiotis and Richard Trefler (2003).
A lattice-theoretic characterization of safety and liveness. *In proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. New York, New York, USA: ACM Press, pages 325–333 (cited on page 39).

Manson, Jeremy, William Pugh, and Sarita V. Adve (2005).
The Java memory model. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. New York, New York, USA: ACM Press, pages 378–391 (cited on page 12).

Martin, Milo M. K., Mark D. Hill, and Daniel J. Sorin (2012).
Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7):78–89 (cited on page 27).

McKenney, Paul E., Dipankar Sarma, and Maneesh Soni (2004).
Scaling dcache with RCU. URL: http://www.linuxjournal.com/article/7124 (cited on page 13).

McKenney, Paul E. and John D. Slingwine (1998).
Read-copy update: Using execution history to solve concurrency problems. *Parallel and Distributed Computing and Systems*, pages 509–518. URL: http://www2.rdrop.com/users/paulmck/RCU/rclockjrnl_tpds_mathtype.pdf (cited on pages 35, 51).

Mellor-Crummey, John M. and Michael L. Scott (1991).
Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65 (cited on pages 38, 39, 167, 204).

Meyer, A. R. and K. Sieber (1988).
Towards fully abstract semantics for local variables. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (cited on page 19).

Michael, Maged M. (2004).
Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 15(6):491–504 (cited on page 35).

Michael, Maged M. and Michael L. Scott (1996).
Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. *In proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, pages 267–275 (cited on pages 167, 189, 204).

Michael, Maged M. and Michael L. Scott (1998).
Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26 (cited on pages 64, 150).

Milner, R (1982).
A Calculus of Communicating Systems. Springer-Verlag New York, Inc. (cited on page 17).

Milner, Robin (1977).
Fully abstract models of the lambda calculus. *Theoretical Computer Science*, 4(1):1–22. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=21953 (cited on page 56).

Milner, Robin (1993).
Elements of interaction: Turing award lecture. *Communications of the ACM*, 36(1):78–89 (cited on page 17).

Mitchell, John C. (1986).
Representation independence and data abstraction. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. New York, New York, USA: ACM Press, pages 263–276 (cited on pages 20, 127).

Molka, Daniel, Daniel Hackenberg, Robert Schone, and Matthias S. Muller (2009).
Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. *In proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, pages 261–270 (cited on page 28).

Moore, Katherine F. and Dan Grossman (2008).
High-level small-step operational semantics for transactions. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. New York, New York, USA: ACM Press, pages 51–62 (cited on page 55).

Morris Jr., James H (1973).
Protection in programming languages. *Communications of the ACM (CACM)*, 16(1):15–21 (cited on page 59).

Nain, Sumit and Moshe Y. Vardi (2007).
Branching vs. Linear Time: Semantical Perspective. *Automated Technology for Verification and Analysis (ATVA)* (cited on page 138).

Odersky, Martin (2002).
An Overview of Functional Nets. *APPSEM Summer School, Caminha, Portugal, September 2000* (cited on page 213).

O'Hearn, Peter W. (2007).
Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307 (cited on page 70).

O'Hearn, Peter W. and David J. Pym (1999).
The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244. URL: http://www.jstor.org/stable/10.2307/421090 (cited on page 90).

O'Hearn, Peter W., Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh (2010).
Verifying linearizability with hindsight. *In proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)* (cited on page 131).

Parkinson, Matthew and Gavin Bierman (2005).
Separation logic and abstraction. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 40(1):247–258 (cited on page 132).

Perlis, Alan J. (1982).
Epigrams on programming. *ACM SIGPLAN Notices*, 17(9):7–13 (cited on page 3).

Peyton Jones, Simon (2001).
Tackling the awkward squad. *Engineering theories of software construction*, pages 47–96. URL: http://research.microsoft.com/en-us/um/people/simonpj/papers/marktoberdorf/ (cited on page 193).

Peyton Jones, Simon, Andrew Gordon, and Sigbjorn Finne (1996).
Concurrent Haskell. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. New York, New York, USA: ACM Press, pages 295–308 (cited on page 15).

Peyton Jones, Simon and Philip Wadler (1993).
Imperative functional programming. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (cited on pages 175, 188, 211).

Pike, Rob (2012).
Concurrency is not Parallelism (it's better). URL: http://concur.rspace.googlecode.com/hg/talk/concur.html (cited on page 11).

Pitts, Andrew M. (2002).
Operational Semantics and Program Equivalence. *Applied Semantics, Advanced Lectures*. Edited by G Barthe, P Dybjer, and J Saraiva. Volume 2395. Lecture Notes in Computer Science, Tutorial. Springer-Verlag, pages 378–412. URL: http://www.cl.cam.ac.uk/~amp12/papers/opespe/opespe-lncs.pdf (cited on page 128).

Pitts, Andrew M. (2005).
Typed Operational Reasoning. *Advanced Topics in Types and Programming Languages*. Edited by B C Pierce. The MIT Press. Chapter 7, pages 245–289 (cited on page 128).

Pitts, Andrew M. and Ian Stark (1998).
Operational reasoning for functions with local state. *Higher order operational techniques in semantics*, pages 227–274. URL: http://www.cl.cam.ac.uk/~amp12/papers/operfl/operfl.pdf (cited on pages 57, 61, 100, 128).

Plociniczak, Hubert and Susan Eisenbach (2010).
JErlang: Erlang with Joins. *Coordination Models and Languages*. Volume 6116. Lecture Notes in Computer Science, pages 61–75 (cited on page 213).

Plotkin, Gordon D. and Martín Abadi (1993).
A logic for parametric polymorphism. *International Conference on Typed Lambda Calculi and Applications (TLCA)*, pages 361–375 (cited on pages 90, 129, 222).

Reppy, John (1991).
CML: A higher concurrent language. *In proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 293–305 (cited on page 209).

Reppy, John (1992).
Higher-order concurrency. PhD thesis. Cornell University. URL: http://people.cs.uchicago.edu/~jhr/papers/1992/phd-thesis.ps.gz (cited on pages 11, 15, 175, 177).

Reppy, John (2007).
Concurrent programming in ML. Cambridge University Press. URL: http://books.google.com/books?hl=en&lr=&id=V_0CCK8wcJUC&oi=fnd&pg=PP1&dq=Concurrent+programming+in+ML&ots=6i8BTg1UXK&sig=xKpYlrdadlTv2rfQs-JAoowJ2hs (cited on page 176).

Reppy, John, Claudio V. Russo, and Yingqi Xiao (2009).
Parallel concurrent ML. *In proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, pages 257–268 (cited on page 209).

Reynolds, John C. (1983).
Types, abstraction and parametric polymorphism. *Information processing*. URL: http://www.cs.cmu.edu/afs/cs/user/jcr/ftp/typesabpara.pdf (cited on pages 20, 52, 83, 127, 130).

Reynolds, John C. (2002).
Separation logic: a logic for shared mutable data structures. *In proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, pages 55–74 (cited on pages 60, 68, 95).

Russo, Claudio V. (2007).
The Joins Concurrency Library. *In proceedings of Practical Aspects of Declarative Languages (PADL)*. Springer-Verlag, pages 260–274 (cited on pages 8, 141, 150, 187, 213).

Russo, Claudio V. (2008).
Join Patterns for Visual Basic. *In proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)* (cited on pages 148, 165, 213).

Sangiorgi, Davide, Naoki Kobayashi, and Eijiro Sumii (2007).
Environmental Bisimulations for Higher-Order Languages. *In proceedings*

of the IEEE Symposium on Logic in Computer Science (LICS). Volume 33. 1, pages 293–302 (cited on pages 57, 129).

Scherer, III, William N., Doug Lea, and Michael L. Scott (2005).
A scalable elimination-based exchange channel. *In proceedings of the Workshop on Synchronization and Concurrency in Object Oriented Languages (SCOOL)*. Citeseer (cited on pages 4, 146, 172, 181).

Scherer, III, William N., Doug Lea, and Michael L. Scott (2006).
Scalable synchronous queues. *In proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. Volume 52. 5. New York, New York, USA: ACM Press, pages 147–156 (cited on page 4).

Scherer, III, William N. and Michael L. Scott (2004).
Nonblocking Concurrent Data Structures with Condition Synchronization. *In proceedings of Distributed Computing (DISC)*. Springer, pages 174–187 (cited on pages 39, 167, 204, 216).

Shavit, Nir and Dan Touitou (1995).
Software transactional memory. *In proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. New York, New York, USA: ACM Press, pages 204–213 (cited on pages 6, 149, 177).

Shavit, Nir and Dan Touitou (1997).
Software transactional memory. *In proceedings of Distributed Computing (DISC)*, 10(2):99–116 (cited on page 210).

Singh, Satnam (2006).
Higher-Order combinators for join patterns using STM. *In proceedings of the TRANSACT workshop*. URL: https://urresearch.rochester.edu/fileDownloadForInstitutionalItem.action?itemId=3699&itemFileId=5278 (cited on page 214).

Smaragdakis, Yannis, Anthony Kay, Reimer Behrends, and Michal Young (2007).
Transactions with isolation and cooperation. *In proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)* (cited on page 212).

Sulzmann, Martin and Edmund S L Lam (2009).
Parallel Join Patterns with Guards and Propagation. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.151.1104&rep=rep1&type=pdf (cited on page 214).

Sumii, Eijiro and Benjamin C. Pierce (2005).
A bisimulation for type abstraction and recursion. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Volume 40. 1, pages 63–74 (cited on pages 57, 129).

Sundell, Hå kan, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas (2011).
A lock-free algorithm for concurrent bags. *In proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. New York, New York, USA: ACM Press, pages 335–344 (cited on page 150).

Sutherland, Ivan E. and Jo Ebergen (2002).
Computers without Clocks. *Scientific American*, 287(2):62–69 (cited on page 17).

Svendsen, Kasper, Lars Birkedal, and Matthew Parkinson (2013).
Modular Reasoning about Separation of Concurrent Data Structures. *In proceedings of the European Symposium on Programming (ESOP)*. URL: http://www.itu.dk/people/kasv/hocap-ext.pdf (cited on page 132).

Swaine, James, Kevin Tew, Peter Dinda, Robert Bruce Findler, and Matthew Flatt (2010).
Back to the futures: incremental parallelization of existing sequential runtime systems. *In proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*. New York, New York, USA: ACM Press (cited on page 223).

Tobin-Hochstadt, Sam (2010).
Typed Scheme: From Scripts to Programs. PhD thesis. Northeastern University (cited on page 223).

Treiber, R. K. (1986).
Systems programming: Coping with parallelism. Technical report. IBM Almaden Research Center (cited on page 34).

Turon, Aaron (2012).
Reagents (cited on page 10).

Turon, Aaron and Claudio V. Russo (2011).
Scalable Join Patterns. *In proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)* (cited on page 10).

Turon, Aaron, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer (2013).
Logical relations for fine-grained concurrency (cited on page 10).

Turon, Aaron and Mitchell Wand (2011).
A separation logic for refining concurrent objects. *In proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (cited on pages 10, 131).

Vafeiadis, Viktor (2008).
Modular fine-grained concurrency verification. PhD thesis. University of Cambridge (cited on page 134).

Vafeiadis, Viktor and Matthew Parkinson (2007).
A Marriage of Rely/Guarantee and Separation Logic. *In proceedings of Concurrency Theory (CONCUR)* (cited on page 134).

Van Roy, Peter and Seif Haridi (2004).
Concepts, Techniques, and Models of Computer Programming. URL: http://www.info.ucl.ac.be/~pvr/book.html (cited on pages 15, 16, 18).

Von Itzstein, G. Stewart and David Kearney (2001).
Join Java: An alternative concurrency semantics for Java. Technical report ACRC-01-001. University of South Australia (cited on page 213).

Part V

TECHNICAL APPENDIX

A

Reference: the F^{μ}_{cas} calculus

Values	v	::=	$()$	Unit value
		|	**true**	Boolean value
		|	**false**	Boolean value
		|	n	Number value
		|	(v, v)	Pair value
		|	**rec** $f(x).e$	Recursive function
		|	$\Lambda.e$	Type abstraction
		|	ℓ	Heap location
		|	**null**	Null optional reference
		|	x	Variable
Expressions	e	::=	v	Value
		|	**if** e **then** e **else** e	Conditional
		|	$e + e$	Addition
		|	(e, e)	Pair introduction
		|	**let** $(x, y) = e$ **in** e	Pair elimination
		|	$e\ e$	Function application
		|	$e\ \_$	Type application
		|	**case**$(e,$ **null** $\Rightarrow e, x \Rightarrow e)$	Optional reference elimination
		|	$\mathbf{inj}_i\ e$	Tagged union injection
		|	**case**$(e, \mathbf{inj}_1\ x \Rightarrow e, \mathbf{inj}_2\ y \Rightarrow e)$	Tagged union elimination
		|	**new** $\overline{e}$	Mutable tuple allocation
		|	**get**$(e[i])$	Mutable tuple dereference
		|	$e[i] := e$	Mutable tuple assignment
		|	**cas**$(e[i], e, e)$	Mutable tuple atomic update
		|	**fork** e	Process forking

Figure A.1: Syntax of values and expressions

Figure A.2: Syntax of types

Comparable types	σ	::=	unit	Unit (*i.e.,* nullary tuple)
		|	bool	Boolean
		|	nat	Natural number
		|	$\tau + \tau$	Tagged union
		|	$\mathsf{ref}(\overline{\tau})$	Mutable tuple reference
		|	$\mathsf{ref}_?(\overline{\tau})$	Optional reference
		|	$\mu\alpha.\sigma$	Recursive comparable type
Types	τ	::=	σ	Comparable type
		|	α	Type variable
		|	$\tau \times \tau$	Immutable pair type
		|	$\mu\alpha.\tau$	Recursive type
		|	$\forall\alpha.\tau$	Polymorphic type
		|	$\tau \to \tau$	Function type
Type variable contexts	Δ	::=	$\cdot \mid \Delta, \alpha$	
Term variable contexts	Γ	::=	$\cdot \mid \Gamma, x : \tau$	
Combined contexts	Ω	::=	$\Delta; \Gamma$	

▸ Well-typed terms $\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\Omega \vdash () : \mathsf{unit} \qquad \Omega \vdash \mathsf{true} : \mathsf{bool} \qquad \Omega \vdash \mathsf{false} : \mathsf{bool} \qquad \Omega \vdash n : \mathsf{nat} \qquad \Omega, x : \tau \vdash x : \tau$$

$$\frac{\Omega \vdash e : \mathsf{bool} \qquad \Omega \vdash e_i : \tau}{\Omega \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau} \qquad \frac{\Omega \vdash e_1 : \tau_1 \qquad \Omega \vdash e_2 : \tau_2}{\Omega \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Omega \vdash e : \tau_1 \times \tau_2 \qquad \Omega, x : \tau_1, y : \tau_2 \vdash e' : \tau}{\Omega \vdash \mathsf{let}\ (x, y) = e\ \mathsf{in}\ e' : \tau}$$

$$\frac{\Omega, f : \tau' \to \tau, x : \tau' \vdash e : \tau}{\Omega \vdash \mathsf{rec}\ f(x).e : \tau' \to \tau} \qquad \frac{\Omega \vdash e : \tau' \to \tau \qquad \Omega \vdash e' : \tau'}{\Omega \vdash e\ e' : \tau} \qquad \Omega \vdash \mathsf{null} : \mathsf{ref}_?(\overline{\tau}) \qquad \frac{\Omega \vdash e : \mathsf{ref}(\overline{\tau})}{\Omega \vdash e : \mathsf{ref}_?(\overline{\tau})}$$

$$\frac{\Omega \vdash e : \mathsf{ref}_?(\overline{\tau}) \qquad \Omega \vdash e_1 : \tau \qquad \Omega, x : \mathsf{ref}(\overline{\tau}) \vdash e_2 : \tau}{\Omega \vdash \mathsf{case}(e, \mathsf{null} \Rightarrow e_1, x \Rightarrow e_2) : \tau} \qquad \frac{\Omega \vdash e : \tau_i}{\Omega \vdash \mathsf{inj}_i\ e : \tau_1 + \tau_2}$$

$$\frac{\Omega \vdash e : \tau_1 + \tau_2 \qquad \Omega, x : \tau_i \vdash e_i : \tau}{\Omega \vdash \mathsf{case}(e, \mathsf{inj}_1\ x \Rightarrow e_1, \mathsf{inj}_2\ x \Rightarrow e_2) : \tau} \qquad \frac{\Omega, \alpha \vdash e : \tau}{\Omega \vdash \Lambda.e : \forall\alpha.\tau} \qquad \frac{\Omega \vdash e : \forall\alpha.\tau}{\Omega \vdash e\ \_ : \tau[\tau'/\alpha]} \qquad \frac{\Omega \vdash e_i : \tau_i}{\Omega \vdash \mathsf{new}\ (\overline{e}) : \mathsf{ref}(\overline{\tau})}$$

$$\frac{\Omega \vdash e : \mathsf{ref}(\overline{\tau})}{\Omega \vdash \mathsf{get}(e[i]) : \tau_i} \qquad \frac{\Omega \vdash e : \mathsf{ref}(\overline{\tau}) \qquad \Omega \vdash e' : \tau_i}{\Omega \vdash e[i] := e' : \mathsf{unit}} \qquad \frac{\Omega \vdash e : \mathsf{ref}(\overline{\tau}) \qquad \tau_i = \sigma \qquad \Omega \vdash e_o : \sigma \qquad \Omega \vdash e_n : \sigma}{\Omega \vdash \mathsf{cas}(e[i], e_o, e_n) : \mathsf{bool}}$$

$$\frac{\Omega \vdash e : \mathsf{unit}}{\Omega \vdash \mathsf{fork}\ e : \mathsf{unit}} \qquad \frac{\Omega \vdash e : \mu\alpha.\tau}{\Omega \vdash e : \tau[\mu\alpha.\tau/\alpha]} \qquad \frac{\Omega \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Omega \vdash e : \mu\alpha.\tau}$$

Figure A.3: Typing rules

$$
\begin{array}{lrcl}
\textsc{Heap-stored values} & u & ::= & (\overline{v}) \mid \mathsf{inj}_i\ v \\
\textsc{Heaps} & h & \in & \mathrm{Heap} \triangleq \mathrm{Loc} \rightharpoonup \mathrm{HeapVal} \\
\textsc{Thread pools} & T & \in & \mathrm{ThreadPool} \triangleq \mathbb{N} \overset{\mathrm{fin}}{\rightharpoonup} \mathrm{Expression} \\
\textsc{Configurations} & \varsigma & ::= & h; T \\
\textsc{Evaluation contexts} & K & ::= & [\,] \mid \mathsf{if}\ K\ \mathsf{then}\ e\ \mathsf{else}\ e \mid K + e \mid v + K \mid (K, e) \mid (v, K) \\
 & & \mid & \mathsf{let}\ (x, y) = K\ \mathsf{in}\ e \mid K\ e \mid v\ K \mid \mathsf{inj}_i\ K \mid K\ \_ \\
 & & \mid & \mathsf{case}(K, \mathsf{inj}_1\ x \Rightarrow e, \mathsf{inj}_2\ x \Rightarrow e) \mid \mathsf{case}(K, \mathsf{null} \Rightarrow e, x \Rightarrow e) \\
 & & \mid & \mathsf{new}\ (\overline{v}, K, \overline{e}) \mid \mathsf{get}(K[i]) \mid K[i] := e \mid v[i] := K \\
 & & \mid & \mathsf{cas}(K[i], e, e) \mid \mathsf{cas}(v[i], K, e) \mid \mathsf{cas}(v[i], v, K)
\end{array}
$$

Figure A.4: Execution syntax

▸ PRIMITIVE REDUCTIONS $\boxed{h; e \hookrightarrow h'; e'}$

$$
\begin{array}{rcll}
h; n + m & \hookrightarrow & h; k & \text{when } k = n + m \\
h; \mathsf{get}(\ell[i]) & \hookrightarrow & h; v_i & \text{when } h(\ell) = (\overline{v}) \\
h; \ell[i] := v & \hookrightarrow & h[\ell[i] = v]; () & \text{when } \ell \in \mathrm{dom}(h) \\
h; \mathsf{cas}(\ell[i], v_o, v_n) & \hookrightarrow & h[\ell[i] = v_n]; \mathsf{true} & \text{when } h(\ell)[i] = v_o \\
h; \mathsf{cas}(\ell[i], v_o, v_n) & \hookrightarrow & h; \mathsf{false} & \text{when } h(\ell)[i] \neq v_o \\
h; \mathsf{case}(\ell, \mathsf{inj}_1\ x \Rightarrow e_1, \mathsf{inj}_2\ x \Rightarrow e_2) & \hookrightarrow & h; e_i[v/x] & \text{when } h(\ell) = \mathsf{inj}_i\ v
\end{array}
$$

$$
\begin{array}{rcl}
h; \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 & \hookrightarrow & h; e_1 \\
h; \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 & \hookrightarrow & h; e_1 \\
h; \mathsf{case}(\mathsf{null}, \mathsf{null} \Rightarrow e_1, x \Rightarrow e_2) & \hookrightarrow & h; e_1 \\
h; \mathsf{case}(\ell, \mathsf{null} \Rightarrow e_1, x \Rightarrow e_2) & \hookrightarrow & h; e_2[\ell/x] \\
h; \mathsf{let}\ (x, y) = (v_1, v_2)\ \mathsf{in}\ e & \hookrightarrow & h; e[v_1/x, v_2/y] \\
h; \mathsf{rec}\ f(x).e\ v & \hookrightarrow & h; e[\mathsf{rec}\ f(x).e/f, v/x] \\
h; \mathsf{inj}_i\ v & \hookrightarrow & h \uplus [\ell \mapsto \mathsf{inj}_i\ v]; \ell \\
h; \Lambda.e\ \_ & \hookrightarrow & h; e \\
h; \mathsf{new}\ (\overline{v}) & \hookrightarrow & h \uplus [\ell \mapsto (\overline{v})]; \ell
\end{array}
$$

▸ GENERAL REDUCTION $\boxed{h; T \to h'; T'}$

$$
\frac{h; e \hookrightarrow h'; e'}{h; T \uplus [i \mapsto K[e]] \to h'; T \uplus [i \mapsto K[e']]}
\qquad
h; T \uplus [i \mapsto K[\mathsf{fork}\ e]] \to h; T \uplus [i \mapsto K[()]] \uplus [j \mapsto e]
$$

Figure A.5: Operational semantics

Figure A.6: Pure reductions

▸ PURE REDUCTIONS $\boxed{e \overset{\text{pure}}{\hookrightarrow} e'}$

$$
\begin{array}{rcll}
n + m & \overset{\text{pure}}{\hookrightarrow} & k & \text{when } k = n + m \\
\textbf{if true then } e_1 \textbf{ else } e_2 & \overset{\text{pure}}{\hookrightarrow} & e_1 & \\
\textbf{if false then } e_1 \textbf{ else } e_2 & \overset{\text{pure}}{\hookrightarrow} & e_1 & \\
\textbf{case}(\textbf{null}, \textbf{null} \Rightarrow e_1, x \Rightarrow e_2) & \overset{\text{pure}}{\hookrightarrow} & e_1 & \\
\textbf{case}(\ell, \textbf{null} \Rightarrow e_1, x \Rightarrow e_2) & \overset{\text{pure}}{\hookrightarrow} & e_2[\ell/x] & \\
\textbf{let } (x, y) = (v_1, v_2) \textbf{ in } e & \overset{\text{pure}}{\hookrightarrow} & e[v_1/x, v_2/y] & \\
\textbf{rec } f(x).e\ v & \overset{\text{pure}}{\hookrightarrow} & e[\textbf{rec } f(x).e/f, v/x] & \\
\Lambda.e\ \_ & \overset{\text{pure}}{\hookrightarrow} & e &
\end{array}
$$

Figure A.7: Contextual refinement

If $\Omega \vdash e_{\text{I}} : \tau$ and $\Omega \vdash e_{\text{S}} : \tau$, we say e_{I} *contextually refines* e_{S}, written $\Omega \vDash e_{\text{I}} \preceq e_{\text{S}} : \tau$, if:

$$
\begin{array}{l}
\text{for every } i, j \text{ and } C : (\Omega, \tau) \rightsquigarrow (\varnothing, \textbf{nat}) \text{ we have} \\
\quad \forall n. \forall T_{\text{I}}. \quad \varnothing; [i \mapsto C[e_{\text{I}}]] \to^* h_{\text{I}}; [i \mapsto n] \uplus T_{\text{I}} \\
\quad \Longrightarrow \exists T_{\text{S}}. \quad \varnothing; [j \mapsto C[e_{\text{S}}]] \to^* h_{\text{S}}; [j \mapsto n] \uplus T_{\text{S}}
\end{array}
$$

Figure A.8: Derived forms

$$
\begin{array}{rcl}
\lambda x.e & \triangleq & \textbf{rec } \_(x).e \\
\textbf{let } x = e \textbf{ in } e' & \triangleq & (\lambda \_.e')\ e \\
e; e' & \triangleq & \textbf{let } \_ = e \textbf{ in } e' \\
\\
\mathsf{acq} & \triangleq & \textbf{rec } f(x).\ \textbf{if } \textbf{cas}(x, \textbf{false}, \textbf{true}) \textbf{ then } () \textbf{ else } f(x) \\
\mathsf{rel} & \triangleq & \lambda x.\ x := \textbf{false} \\
\mathsf{withLock}(\mathsf{lock}, e) & \triangleq & \lambda x.\ \mathsf{acq}(\mathsf{lock});\ \textbf{let } r = e(x) \textbf{ in } \mathsf{rel}(\mathsf{lock});\ r \\
\mathsf{mkAtomic}(e_1, \ldots, e_n) & \triangleq & \textbf{let } \mathsf{lock} = \textbf{new } (\textbf{false}) \textbf{ in} \\
 & & (\mathsf{withLock}(\mathsf{lock}, e_1),\ \ldots,\ \mathsf{withLock}(\mathsf{lock}, e_n))
\end{array}
$$

B

Reference: the logic of local protocols

Assertions	P	::=	$v = v$	Equality of values
		$\mid$	emp	Empty resource
$(\underline{\ell} ::= \ell \mid x)$		$\mid$	$\underline{\ell} \mapsto_{\mathrm{I}} u$	Singleton implementation heap
		$\mid$	$\underline{\ell} \mapsto_{\mathrm{S}} u$	Singleton specification heap
$(\underline{i} ::= i \mid x)$		$\mid$	$\underline{i} \rightarrowtail_{\mathrm{S}} e$	Singleton specification thread
		$\mid$	$\underline{i} \mapsto \iota$	Island assertion
		$\mid$	$P * P$	Separating conjunction
		$\mid$	$P \Rightarrow P$	Implication
		$\mid$	$P \wedge P$	Conjunction
		$\mid$	$P \vee P$	Disjunction
		$\mid$	$\exists x.P$	Existential quantification
		$\mid$	$\forall x.P$	Universal quantification
		$\mid$	$P \oplus P$	Speculative disjunction
		$\mid$	φ	Pure code assertion
		$\mid$	$\triangleright P$	Later modality
		$\mid$	$T@m\ \{x.\ P\}$	Threadpool simulation
Pure code assertions	φ	::=	$\{P\}\ e\ \{x.\ Q\}$	Hoare triple
		$\mid$	$v \preceq^{\mathcal{V}} v : \tau$	Value refinement
		$\mid$	$\Omega \vdash e \preceq^{\mathcal{E}} e : \tau$	Expression refinement
Island descriptions	ι	::=	(θ, I, s, A)	
			where $I \in \theta.S \to \mathrm{Assert}$,	State interpretation
			$s \in \theta.S$,	Current state (rely-lower-bound)
			$A \subseteq \theta.A$,	Owned tokens
			$A \# \theta.F(s)$	(which must not be free)
State transition systems	θ	::=	$(S, A, \rightsquigarrow, F)$	
			where S a set,	States
			A a set,	Tokens
			$\rightsquigarrow \subseteq S \times S$,	Transition relation
			$F \in S \to \wp(A)$	Free tokens
Main thread indicators	m	::=	i	ID of main thread
		$\mid$	none	No main thread

Figure B.1: Syntax of assertions

▸ Domains

$$\begin{aligned}
\text{StateSet} &\triangleq \{\, \Sigma \subseteq \text{Heap} \times \text{ThreadPool} \mid \Sigma \text{ finite, nonempty} \,\} \\
\text{Resource} &\triangleq \{\, \eta \in \text{Heap} \times \text{StateSet} \,\} \\
\text{Island}_n &\triangleq \left\{ \iota = (\theta, J, s, A) \;\middle|\; \begin{array}{l} \theta \in \text{STS},\ \ s \in \theta.S,\ \ J \in \theta.S \to \text{UWorld}_n \overset{\text{mon}}{\to} \wp(\text{Resource}), \\ A \subseteq \theta.A,\ \ A \# \theta.F(s),\ \ J(s) \neq \varnothing \end{array} \right\} \\
\text{World}_n &\triangleq \left\{ W = (k, \omega) \;\middle|\; k < n, \quad \omega \in \mathbb{N} \overset{\text{fin}}{\rightharpoonup} \text{Island}_k \right\} \\
\text{UWorld}_n &\triangleq \{\, U \in \text{World}_n \mid U = |U| \,\} \\
\text{VRel}_n &\triangleq \left\{ V \in \text{UWorld}_n \overset{\text{mon}}{\to} \wp(\text{Val} \times \text{Val}) \right\}
\end{aligned}$$

▸ Island and world operations

$$\begin{aligned}
|(\theta, J, s, A)| &\triangleq (\theta, J, s, \varnothing) & \lfloor (\theta, J, s_0, A) \rfloor_k &\triangleq (\theta, \lambda s.I(s) \upharpoonright \text{UWorld}_k, s_0, A) \\
|(k, \omega)| &\triangleq (k, \lambda i.|\omega(i)|) & \triangleright(k+1, \omega) &\triangleq (k, \lambda i.\lfloor \omega(i) \rfloor_k) \\
\text{frame}(\theta, J, s, A) &\triangleq (\theta,\ J,\ s,\ \theta.A - \theta.F(s) - A) & \text{interp}(\theta, J, s, A) &\triangleq J(s) \\
\text{frame}(k, \omega) &\triangleq (k,\ \lambda i.\text{frame}(\omega(i)))
\end{aligned}$$

▸ Composition

$$\begin{aligned}
&\text{State sets} && \Sigma_1 \otimes \Sigma_2 && \triangleq \{\, h_1 \uplus h_2; T_1 \uplus T_2 \mid h_i; T_i \in \Sigma_i \,\} && \text{when } \textit{all} \text{ compositions are defined} \\
&\text{Resources} && (h_1, \Sigma_1) \otimes (h_2, \Sigma_2) && \triangleq (h_1 \uplus h_2, \Sigma_1 \otimes \Sigma_2) \\
&\text{Islands} && (\theta, J, s, A) \otimes (\theta', J', s', A') && \triangleq (\theta, J, s, A \uplus A') && \text{when } \theta = \theta',\ s = s',\ J = J' \\
&\text{Worlds} && (k, \omega) \otimes (k', \omega') && \triangleq (k, \lambda i.\omega(i) \otimes \omega'(i)) && \text{when } k = k',\ \text{dom}(\omega) = \text{dom}(\omega')
\end{aligned}$$

▸ Protocol conformance

$$\begin{aligned}
&\text{Protocol step} && \theta \vdash (s, A) \rightsquigarrow (s', A') && \triangleq s \rightsquigarrow_\theta s', \quad \theta.F(s) \uplus A = \theta.F(s') \uplus A' \\
&\text{Island guarantee move} && (\theta, J, s, A) \overset{\text{guar}}{\sqsubseteq} (\theta', J', s', A') && \triangleq \theta = \theta', \quad J = J', \quad \theta \vdash (s, A) \rightsquigarrow^* (s', A') \\
&\text{Island rely move} && \iota \overset{\text{rely}}{\sqsubseteq} \iota' && \triangleq \text{frame}(\iota) \overset{\text{guar}}{\sqsubseteq} \text{frame}(\iota') \\
&\text{World guarantee move} && (k, \omega) \overset{\text{guar}}{\sqsubseteq} (k', \omega') && \triangleq k \geq k', \quad \forall i \in \text{dom}(\omega).\ \lfloor \omega(i) \rfloor_{k'} \overset{\text{guar}}{\sqsubseteq} \omega'(i) \\
&\text{World rely move} && W \overset{\text{rely}}{\sqsubseteq} W' && \triangleq \text{frame}(W) \overset{\text{guar}}{\sqsubseteq} \text{frame}(W')
\end{aligned}$$

▸ World satisfaction

$$\eta : W, \eta' \triangleq W.k > 0 \implies \eta = \eta' \otimes \overline{\eta_i},\ \forall i \in \text{dom}(W.\omega).\ \eta_i \in \text{interp}(W.\omega(i))(\triangleright|W|)$$

Figure B.2: Semantic structures and operations on them

▸ The semantics of resource and protocol assertions, and the connectives.

R	$W, \eta \vDash^\rho R$ iff
φ	$\lvert W\rvert \vDash^\rho \varphi$
$v_1 = v_2$	$v_1 = v_2$
emp	$W = \lvert W\rvert, \eta = (\varnothing, \{\varnothing;\varnothing\})$
$P \wedge Q$	$W, \eta \vDash^\rho P$ and $W, \eta \vDash^\rho Q$
$P \vee Q$	$W, \eta \vDash^\rho P$ or $W, \eta \vDash^\rho Q$
$\forall x.P$	$\forall v.\ W, \eta \vDash^\rho P[v/x]$
$\exists x.P$	$\exists v.\ W, \eta \vDash^\rho P[v/x]$
$\triangleright P$	$W.k > 0 \implies \triangleright W, \eta \vDash^\rho P$

R	$W, \eta \vDash^\rho R$ iff
$P \Rightarrow Q$	$\forall W' \overset{\text{rely}}{\sqsupseteq} W.\ W', \eta \vDash^\rho P \implies W', \eta \vDash^\rho Q$
$\ell \mapsto_{\mathrm{I}} u$	$\eta = ([\ell \mapsto u], \{\varnothing;\varnothing\})$
$\ell \mapsto_{\mathrm{s}} u$	$\eta = (\varnothing, \{[\ell \mapsto u];\varnothing\})$
$i \rightarrowtail_{\mathrm{s}} e$	$\eta = (\varnothing, \{\varnothing;[i \mapsto e]\})$
$i \mapsto (\theta, I, s, A)$	$W.\omega(i) \overset{\text{rely}}{\sqsupseteq} (\theta, [\![I]\!], s, A)$ where $[\![I]\!] \triangleq \lambda s.\lambda U.\{\eta \mid U, \eta \vDash^\rho I(s)\}$
$P_1 * P_2$	$W = W_1 \otimes W_2,\ \eta = \eta_1 \otimes \eta_2,\ W_i, \eta_i \vDash^\rho P_i$
$P_1 \oplus P_2$	$\eta.\Sigma = \Sigma_1 \cup \Sigma_2,\ W, (\eta.h, \Sigma_i) \vDash^\rho P_i$

▸ The semantics of value refinement.

τ_0	v_{I}	v_{s}	$U \vDash^\rho v_{\mathrm{I}} \preceq^{\mathcal{V}} v_{\mathrm{s}} : \tau_0$ iff
τ_b	v	v	$\vdash v : \tau_b$ for $\tau_b \in \{\mathsf{unit}, \mathsf{bool}, \mathsf{nat}\}$
α	v_{I}	v_{s}	$(v_{\mathrm{I}}, v_{\mathrm{s}}) \in \rho(\alpha)(U)$
$\tau_1 \times \tau_2$	$(v_1^{\mathrm{I}}, v_2^{\mathrm{I}})$	$(v_1^{\mathrm{s}}, v_2^{\mathrm{s}})$	$U \vDash^\rho \triangleright(v_1^{\mathrm{I}} \preceq^{\mathcal{V}} v_1^{\mathrm{s}} : \tau_1 \ \wedge\ v_2^{\mathrm{I}} \preceq^{\mathcal{V}} v_2^{\mathrm{s}} : \tau_2)$
$\tau \to \tau'$	$\mathsf{rec}\ f\ x.e_{\mathrm{I}}$	$\mathsf{rec}\ f\ x.e_{\mathrm{s}}$	$U \vDash^\rho \triangleright(x : \tau \vdash e_{\mathrm{I}}[v_{\mathrm{I}}/f] \preceq^{\mathcal{E}} e_{\mathrm{s}}[v_{\mathrm{s}}/f] : \tau')$
$\forall \alpha.\tau$	$\Lambda.e_{\mathrm{I}}$	$\Lambda.e_{\mathrm{s}}$	$U \vDash^\rho \triangleright(\alpha \vdash e_{\mathrm{I}} \preceq^{\mathcal{E}} e_{\mathrm{s}} : \tau)$
$\mu\alpha.\tau$	v_{I}	v_{s}	$U \vDash^\rho v_{\mathrm{I}} \preceq^{\mathcal{V}} v_{\mathrm{s}} : \tau[\mu\alpha.\tau/\alpha]$
$\mathsf{ref}_?(\overline{\tau})$	null ℓ_{I}	null ℓ_{s}	always $U \vDash^\rho \triangleright \ell_{\mathrm{I}} \preceq^{\mathcal{V}} \ell_{\mathrm{s}} : \mathsf{ref}(\overline{\tau})$
$\mathsf{ref}(\overline{\tau})$	ℓ_{I}	ℓ_{s}	$U \vDash^\rho \mathrm{inv}(\exists \overline{x}, \overline{y}.\ \bigwedge \overline{x \preceq^{\mathcal{V}} y : \tau} \wedge \ell_{\mathrm{I}} \mapsto_{\mathrm{I}} (\overline{x}) * \ell_{\mathrm{s}} \mapsto_{\mathrm{s}} (\overline{y}))$
$\tau_1 + \tau_2$	ℓ_{I}	ℓ_{s}	$\exists i.\ U \vDash^\rho \exists x, y.\ \triangleright x \preceq^{\mathcal{V}} y : \tau_i \wedge \mathrm{inv}(v_{\mathrm{I}} \mapsto_{\mathrm{I}} \mathsf{inj}_i\ x * v_{\mathrm{s}} \mapsto_{\mathrm{s}} \mathsf{inj}_i\ y)$

where $\mathrm{inv}(P) \triangleq ((\{\mathsf{dummy}\}, \varnothing, \varnothing, \lambda\_.\varnothing), \lambda\_.P, \mathsf{dummy}, \varnothing)$

▸ The semantics of expression refinement.

Ω	$U \vDash^\rho \Omega \vdash e_{\mathrm{I}} \preceq^{\mathcal{E}} e_{\mathrm{s}} : \tau$ iff
$\cdot$	$\forall K, j.\ U \vDash^\rho \{j \rightarrowtail_{\mathrm{s}} K[e_{\mathrm{s}}]\}\ e_{\mathrm{I}}\ \{x.\ \exists y.\ x \preceq^{\mathcal{V}} y : \tau \wedge j \rightarrowtail_{\mathrm{s}} K[y]\}$
$x : \tau', \Omega'$	$\forall v_{\mathrm{I}}, v_{\mathrm{s}}.\ U \vDash^\rho v_{\mathrm{I}} \preceq^{\mathcal{V}} v_{\mathrm{s}} : \tau' \ \Rightarrow\ \Omega' \vdash e_{\mathrm{I}}[v_{\mathrm{I}}/x] \preceq^{\mathcal{E}} e_{\mathrm{s}}[v_{\mathrm{s}}/x] : \tau$
α, Ω'	$\forall V.\ U \vDash^{\rho[\alpha \mapsto V]} \Omega' \vdash e_{\mathrm{I}} \preceq^{\mathcal{E}} e_{\mathrm{s}} : \tau$

▸ The semantics of Hoare triples.

$$U \vDash^\rho \{P\}\ e\ \{x.\ Q\} \ \triangleq\ \forall i.\ U \vDash^\rho P \Rightarrow [i \mapsto e]@i\ \{x.\ Q\}$$

▸ The semantics of threadpool simulation.

$$W_0, \eta \vDash^\rho T@m\ \{x.\ Q\} \ \triangleq\ \forall W \overset{\text{rely}}{\sqsupseteq} W_0,\ \eta_F \# \eta. \quad \text{if } W.k > 0 \text{ and } h, \Sigma : W, \eta \otimes \eta_F \text{ then:}$$

$$h; T \to h'; T' \implies \exists \Sigma', \eta', W' \overset{\text{guar}}{\sqsupseteq_1} W.\ \Sigma \rightrightarrows \Sigma',\ \ h', \Sigma' : W', \eta' \otimes \eta_F,\ \ W', \eta' \vDash^\rho T'@m\ \{x.\ Q\}$$

$$T = T_0 \uplus [m \mapsto v] \implies \exists \Sigma', \eta', W' \overset{\text{guar}}{\sqsupseteq_0} W.\ \Sigma \rightrightarrows \Sigma',\ \ h, \Sigma' : W', \eta' \otimes \eta_F,\ \ W', \eta' \vDash^\rho Q[v/x] * T_0@\mathsf{none}\ \{x.\ \mathsf{tt}\}$$

where $W' \overset{\text{guar}}{\sqsupseteq_n} W \ \triangleq\ W' \overset{\text{guar}}{\sqsupseteq} W \ \wedge\ W.k = W'.k + n$

$\Sigma \rightrightarrows \Sigma' \ \triangleq\ \forall \varsigma' \in \Sigma'.\ \exists \varsigma \in \Sigma.\ \varsigma \to^* \varsigma'$

▸ Laws of intuitionistic first-order logic.

$$\frac{P \in \mathcal{P}}{\mathcal{P} \vdash P} \qquad \frac{\mathcal{P} \vdash P[v/x] \qquad \mathcal{P} \vdash v = v'}{\mathcal{P} \vdash P[v'/x]} \qquad \frac{\mathcal{P} \vdash P \qquad \mathcal{P} \vdash Q}{\mathcal{P} \vdash P \wedge Q} \qquad \frac{\mathcal{P} \vdash P \wedge Q}{\mathcal{P} \vdash P} \qquad \frac{\mathcal{P} \vdash P \wedge Q}{\mathcal{P} \vdash Q}$$

$$\frac{\mathcal{P} \vdash P \vee Q \qquad \mathcal{P}, P \vdash R \qquad \mathcal{P}, Q \vdash R}{\mathcal{P} \vdash R} \qquad \frac{\mathcal{P} \vdash P}{\mathcal{P} \vdash P \vee Q} \qquad \frac{\mathcal{P} \vdash Q}{\mathcal{P} \vdash P \vee Q} \qquad \frac{\mathcal{P}, P \vdash Q}{\mathcal{P} \vdash P \Rightarrow Q} \qquad \frac{\mathcal{P} \vdash P \Rightarrow Q \qquad \mathcal{P} \vdash P}{\mathcal{P} \vdash Q}$$

$$\frac{\mathcal{P} \vdash P[y/x] \qquad y \text{ fresh}}{\mathcal{P} \vdash \forall x.P} \qquad \frac{\mathcal{P} \vdash \forall x.P}{\mathcal{P} \vdash P[v/x]} \qquad \frac{\mathcal{P} \vdash \exists x.P \qquad \mathcal{P}, P[y/x] \vdash Q \qquad y \text{ fresh}}{\mathcal{P} \vdash Q} \qquad \frac{\mathcal{P} \vdash P[v/x]}{\mathcal{P} \vdash \exists x.P}$$

▸ Axioms from the logic of bunched implications.

$$\begin{array}{rcl} P * Q & \iff & Q * P \\ (P * Q) * R & \iff & P * (Q * R) \\ P * \mathsf{emp} & \iff & P \end{array} \qquad \begin{array}{rcl} (P \vee Q) * R & \iff & (P * R) \vee (Q * R) \\ (P \wedge Q) * R & \implies & (P * R) \wedge (Q * R) \\ (\exists x.\, P) * Q & \iff & \exists x.\, (P * Q) \\ (\forall x.\, P) * Q & \implies & \forall x.\, (P * Q) \end{array} \qquad \frac{\mathcal{P}, P_1 \vdash Q_1 \qquad \mathcal{P}, P_2 \vdash Q_2}{\mathcal{P}, P_1 * P_2 \vdash Q_1 * Q_2}$$

▸ Laws for the "later" modality.

$$\frac{\mathcal{P} \vdash P}{\mathcal{P} \vdash \triangleright P}\ \text{Mono} \qquad \frac{\mathcal{P}, \triangleright P \vdash P}{\mathcal{P} \vdash P}\ \text{Löb}$$

$$\begin{array}{rcl} \triangleright(P \wedge Q) & \iff & \triangleright P \wedge \triangleright Q \\ \triangleright(P \vee Q) & \iff & \triangleright P \vee \triangleright Q \end{array} \qquad \begin{array}{rcl} \triangleright \forall x.P & \iff & \forall x. \triangleright P \\ \triangleright \exists x.P & \iff & \exists x. \triangleright P \\ \triangleright(P * Q) & \iff & \triangleright P * \triangleright Q \end{array}$$

▸ Reasoning about refinement.

$$\Phi \vdash () \preceq^{\mathcal{V}} () : \mathbf{unit} \qquad \Phi \vdash \mathbf{true} \preceq^{\mathcal{V}} \mathbf{true} : \mathbf{bool} \qquad \Phi \vdash \mathbf{false} \preceq^{\mathcal{V}} \mathbf{false} : \mathbf{bool} \qquad \Phi \vdash n \preceq^{\mathcal{V}} n : \mathbf{nat}$$

$$\frac{\Phi \vdash \triangleright v_1^{\mathrm{I}} \preceq^{\mathcal{V}} v_1^{\mathrm{S}} : \tau_1 \qquad \Phi \vdash \triangleright v_2^{\mathrm{I}} \preceq^{\mathcal{V}} v_2^{\mathrm{S}} : \tau_2}{\Phi \vdash (v_1^{\mathrm{I}}, v_2^{\mathrm{I}}) \preceq^{\mathcal{V}} (v_1^{\mathrm{S}}, v_2^{\mathrm{S}}) : \tau_1 \times \tau_2} \qquad \frac{\begin{array}{c} v_{\mathrm{I}} = \mathbf{rec}\, f(x_{\mathrm{I}}).e_{\mathrm{I}} \qquad v_{\mathrm{S}} = \mathbf{rec}\, f(x_{\mathrm{S}}).e_{\mathrm{S}} \\ \Phi, x_{\mathrm{I}} \preceq^{\mathcal{V}} x_{\mathrm{S}} : \tau \vdash \triangleright e_{\mathrm{I}}[v_{\mathrm{I}}/f] \preceq^{\mathcal{E}} e_{\mathrm{S}}[v_{\mathrm{S}}/f] : \tau' \end{array}}{\Phi \vdash v_{\mathrm{I}} \preceq^{\mathcal{V}} v_{\mathrm{S}} : \tau \to \tau'}$$

$$\frac{\Phi \vdash \triangleright e_{\mathrm{I}} \preceq^{\mathcal{E}} e_{\mathrm{S}} : \tau}{\Phi \vdash \Lambda.e_{\mathrm{I}} \preceq^{\mathcal{V}} \Lambda.e_{\mathrm{S}} : \forall \alpha.\tau} \qquad \frac{\Phi \vdash v_{\mathrm{I}} \preceq^{\mathcal{V}} v_{\mathrm{S}} : \tau[\mu\alpha.\tau/\alpha]}{\Phi \vdash v_{\mathrm{I}} \preceq^{\mathcal{V}} v_{\mathrm{S}} : \mu\alpha.\tau} \qquad \Phi \vdash \mathbf{null} \preceq^{\mathcal{V}} \mathbf{null} : \mathbf{ref}_?(\overline{\tau}) \qquad \frac{\Phi \vdash \triangleright v_{\mathrm{I}} \preceq^{\mathcal{V}} v_{\mathrm{S}} : \mathbf{ref}(\overline{\tau})}{\Phi \vdash v_{\mathrm{I}} \preceq^{\mathcal{V}} v_{\mathrm{S}} : \mathbf{ref}_?(\overline{\tau})}$$

$$\frac{\Phi \vdash \mathrm{inv}\left(\exists \overline{x}, \overline{y}.\ \bigwedge \overline{x \preceq^{\mathcal{V}} y : \tau} \wedge v_{\mathrm{I}} \mapsto_{\mathrm{I}} (\overline{x}) * v_{\mathrm{S}} \mapsto_{\mathrm{S}} (\overline{y})\right)}{\Phi \vdash v_{\mathrm{I}} \preceq^{\mathcal{V}} v_{\mathrm{S}} : \mathbf{ref}(\overline{\tau})} \qquad \frac{\Phi \vdash \exists x, y.\ \triangleright x \preceq^{\mathcal{V}} y : \tau_i \wedge \mathrm{inv}(v_{\mathrm{I}} \mapsto_{\mathrm{I}} \mathbf{inj}_i\ x * v_{\mathrm{S}} \mapsto_{\mathrm{S}} \mathbf{inj}_i\ y)}{\Phi \vdash v_{\mathrm{I}} \preceq^{\mathcal{V}} v_{\mathrm{S}} : \tau_1 + \tau_2}$$

▸ "Glue" (logical and structural) rules for concurrent Hoare logic.

Bind
$$\frac{\{P\}\ e\ \{x.\ Q\} \qquad \forall x.\ \{Q\}\ K[x]\ \{y.\ R\}}{\{P\}\ K[e]\ \{y.\ R\}}$$

Return
$$\frac{}{\{\mathsf{emp}\}\ v\ \{x.\ x = v \land \mathsf{emp}\}}$$

Consequence
$$\frac{P \vdash P' \qquad \{P'\}\ e\ \{x.\ Q'\} \qquad Q' \vdash Q}{\{P\}\ e\ \{x.\ Q\}}$$

Disjunction
$$\frac{\{P_1\}\ e\ \{x.\ Q\} \qquad \{P_2\}\ e\ \{x.\ Q\}}{\{P_1 \lor P_2\}\ e\ \{x.\ Q\}}$$

Frame
$$\frac{\{P\}\ e\ \{x.\ Q\}}{\{P * R\}\ e\ \{x.\ Q * R\}}$$

Hypo
$$\frac{\Phi \vdash P \qquad \{P \land Q\}\ e\ \{x.\ R\}}{\Phi \vdash \{Q\}\ e\ \{x.\ R\}}$$

HypoOut
$$\frac{P\ \text{pure} \qquad \Phi, P \vdash \{Q\}\ e\ \{x.\ R\}}{\Phi \vdash \{P \land Q\}\ e\ \{x.\ R\}}$$

▸ Primitive rules for concurrent Hoare logic.

Pure
$$\frac{e \overset{\text{pure}}{\hookrightarrow} e' \qquad \{P\}\ e'\ \{Q\}}{\{\triangleright P\}\ e\ \{Q\}}$$

Private
$$\frac{(\!|P|\!)\ a\ (\!|x.\ Q|\!)}{\{\triangleright P\}\ a\ \{x.\ Q\}}$$

Shared
$$\frac{\forall \iota \overset{\text{rely}}{\supseteq} \iota_0.\ \exists \iota' \overset{\text{guar}}{\supseteq} \iota.\ \exists Q.\ \ (\!|\iota.I(\iota.s) * P|\!)\ a\ (\!|x.\ \triangleright \iota'.I(\iota'.s) * Q|\!) \ \land\ (\underline{i} \mapsto \iota' * Q) \vdash R}{\{\underline{i} \mapsto \iota_0 * \triangleright P\}\ a\ \{x.\ R\}}$$

NewIsland
$$\frac{\{P\}\ e\ \{x.\ Q * \triangleright \iota.I(\iota.s)\}}{\{P\}\ e\ \{x.\ Q * \iota\}}$$

PrivateSumElim
$$\frac{i \in \{1,2\} \qquad \{\underline{\ell} \mapsto_{\mathrm{I}} \mathsf{inj}_i\ x * P\}\ e_i\ \{\mathsf{ret}.\ Q\}}{\{\underline{\ell} \mapsto_{\mathrm{I}} \mathsf{inj}_i\ x * \triangleright P\}\ \mathsf{case}(\ell, \mathsf{inj}_1\ x \Rightarrow e_1, \mathsf{inj}_2\ x \Rightarrow e_2)\ \{\mathsf{ret}.\ Q\}}$$

ExecSpec
$$\frac{\{P\}\ e\ \{x.\ Q\} \qquad Q \rightrightarrows R}{\{P\}\ e\ \{x.\ R\}}$$

▸ Reasoning about atomic expressions.

Inject
$$(\!|\mathsf{emp}|\!)\ \mathsf{inj}_i\ v\ (\!|\mathsf{ret}.\ \mathsf{ret} \mapsto_{\mathrm{I}} \mathsf{inj}_i\ v|\!)$$

Alloc
$$(\!|\mathsf{emp}|\!)\ \mathsf{new}\ \overline{v}\ (\!|\mathsf{ret}.\ \mathsf{ret} \mapsto_{\mathrm{I}} (\overline{v})|\!)$$

Deref
$$(\!|v \mapsto_{\mathrm{I}} (\overline{v})|\!)\ \mathsf{get}(v[i])\ (\!|\mathsf{ret}.\ \mathsf{ret} = v_i \land v \mapsto_{\mathrm{I}} (\overline{v})|\!)$$

Assign
$$(\!|v \mapsto_{\mathrm{I}} (v_1, \ldots, v_n)|\!)\ v[i] \ :=\ v_i'\ (\!|\mathsf{ret}.\ \mathsf{ret} = () \land v \mapsto_{\mathrm{I}} (v_1, \ldots, v_{i-1}, v_i', v_{i+1}, \ldots, v_n)|\!)$$

CASTrue
$$(\!|v \mapsto_{\mathrm{I}} (v_1, \ldots, v_n)|\!)\ \mathsf{cas}(v[i], v_i, v_i')\ (\!|\mathsf{ret}.\ \mathsf{ret} = \mathsf{true} \land v \mapsto_{\mathrm{I}} (v_1, \ldots, v_{i-1}, v_i', v_{i+1}, \ldots, v_n)|\!)$$

CASFalse
$$(\!|v \mapsto_{\mathrm{I}} (\overline{v}) \land v_o \neq v_i|\!)\ \mathsf{cas}(v[i], v_o, v_i')\ (\!|\mathsf{ret}.\ \mathsf{ret} = \mathsf{false} \land v \mapsto_{\mathrm{I}} (\overline{v})|\!)$$

▸ Logical and structural rules.

AConsequence
$$\frac{P \vdash P' \qquad (\!|P'|\!)\ a\ (\!|x.\ Q'|\!) \qquad Q' \vdash Q}{(\!|P|\!)\ a\ (\!|x.\ Q|\!)}$$

AFrame
$$\frac{(\!|P|\!)\ a\ (\!|x.\ Q|\!)}{(\!|P * R|\!)\ a\ (\!|x.\ Q * R|\!)}$$

ADisjunction
$$\frac{(\!|P_1|\!)\ a\ (\!|x.\ Q|\!) \qquad (\!|P_2|\!)\ a\ (\!|x.\ Q|\!)}{(\!|P_1 \lor P_2|\!)\ a\ (\!|x.\ Q|\!)}$$

AExecSpec
$$\frac{(\!|P|\!)\ a\ (\!|x.\ Q|\!) \qquad Q \rightrightarrows R}{(\!|P|\!)\ a\ (\!|x.\ R|\!)}$$

▸ Derived rules.

$$\frac{\Phi \vdash \{P\}\ e\ \{x.\ \triangleright Q\} \qquad \Phi \vdash \forall x.\ \{Q\}\ e'\ \{y.\ R\}}{\Phi \vdash \{P\}\ \mathbf{let}\ x = e\ \mathbf{in}\ e'\ \{y.\ R\}}$$

$$\frac{\Phi \vdash \{P\}\ e\ \{x.\ (x = \mathbf{true} \land \triangleright Q_1) \lor (x = \mathbf{false} \land \triangleright Q_2)\} \qquad \Phi \vdash \{Q_1\}\ e_1\ \{\mathsf{ret}.\ R\} \qquad \Phi \vdash \{Q_2\}\ e_2\ \{\mathsf{ret}.\ R\}}{\Phi \vdash \{P\}\ \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\ \{\mathsf{ret}.\ R\}}$$

$$\frac{\begin{array}{c}\Phi \vdash \{P\}\ e\ \{x.\ (x = \mathbf{null} \land \triangleright Q_1) \lor (\exists \ell.\ x = \ell \land \triangleright Q_2)\} \\ \Phi \vdash \{Q_1\}\ e_1\ \{\mathsf{ret}.\ R\} \qquad \Phi \vdash \forall \ell.\ \{Q_2[\ell/x]\}\ e_2\ \{\mathsf{ret}.\ R\}\end{array}}{\Phi \vdash \{P\}\ \mathbf{case}(e, \mathbf{null} \Rightarrow e_1, x \Rightarrow e_2)\ \{\mathsf{ret}.\ R\}}$$

UnfoldRec

$$\frac{\Phi \vdash \forall f, x.\ \{P \land \forall x.\{P\}\ f\ x\ \{\mathsf{ret}.\ Q\}\}\ e\ \{\mathsf{ret}.\ Q\}}{\Phi \vdash \forall x.\ \{P\}\ e[\mathbf{rec}\ f(x).e/f]\ \{\mathsf{ret}.\ Q\}}$$

C

Metatheory for the logic of local protocols

C.1 BASIC PROPERTIES OF THE LOGIC OF LOCAL PROTOCOLS

Lemma 6 (Rely-guarantee Preorders)**.** The relations $\overset{\text{rely}}{\sqsubseteq}$ and $\overset{\text{guar}}{\sqsubseteq}$ are preorders.

Lemma 7 (Rely-closure of Assertions)**.** $W, \eta \vDash^{\rho} P$ and $W' \overset{\text{rely}}{\sqsupseteq} W$ implies $W', \eta \vDash^{\rho} P$.

Lemma 8. $|W| \otimes W = W$.

Lemma 9. If $W \overset{\text{rely}}{\sqsubseteq} W'$ then $|W| \overset{\text{rely}}{\sqsubseteq} |W'|$.

Lemma 10 (Rely Decomposition)**.** If $W_1 \otimes W_2 \overset{\text{rely}}{\sqsubseteq} W'$ then there are W_1' and W_2' with $W' = W_1' \otimes W_2'$, $W_1 \overset{\text{rely}}{\sqsubseteq} W_1'$ and $W_2 \overset{\text{rely}}{\sqsubseteq} W_2'$.

Lemma 11 (Token Framing)**.** If $W \overset{\text{guar}}{\sqsubseteq} W'$ and $W \otimes W_f$ is defined then there exists some $W_f' \overset{\text{rely}}{\sqsupseteq} W_f$ such that $W' \otimes W_f'$ is defined and $W \otimes W_f \overset{\text{guar}}{\sqsubseteq} W' \otimes W_f'$.

Lemma 12. If $h, \Sigma : W, \eta$ then $h, \Sigma : W \otimes W', \eta$.

Lemma 13. If $h, \Sigma : W \otimes W', \eta$ then $h, \Sigma : W, \eta$.

Lemma 14. If $W.k > 0$ then $\triangleright W \overset{\text{guar}}{\sqsupseteq} W$ and $\triangleright W \overset{\text{rely}}{\sqsupseteq} W$.

Lemma 15. If $W.k > 0$ then $|\triangleright W| = \triangleright |W|$.

Lemma 16 (Later Satisfaction)**.** If $W.k > 0$ and $h, \Sigma : W, \eta$ then $h, \Sigma : \triangleright W, \eta$.

Lemma 17. $\rightrightarrows$ is transitive.

C.2 SOUNDNESS OF HOARE-STYLE REASONING

C.2.1 *Constructions with Threadpool Triples*

Lemma 18 (Framing). $W, \eta \vDash^{\rho} T@m \left\{x.\ Q\right\}$ and $W_f, \eta_f \vDash^{\rho} R$ with $W \# W_f$, $\eta \# \eta_f$ gives

$$W \otimes W_f, \eta \otimes \eta_f \vDash^{\rho} T@m \left\{x.\ Q * R\right\}.$$

Proof. The proof proceeds by induction on $W.k$.

Case $\boxed{W.k = 0}$

1. $(W \otimes W_f).k = W.k = 0$.

Case $\boxed{W.k > 0}$

2. Let $W' \overset{\text{rely}}{\sqsupseteq} W \otimes W_f, \eta'_f \# \eta \otimes \eta_f$.
3. Write $W' = W'_1 \otimes W'_2$ with $W'_1 \overset{\text{rely}}{\sqsupseteq} W, W'_2 \overset{\text{rely}}{\sqsupseteq} W_f$ by Lem. 10.
4. Suppose $(W'_1 \otimes W'_2).k > 0$.
5. $W'_1.k = (W'_1 \otimes W'_2).k > 0$.
6. Suppose $h, \Sigma : W'_1 \otimes W'_2, \eta \otimes \eta_f \otimes \eta'_f$.
7. $h, \Sigma : W'_1, \eta \otimes \eta_f \otimes \eta'_f$ by Lem. 13.

Case $\boxed{h; T \to h'; T'}$

8. Pick Σ', η', and W''_1 with by assumption.
   $W''_1 \overset{\text{guar}}{\sqsupseteq} W'_1$,
   $\Sigma \rightrightarrows \Sigma'$,
   $h', \Sigma' : W''_1, \eta' \otimes \eta_f \otimes \eta'_f$,
   $W''_1.k = W'_1.k - 1$,
   $W''_1, \eta' \vDash^{\rho} T'@m \left\{x.\ Q\right\}$
9. Pick $W''_2 \overset{\text{rely}}{\sqsupseteq} W'_2$ with $W''_1 \otimes W''_2 \overset{\text{guar}}{\sqsupseteq} W'_1 \otimes W'_2$ by Lem. 11.
10. $h', \Sigma' : W''_1 \otimes W''_2, \eta' \otimes \eta_f \otimes \eta'_f$ by Lem. 12.
11. $(W''_1 \otimes W''_2).k = W''_1.k = W'_1.k - 1 = (W'_1 \otimes W'_2).k - 1$.
12. $W''_2, \eta_f \vDash^{\rho} R$.
13. $W''_1 \otimes W''_2, \eta' \otimes \eta_f \vDash^{\rho} T'@m \left\{x.\ Q * R\right\}$ by induction hypothesis.

Case $\boxed{m = i \text{ and } T = T_0 \uplus [i \mapsto v]}$

14. Pick Σ', η', and W_1'' with — by assumption.
    $W_1'' \overset{\text{guar}}{\sqsupseteq} W_1'$,
    $\Sigma \rightrightarrows \Sigma'$,
    $h, \Sigma' : W_1'', \eta' \otimes \eta_f \otimes \eta_f'$,
    $W_1''.k = W_1'.k$,
    $W_1'', \eta' \vDash^\rho Q[v/x] * T_0@\mathsf{none}\left\{x.\ \mathsf{tt}\right\}$
15. Pick $W_2'' \overset{\text{rely}}{\sqsupseteq} W_2'$ with $W_1'' \otimes W_2'' \overset{\text{guar}}{\sqsupseteq} W_1' \otimes W_2'$ — by Lem. 11.
16. $h', \Sigma' : W_1'' \otimes W_2'', \eta' \otimes \eta_f \otimes \eta_f'$ — by Lem. 12.
17. $(W_1'' \otimes W_2'').k = W_1''.k = W_1'.k = (W_1' \otimes W_2').k$.
18. $W_2'', \eta_f \vDash^\rho R$.
19. $W_1'' \otimes W_2'', \eta' \otimes \eta_f \vDash^\rho Q[v/x] * R * T_0@\mathsf{none}\left\{x.\ \mathsf{tt}\right\}$.

□

Corollary 1 (Precondition Extension). $W, \eta \vDash^\rho T@m\left\{x_1.\ \exists x_2.\ x_1 \preceq^{\mathcal{V}} x_2 : \tau \wedge j \rightarrowtail_{\mathrm{s}} K[x_2]\right\}$ together with $W_f \# W$ gives $W \otimes W_f, \eta \vDash^\rho T@m\left\{x_1.\ \exists x_2.\ x_1 \preceq^{\mathcal{V}} x_2 : \tau \wedge j \rightarrowtail_{\mathrm{s}} K[x_2]\right\}$.

For the proofs below, we add an additional syntactic assertion, $\cdot \overset{\text{rely}}{\sqsupseteq} U$, with

$$U \vDash^\rho \cdot \overset{\text{rely}}{\sqsupseteq} U_0 \triangleq U \overset{\text{rely}}{\sqsupseteq} U_0$$

Corollary 2 (Postcondition Strengthening). If we have $W, \eta \vDash^\rho T@m\left\{x.\ Q\right\}$ then $W, \eta \vDash^\rho T@m\left\{x.\ Q \wedge \cdot \overset{\text{rely}}{\sqsupseteq} |W|\right\}$ holds too.

Lemma 19 (Parallel Composition)**.** If we have $W_1, \eta_1 \vDash^\rho T_1@m_1 \left\{x.\ Q_1\right\}$ and $W_2, \eta_2 \vDash^\rho T_2@m_2 \left\{x.\ Q_2\right\}$ with $W_1\#W_2$, $\eta_1\#\eta_2$, $T_1\#T_2$ and $m_1 \neq \mathsf{none} \Rightarrow m_1 \in \mathrm{dom}(T_1)$ then we also have

$$W_1 \otimes W_2, \eta_1 \otimes \eta_2 \vDash^\rho T_1 \uplus T_2@m_1 \left\{x.\ Q_1\right\}.$$

Proof. By induction on the measure $M(W_1, m_1)$ defined by

$$M(W, m) = \begin{cases} W.k & m = \mathsf{none} \\ W.k + 1 & m \neq \mathsf{none}. \end{cases}$$

Case $\boxed{M(W_1, m_1) = 0}$

1. $(W_1 \otimes W_2).k = W_1.k = 0$.

Case $\boxed{M(W_1, m_1) > 0}$

2. Let $W' \overset{\text{rely}}{\sqsupseteq} W_1 \otimes W_2, \eta_f\#\eta_1 \otimes \eta_2$.
3. Write $W' = W_1' \otimes W_2'$ with $W_1' \overset{\text{rely}}{\sqsupseteq} W_1, W_2' \overset{\text{rely}}{\sqsupseteq} W_2$ by Lem. 10.
4. Suppose $(W_1' \otimes W_2').k > 0$.
5. $W_1'.k = (W_1' \otimes W_2').k > 0$.
6. Suppose $h, \Sigma : W_1' \otimes W_2', \eta_1 \otimes \eta_2 \otimes \eta_f$.
7. $h, \Sigma : W_1', \eta_1 \otimes \eta_2 \otimes \eta_f$ by Lem. 13.

Case $\boxed{h; T_1 \uplus T_2 \to h'; T'}$

8. Write $T' = T_1' \uplus T_2$ WLOG.
9. $h; T_1 \to h'; T_1'$ by nondeterminism of fork.
10. Pick Σ', η_1', and W_1'' with by assumption.
    $W_1'' \overset{\text{guar}}{\sqsupseteq} W_1'$,
    $\Sigma \rightrightarrows \Sigma'$,
    $W_1''.k = W_1'.k - 1$,
    $h', \Sigma' : W_1'', \eta_1' \otimes \eta_2 \otimes \eta_f$,
    $W_1'', \eta_1' \vDash^\rho T_1'@m_1 \left\{x_1.\ Q_1\right\}$
11. Pick $W_2'' \overset{\text{rely}}{\sqsupseteq} W_2'$ with $W_1'' \otimes W_2'' \overset{\text{guar}}{\sqsupseteq} W_1' \otimes W_2'$ by Lem. 11.
12. $(W_1'' \otimes W_2'').k = W_1''.k = W_1'.k - 1 = (W_1' \otimes W_2').k - 1$.
13. $h', \Sigma' : W_1'' \otimes W_2'', \eta_1' \otimes \eta_2 \otimes \eta_f$ by Lem. 12.
14. $W_2'', \eta_2 \vDash^\rho T_2@m_2 \left\{x_2.\ Q_2\right\}$ by assumption.
15. $W_1'' \otimes W_2'', \eta_1' \otimes \eta_2 \vDash^\rho T_1' \uplus T_2@m_1 \left\{x_1.\ Q_1\right\}$ by induction hypothesis.

Case $\boxed{T_1 * T_2 = T_0 \uplus [m_1 \mapsto v_1]}$

16. $m_1 \in \mathrm{dom}(T_1)$ — by assumption.
17. Write $T_1 = T_1' \uplus [m_1 \mapsto v_1]$.
18. Pick Σ', η_1', and W_1'' with — by assumption.
    $W_1'' \overset{\text{guar}}{\sqsupseteq} W_1'$,
    $\Sigma \rightrightarrows \Sigma'$,
    $W_1''.k = W_1'.k$,
    $h, \Sigma' : W_1'', \eta_1' \otimes \eta_2 \otimes \eta_f$,
    $W_1'', \eta_1' \vDash^{\rho} Q_1[v_1/x_1] * T_1'@\mathsf{none}\ \{x_1.\ \mathsf{tt}\}$
19. Pick $W_2'' \overset{\text{rely}}{\sqsupseteq} W_2'$ with $W_1'' * W_2'' \overset{\text{guar}}{\sqsupseteq} W_1' * W_2'$ — by Lem. 11.
20. $(W_1'' * W_2'').k = W_1''.k = W_1'.k = (W_1' * W_2').k$.
21. $h, \Sigma' : W_1'' * W_2'', \eta_1' * \eta_2 * \eta_f$ — by Lem. 12.
22. $W_2'', \eta_2 \vDash^{\rho} T_2@m_2\ \{x_2.\ Q_2\}$. — by assumption.
23. $W_1'' * W_2'', \eta_1' * \eta_2 \vDash^{\rho} Q[v_1/x_1] * T_1' \uplus T_2@\mathsf{none}\ \{x_1.\ \mathsf{tt}\}$. — by induction hypothesis.

□

Lemma 20 (Sequential Composition)**.** If we have $W, \eta \vDash^{\rho} [i \mapsto e] \uplus T@i \{x.\ Q\}$ and for all v and any W', η' with $W', \eta' \vDash Q[v/x]$ we have $W', \eta' \vDash^{\rho} [i \mapsto K[v]]@i \{x.\ R\}$ then

$$W, \eta \vDash^{\rho} [i \mapsto K[e]] \uplus T@i \{x.\ R\}.$$

Proof. The proof proceeds by induction on $W.k$; the case $W.k = 0$ is trivial so we assume $W.k > 0$. We branch on the structure of e:

1. Let $W' \overset{\text{rely}}{\sqsupseteq} W, \eta_f \# \eta$.
2. Suppose $W'.k > 0$.
3. Suppose $h, \Sigma : W', \eta \otimes \eta_f$.

Case $\boxed{e = v}$

4. Pick Σ', η', and W'' with — by assumption.
   $W'' \overset{\text{guar}}{\sqsupseteq} W'$,
   $\Sigma \rightrightarrows \Sigma'$,
   $h, \Sigma' : W'', \eta' \otimes \eta_f$,
   $W''.k = W'.k$,
   $W'', \eta' \vDash^{\rho} Q[v/x] * T@\mathsf{none} \{x.\ \mathsf{tt}\}$
5. Write $W'' = W_1'' \otimes W_2''$ and $\eta' = \eta_1' \otimes \eta_2'$. with
   $W_1'', \eta_1' \vDash^{\rho} Q[v/x]$,
   $W_2'', \eta_2' \vDash^{\rho} T@\mathsf{none} \{x.\ \mathsf{tt}\}$.
6. $W_1'', \eta_1' \vDash^{\rho} [i \mapsto K[v]]@i \{x.\ R\}$ — by assumption.
7. $W'', \eta' \vDash^{\rho} [i \mapsto K[v]] \uplus T@i \{x.\ R\}$ — by Lem. 19.

Case $\boxed{K[v] = v'}$

8. Pick Σ'', η'', and W''' with — by (7).
   $W''' \overset{\text{guar}}{\sqsupseteq} W''$,
   $\Sigma' \rightrightarrows \Sigma''$,
   $h, \Sigma'' : W''', \eta'' \otimes \eta_f$,
   $W'''.k = W''.k$,
   $W''', \eta'' \vDash^{\rho} R[v'/x] * T@\mathsf{none} \{x.\ \mathsf{tt}\}$
9. $\Sigma \rightrightarrows \Sigma''$.
10. $W'''.k = W''.k = W'.k$.

Case $\boxed{h; [i \mapsto K[v]] \uplus T \to h'; T'}$

11. Pick Σ'', η'', and W''' with — by (7).
    $W''' \overset{\text{guar}}{\sqsupseteq} W''$,
    $\Sigma' \rightrightarrows \Sigma''$,
    $h', \Sigma'' : W''', \eta'' \otimes \eta_f$,
    $W'''.k = W''.k - 1$,
    $W''', \eta'' \vDash^{\rho} T'@i \left\{x.\ R\right\}$
12. $\Sigma \rightrightarrows \Sigma''$.
13. $W'''.k = W''.k - 1 = W'.k - 1$.

Case $\boxed{e \neq v}$

14. Suppose $h; [i \mapsto K[e]] \uplus T \to h'; [i \mapsto K[e']] \uplus T'$.
15. $h; [i \mapsto e] \uplus T \to h'; [i \mapsto e'] \uplus T'$.
16. Pick Σ', η', and W'' with — by assumption.
    $W'' \overset{\text{guar}}{\sqsupseteq} W'$,
    $\Sigma \rightrightarrows \Sigma'$,
    $h', \Sigma' : W'', \eta' \otimes \eta_f$,
    $W''.k = W'.k - 1$,
    $W'', \eta' \vDash^{\rho} [i \mapsto e'] \uplus T'@i \left\{x.\ Q\right\}$
17. $W'', \eta' \vDash^{\rho} [i \mapsto K[e']] \uplus T'@i \left\{x.\ R\right\}$ — by induction hypothesis.

□

C.2.2 *Soundness of key inference rules*

The soundness of FRAME and BIND follow easily from the lemmas proved in the previous section. Here we give proofs for the key rules dealing with islands and lifting atomic triples.

Lemma 21.

$$\frac{\{P\}\ e\ \{x.\ Q * \triangleright \iota.I(\iota.s)\}}{\{P\}\ e\ \{x.\ Q * \iota\}}\ \textsc{NewIsland}$$

We prove the result by a straightforward induction on the step index in the underlying threadpool simulation, appealing to the following lemma in the case that the main thread terminates: if

$$h, \Sigma : W, \eta \otimes \eta_F \quad \text{and} \quad W, \eta \vDash^{\rho} Q * \triangleright \iota.I(\iota.s) * T@\mathsf{none}\ \{\textbf{true}\}$$

then

$$\exists \eta', W' \overset{\text{guar}}{\sqsupseteq} W.\ h, \Sigma : W', \eta' \otimes \eta_F \quad \text{and} \quad W', \eta' \vDash^{\rho} Q * \iota * T@\mathsf{none}\ \{\textbf{true}\}$$

The proof of the lemma is as follows:
Proof.

1. $W = W_1 \otimes W_2 \otimes W_3, \quad \eta = \eta_1 \otimes \eta_2 \otimes \eta_3,$
   $W_1, \eta_1 \vDash^{\rho} Q, \quad W_2, \eta_2 \vDash^{\rho} \triangleright \iota.I(\iota.s), \quad W_3, \eta_3 \vDash^{\rho} T@\mathsf{none}\ \{\textbf{true}\}$
2. Let $W' = (W.k, W.\omega \uplus [i \mapsto \mathcal{I}[\![\iota]\!]^{\rho}_{k}])$
3. $W' \overset{\text{guar}}{\sqsupseteq} W$
4. $\triangleright |W'|, \eta_2 \vDash^{\rho} \triangleright \iota.I(\iota.s)$
5. Let $\eta' = \eta_1 \otimes \eta_3$
6. $h, \Sigma : W', \eta' \otimes \eta_F$
7. $W_2 \otimes W_3, \eta_3 \vDash^{\rho} T@\mathsf{none}\ \{\textbf{true}\}$ by framing
8. $W', \eta' \vDash^{\rho} Q * \iota * T@\mathsf{none}\ \{\textbf{true}\}$

□

Let

$$\eta \vDash \mathcal{I}[\![W]\!] \quad \triangleq \quad W.K > 0 \ \wedge \ \eta = \overline{\eta_i} \ \wedge \ \forall i \in \mathrm{dom}(W.\omega).\ \eta_i \in \mathrm{interp}(W.\omega(i))(\triangleright|W|)$$

Lemma 22.

$$\textsc{Private}\ \frac{(\!|P|\!)\ a\ (\!|x.\ Q|\!)}{\{\triangleright P\}\ a\ \{x.\ Q\}}$$

Proof.

1. Fix $i, W_0, W, \eta, \eta_F, \rho$
2. Suppose $W_0, \eta \vDash^{\rho} \triangleright P, \quad W \overset{\text{rely}}{\sqsupseteq} W_0, \quad W.k > 0,$
   $\eta_F \# \eta, \quad h, \Sigma : W, \eta \otimes \eta_F, \quad h; [i \mapsto a] \to h'; T$
3. $\exists \eta_W.\quad (h, \Sigma) = \eta \otimes \eta_F \otimes \eta_W, \quad \eta_W \in \mathcal{I}[\![W.\omega]\!]_W$
4. $\exists v.\quad h; a \hookrightarrow h'; v, \quad T = [i \mapsto v]$ by inversion on operational semantics
5. Let $\eta'_F = \eta_F \otimes \eta_W$
6. $\exists \eta' \# \eta'_F.\quad h' = (\eta' \otimes \eta'_F).h, \quad (\eta \otimes \eta'_F).\Sigma \Rightarrow (\eta' \otimes \eta'_F).\Sigma, \quad \triangleright W, \eta' \vDash^{\rho} Q[v/x]$
   by assumption
7. Let $\Sigma' = (\eta' \otimes \eta'_F).\Sigma$
8. $\triangleright W \overset{\text{rely}}{\sqsupseteq} W$
9. $h', \Sigma' : \triangleright W, \eta' \otimes \eta_F$

□

Lemma 23.

SHARED

$$\frac{\forall \iota \overset{\text{rely}}{\sqsupseteq} \iota_0.\ \exists \iota' \overset{\text{guar}}{\sqsupseteq} \iota.\ \exists Q.\ \ (\!|\iota.I(\iota.s) * P|\!)\ a\ (\!|x.\ \triangleright \iota'.I(\iota'.s) * Q|\!) \ \wedge\ (\underline{i} \mapsto \iota' * Q) \vdash R}{\{\underline{i} \mapsto \iota_0 * \triangleright P\}\ a\ \{x.\ R\}}$$

Proof.

1. Fix j and $W_0, \eta \vDash^\rho \iota_0 * \triangleright P$
2. Suffices to show $W_0, \eta \vDash^\rho [i \mapsto e]@i\ \{x.\ R\}$
3. Fix $W \overset{\text{rely}}{\sqsupseteq} W_0$ and $\eta_F \# \eta$
4. Suppose $W.k > 0$ and $h, \Sigma : W, \eta \otimes \eta_F$
5. $W, \eta \vDash^\rho \iota_0 * \triangleright P$
6. $W = W' \otimes W'', \quad \eta = \eta' \otimes \eta'', \quad W', \eta' \vDash^\rho \iota_0, \quad W'', \eta'' \vDash^\rho \triangleright P$
7. $\exists \iota \overset{\text{rely}}{\sqsupseteq} \iota_0, \omega_F.\ W.\omega = \omega_F \uplus [j \mapsto \iota]$
8. $\exists \eta_\iota, \eta'_F.$
   $(h, \Sigma) = \eta_\iota \otimes \eta \otimes \eta_F \otimes \eta'_F$
   $\eta_\iota \in \iota.I(\iota.s)(\triangleright|W|), \quad \eta'_F \in \ast_i \omega_F(i).I(\omega_F(i).s)(\triangleright|W|)$
   by semantics of world satisfaction
9. $\exists \iota' \overset{\text{guar}}{\sqsupseteq} \iota,\ Q.$ by assumption
   $i \mapsto \iota' * Q \vDash^\rho R$
   $(\!|\iota.J(\iota.s) * P|\!)\ e\ (\!|x.\ \triangleright \iota'.J(\iota'.s) * Q|\!)$
10. Let $\widehat{\eta} = \eta'' \otimes \eta_\iota$
11. $W'', \widehat{\eta} \vDash^\rho \triangleright(\iota.J(\iota.s) * P)$ by (1, 8)
12. Let $\widehat{\eta}_F = \eta_F \otimes \eta'_F$
13. $h = (\widehat{\eta} \otimes \widehat{\eta}_F).h$
14. Suppose $h; [j \mapsto a] \to h'; T$
15. $\exists v.\ h; a \hookrightarrow h'; v,\ T = [j \mapsto v]$ by inversion
16. $\exists \widehat{\eta}'.\ h' = (\widehat{\eta}' \otimes \widehat{\eta}_F).h,$ by (9)
    $(\widehat{\eta} \otimes \widehat{\eta}_F).\Sigma \rightrightarrows (\widehat{\eta}' \otimes \widehat{\eta}_F).\Sigma,$
    $\triangleright W'', \widehat{\eta}' \vDash^\rho \triangleright \iota'.J(\iota'.s) * Q[v/x]$
17. Let $\widehat{W} = (W.k - 1, \lfloor \omega_F \rfloor_{W.k-1} \uplus [j \mapsto \iota'])$
18. $\widehat{W} \overset{\text{guar}}{\sqsupseteq} W$
19. $\exists \eta'_\iota, \eta'.\ \widehat{\eta}' = \eta' \otimes \eta'_\iota,$ by semantics of assertions, token-purity of island interpretations
    $\triangleright|\widehat{W}|, \eta'_\iota \vDash i \mapsto \iota'.J(\iota'.s),$
    $\widehat{W}, \eta' \vDash^\rho Q[v/x]$
20. Write $\Sigma' = (\widehat{\eta}' \otimes \widehat{\eta}_F).\Sigma$
21. $\Sigma \rightrightarrows \Sigma'$
22. $h', \Sigma' : \widehat{W}, \eta' \otimes \eta_F$ by (8, 19)

23. $\widehat{W}, \eta' \vDash^{\rho} Q[v/x] * i \mapsto \iota'$
24. $\widehat{W}, \eta' \vDash^{\rho} R[v/x]$ by (9)
25. $\widehat{W}, \eta' \vDash^{\rho} [j \mapsto v]@i \left\{x.\ R\right\}$

□

C.3 SOUNDNESS OF REFINEMENT REASONING

C.3.1 *Congruence*

Lemma 24 (Soundness Shortcut)**.** $\Delta; \Gamma \vDash e_1 \le e_2 : \tau$ is equivalent to

$$
\begin{array}{l}
\forall U\ \forall \rho : \Delta \to \mathrm{VRel}\ \forall \gamma_1, \gamma_2 : \mathrm{dom}(\Gamma) \to \mathrm{Val}. \\
\quad \left[\forall x \in \mathrm{dom}(\Gamma).\ U \vDash^{\rho} \gamma_1(x) \preceq^{\mathcal{V}} \gamma_2(x) : \Gamma(x)\right] \\
\qquad \Longrightarrow \\
\quad \left[\forall K, j, i.\ U, (\varnothing, \{\varnothing; [j \mapsto K[e_2[\gamma_2/\Gamma]]]\}) \vDash^{\rho} \right. \\
\qquad \left. [i \mapsto e_1[\gamma_1/\Gamma]]@i \left\{x_1.\ \exists x_2.\ x_1 \preceq^{\mathcal{V}} x_2 : \tau \wedge j \rightarrowtail_{\mathrm{s}} K[x_2]\right\}\right].
\end{array}
$$

C.3.1.1▸ *New*

Lemma 25.

$$\frac{\Delta; \Gamma \vDash e_i \preceq f_i : \tau_i}{\Delta; \Gamma \vDash \mathsf{new}\ \overline{e} \preceq \mathsf{new}\ \overline{f} : \mathsf{ref}(\overline{\tau})}$$

Proof.

1. Let $U, \rho : \Delta \to \mathrm{VRel}, \gamma_1, \gamma_2 : \mathrm{dom}(\Gamma) \to \mathrm{Val}$.
2. Suppose $\forall x \in \mathrm{dom}(\Gamma).\ U \vDash^{\rho} \gamma_1(x) \preceq^{\mathcal{V}} \gamma_2(x) : \Gamma(x)$.
3. Write $e_i' = e_i[\gamma_1/\Gamma], f_i' = f_i[\gamma_2/\Gamma]$.
4. Let K, i, j.
5. Write $\eta = (\varnothing, \{\varnothing; [j \mapsto K[\mathsf{new}\ \overline{f'}]]\})$.
6. Write $Q = \exists y.\ x \preceq^{\mathcal{V}} y : \mathsf{ref}(\overline{\tau}) \wedge j \rightarrowtail_{\mathrm{s}} K[y]$.
7. Suffices to show $U, \eta \vDash^{\rho} [i \mapsto \mathsf{new}\ \overline{e'}]@i \left\{x.\ Q\right\}$. by Lem. 24.

Let $M = |\mathsf{ref}(\overline{\tau})|$. We now proceed to make a claim: for any $0 \le m \le M$ it sufices to prove

$$U', \eta_m \vDash^{\rho} [i \mapsto \mathsf{new}\ v_1, \ldots, v_m, e'_{m+1}, \ldots, e'_M]@i \left\{x.\ Q\right\},$$

for all $U' \overset{\mathrm{rely}}{\sqsupseteq} U$ and all $U' \vDash^{\rho} v_1 \preceq^{\mathcal{V}} w_1 : \tau_1, \ldots, U' \vDash^{\rho} v_m \preceq^{\mathcal{V}} w_m : \tau_m$, where

$$\eta_m = (\varnothing, \{\varnothing; [j \mapsto K[\mathsf{new}\ w_1, \ldots, w_m, f'_{m+1}, \ldots, f'_M]]\})$$

We prove the claim by induction on m; the case $m = 0$ was proved above. So, suppose the claim holds for $0 \le m < M$ and assume that we know that

$$U'', \eta_{m+1} \vDash^{\rho} [i \mapsto \mathsf{new}\ v_1, \ldots, v_{m+1}, e'_{m+2}, \ldots, e'_M]@i \left\{x.\ Q\right\},$$

holds for all for all $U'' \overset{\mathrm{rely}}{\sqsupseteq} U$ and all $U'' \vDash^{\rho} v_1 \preceq^{\mathcal{V}} w_1 : \tau_1, \ldots, U'' \vDash^{\rho} v_{m+1} \preceq^{\mathcal{V}} w_{m+1} : \tau_{m+1}$, where

$$\eta_{m+1} = (\varnothing, \{\varnothing; [j \mapsto K[\mathsf{new}\ w_1, \ldots, w_{m+1}, f'_{m+2}, \ldots, f'_M]]\})$$

In the interest of applying the induction hypothesis, we pick arbitrary $U' \overset{\mathrm{rely}}{\sqsupseteq} U$ and

$$U' \vDash^{\rho} v_1 \preceq^{\mathcal{V}} w_1 : \tau_1, \ldots, U' \vDash^{\rho} v_m \preceq^{\mathcal{V}} w_m : \tau_m$$

By induction, it will suffice for the claim to prove that

$$U', \eta_m \vDash^{\rho} [i \mapsto \mathsf{new}\ v_1, \ldots, v_m, e'_{m+1}, \ldots, e'_M]@i \left\{x.\ Q\right\},$$

holds, where

$$\eta_m = (\varnothing, \{\varnothing; [j \mapsto K[\mathsf{new}\ w_1, \ldots, w_m, f'_{m+1}, \ldots, f'_M]]\})$$

Now, by assumption and Lemma 24 and Corollary 2 we have

$$U', \eta_m \vDash^{\rho} [i \mapsto e'_{m+1}]@i \left\{x_{m+1}.\ Q_{m+1} \wedge \cdot \overset{\mathrm{rely}}{\sqsupseteq} U'\right\},$$

where

$$Q_{m+1} = \exists y_{m+1}.\ x_{m+1} \preceq^{\mathcal{V}} y_{m+1} : \tau_{m+1} \wedge j \rightarrowtail_{\mathrm{s}} K[\textbf{new}\ w_1, \ldots, w_m, y'_{m+1}, \ldots, f'_M]$$

Now, let v_{m+1} be arbitrary and take $W'', \eta'' \vDash^{\rho} Q_{m+1}[v_{m+1}/x_{m+1}] \wedge \cdot \overset{\text{rely}}{\sqsupseteq} U'$ and by an application of Lemma 20 we have the claim if we can show

$$W'', \eta'' \vDash^{\rho} [i \mapsto \textbf{new}\ v_1, \ldots, v_{m+1}, e'_{m+2}, \ldots, e'_M]@i\ \{x.\ Q\}.$$

Luckily, we can pick w_{n+1} such that we have

$$\begin{aligned}
&|W''| \vDash^{\rho} v_{m+1} \preceq^{\mathcal{V}} w_{m+1} : \tau_{m+1},\\
&\eta'' = (\varnothing, \{\varnothing; [j \mapsto K[\textbf{new}\ w_1, \ldots, w_{m+1}, f'_{m+2}, \ldots, f'_M]]\}),\\
&|W''| \overset{\text{rely}}{\sqsupseteq} U'
\end{aligned}$$

and we can apply our original assumption. After this detour, we proceed with the proof proper:

8. Let $U' \overset{\text{rely}}{\sqsupseteq} U$.
9. Let $\bigwedge \overline{U' \vDash^{\rho} v \preceq^{\mathcal{V}} w : \tau}$.
10. Write $\eta' = (\varnothing, \{\varnothing; [j \mapsto K[\textbf{new}\ \overline{w}]]\})$.
11. Suffices to show $U', \eta' \vDash^{\rho} [i \mapsto \textbf{new}\ \overline{v}]@i\ \{x.\ Q\}$.
12. Let $W' \overset{\text{rely}}{\sqsupseteq} U', \eta_f \# \eta'$.
13. Suppose $W'.k > 0$ and $h, \Sigma : W', \eta' \otimes \eta_f$.
14. $h; [i \mapsto \textbf{new}\ \overline{v}] \to h \uplus [\ell_1 \mapsto \overline{v}]; [i \mapsto \ell_1]$.
15. Pick ℓ_2 with $\forall h_2; T_2 \in \Sigma.\ \ell_2 \notin \mathrm{dom}(h_2)$.
16. Write $\Sigma = \{\varnothing; [j \mapsto K[\textbf{new}\ \overline{w}]]\} \otimes \Sigma_0$.
17. Write $\Sigma' = \{[\ell_2 \mapsto \overline{w}]; [j \mapsto K[\ell_2]]\} \otimes \Sigma_0$.
18. $\Sigma \rightrightarrows \Sigma'$.
19. Write $\eta'' = (\varnothing, \{\varnothing; [j \mapsto K[\ell_2]]\})$.
20. Pick $n \notin \mathrm{dom}(W'.\omega)$.
21. Write $P = \exists \overline{x}, \overline{y}.\ \bigwedge \overline{x \preceq^{\mathcal{V}} y : \tau} \wedge (\ell_1 \mapsto_{\mathrm{I}} (\overline{x}) * \ell_2 \mapsto_{\mathrm{s}} (\overline{y}))$.
22. Write $\iota = ((\{1\}, \varnothing, \varnothing, \lambda\_.\varnothing), [\![\lambda\_.P]\!]^{\rho}_{W'k}, 1, \varnothing)$.
23. Write $W'' = (W'.k, W.\omega \uplus [n \mapsto \iota])$.
24. $|W''| \vDash^{\rho} \ell_1 \preceq^{\mathcal{V}} \ell_2 : \textbf{ref}(\overline{\tau})$.
25. $\triangleright W'' \overset{\text{guar}}{\sqsupseteq} W'$.
26. $\triangleright |W''| \overset{\text{rely}}{\sqsupseteq} \triangleright |W'|$.
27. $\triangleright |W''| \overset{\text{rely}}{\sqsupseteq} U'$.
28. $h \uplus [\ell_1 \mapsto \overline{v}], \Sigma' : W'', \eta'' \otimes \eta_f$.
29. $h \uplus [\ell_1 \mapsto \overline{v}], \Sigma' : \triangleright W'', \eta'' \otimes \eta_f$ by Lem. 16 .
30. $\triangleright W'', \eta'' \vDash^{\rho} [i \mapsto \ell_1]@i\ \{x.\ Q\}$.

□

C.3.1.2▸ *Fork*

Lemma 26.

$$\frac{\Delta; \Gamma \vDash e_1 \preceq e_2 : \mathsf{unit}}{\Delta; \Gamma \vDash \mathsf{fork}\ e_1 \preceq \mathsf{fork}\ e_2 : \mathsf{unit}}$$

Proof.

1. Let $U, \rho : \Delta \to \mathrm{VRel}, \gamma_1, \gamma_2 : \mathrm{dom}(\Gamma) \to \mathrm{Val}$.
2. Suppose $\forall x \in \mathrm{dom}(\Gamma).\ U \vDash^{\rho} \gamma_1(x) \preceq^{\mathcal{V}} \gamma_2(x) : \Gamma(x)$.
3. Write $e_1' = e_1[\gamma_1/\Gamma], e_2' = e_2[\gamma_2/\Gamma]$.
4. Let K, i, j.
5. Write $\eta = (\varnothing, \{\varnothing; [j \mapsto K[\mathsf{fork}\ e_2']]\})$.
6. Write $Q = \exists x_2.\ x_1 \preceq^{\mathcal{V}} x_2 : \mathsf{unit} \wedge j \rightarrowtail_{\mathrm{s}} K[x_2]$.
7. Suffices to show $U, \eta \vDash^{\rho} [i \mapsto \mathsf{fork}\ e_1']@i \{x_1.\ Q\}$. by Lem. 24.
8. Let $W \overset{\text{rely}}{\sqsupseteq} U, \eta_f \# \eta$.
9. Suppose $W.k > 0$ and $h, \Sigma : W, \eta \otimes \eta_f$.
10. $h; [i \mapsto \mathsf{fork}\ e_1'] \to h; [i \mapsto ()] \uplus [i' \mapsto e_1']$.
11. Pick j' with $\forall h'; T' \in \Sigma.\ j' \notin \mathrm{dom}(T')$.
12. Write $\Sigma = \{\varnothing; [j \mapsto K[\mathsf{fork}\ e_2']]\} \otimes \Sigma_0$.
13. Write $\Sigma' = \{\varnothing; [j \mapsto K[()]] \uplus [j' \mapsto e_2']\} \otimes \Sigma_0$.
14. $\Sigma \rightrightarrows \Sigma'$.
15. Write $\eta' = (\varnothing, \{\varnothing; [j \mapsto K[()]]\})$.
16. Write $\eta'' = (\varnothing, \{\varnothing; [j' \mapsto e_2']\})$.
17. $h, \Sigma' : W, \eta' \otimes \eta'' \otimes \eta_f$.
18. $h, \Sigma' : \triangleright W, \eta' \otimes \eta'' \otimes \eta_f$ by Lem. 16.
19. $\triangleright W, \eta' \vDash^{\rho} [i \mapsto ()]@i \{x_1.\ Q\}$.
20. $|\triangleright W|, \eta'' \vDash^{\rho} [i' \mapsto e_1']@i' \{x_1.\ \mathrm{tt}\}$ by assumption and Lem. 24.
21. $\triangleright W, \eta' \otimes \eta'' \vDash^{\rho} [i \mapsto ()] \uplus [i' \mapsto e_1']@i \{x_1.\ Q\}$ by Lem. 19.

□

C.3.1.3▸ *Function Application and Abstraction*

Lemma 27. For $U.k \neq 0$ we have $U \vDash^\rho \textbf{rec}\ f(x).e_1 \preceq^{\mathcal{V}} \textbf{rec}\ f(x).e_2 : \tau_1 \to \tau_2$ equivalent to

$$\begin{aligned}
&\forall w_1, w_2.\ \forall U' \overset{\text{rely}}{\sqsupseteq} \triangleright U.\\
&\quad \left[U' \vDash^\rho w_1 \preceq^{\mathcal{V}} w_2 : \tau_1\right]\\
&\qquad \Longrightarrow\\
&\quad \left[\forall K, j, i.\ U', (\varnothing, \{\varnothing; [j \mapsto K[e_2[\textbf{rec}\ f(x).e_2/f, w_2/x]]]\}) \vDash^\rho\right.\\
&\qquad \left.[i \mapsto e_1[\textbf{rec}\ f(x).e_1/f, w_1/x]]@i\ \left\{x_1.\ \exists x_2.\ x_1 \preceq^{\mathcal{V}} x_2 : \tau_2 \wedge j \rightarrowtail_{\mathrm{s}} K[x_2]\right\}\right].
\end{aligned}$$

Lemma 28.

$$\frac{\Delta; \Gamma \vDash e_1 \preceq e_2 : \tau_1 \to \tau_2 \qquad \Delta; \Gamma \vDash f_1 \preceq f_2 : \tau_1}{\Delta; \Gamma \vDash e_1\ f_1 \preceq e_2\ f_2 : \tau_2}$$

Proof.

1. Let $U, \rho : \Delta \to \mathrm{VRel}, \gamma_1, \gamma_2 : \mathrm{dom}(\Gamma) \to \mathrm{Val}$.
2. Suppose $\forall x \in \mathrm{dom}(\Gamma).\ U \vDash^\rho \gamma_1(x) \preceq^{\mathcal{V}} \gamma_2(x) : \Gamma(x)$.
3. Write $e_1' = e_1[\gamma_1/\Gamma], e_2' = e_2[\gamma_2/\Gamma]$.
4. Write $f_1' = f_1[\gamma_1/\Gamma], f_2' = f_2[\gamma_2/\Gamma]$.
5. Let K, i, j.
6. Write $\eta = (\varnothing, \{\varnothing; [j \mapsto K[e_2'\ f_2']]\})$.
7. Write $Q = \exists x_2.\ x_1 \preceq^{\mathcal{V}} x_2 : \tau_2 \wedge j \rightarrowtail_{\mathrm{s}} K[x_2]$.
8. Suffices to show $U, \eta \vDash^\rho [i \mapsto e_1'\ f_1']@i\ \{x_1.\ Q\}$. by Lem. 24.
9. Write $Q' = \exists x_2'.\ x_1' \preceq^{\mathcal{V}} x_2' : \tau_1 \to \tau_2 \wedge j \rightarrowtail_{\mathrm{s}} K[x_2'\ f_2']$.
10. $U, \eta \vDash^\rho [i \mapsto e_1']@i\ \{x_1'.\ Q'\}$. by assumption and Lem. 24.
11. $U, \eta \vDash^\rho [i \mapsto e_1']@i\ \{x_1'.\ Q' \wedge \cdot \overset{\text{rely}}{\sqsupseteq} U\}$ by Cor. 2.
12. Let $v_1' \in \mathrm{Val}$.
13. Let W', η' with $W', \eta' \vDash^\rho Q'[v_1'/x_1'] \wedge \cdot \overset{\text{rely}}{\sqsupseteq} U$.
14. Suffices to show $W', \eta' \vDash^\rho [i \mapsto v_1'\ f_1']@i\ \{x_1.\ Q\}$ by Lem. 20.
15. Suffices to show $|W'|, \eta' \vDash^\rho [i \mapsto v_1'\ f_1']@i\ \{x_1.\ Q\}$ by Cor. 1.
16. Suppose $W'.k > 0$ WLOG.
17. Pick v_2' with
    $|W'| \vDash^\rho v_1' \preceq^{\mathcal{V}} v_2' : \tau_1 \to \tau_2$,
    $\eta' = (\varnothing, \{\varnothing; [j \mapsto K[v_2'\ f_2']]\})$,
    $|W'| \overset{\text{rely}}{\sqsupseteq} U$.
18. Write $Q'' = \exists x_2''.\ x_1'' \preceq^{\mathcal{V}} x_2'' : \tau_1 \wedge j \rightarrowtail_{\mathrm{s}} K[v_2'\ x_2'']$.
19. $|W'|, \eta' \vDash^\rho [i \mapsto f_1']@i\ \{x_1''.\ Q''\}$. by assumption and Lem. 24.

20. $|W'|, \eta' \vDash^{\rho} [i \mapsto f_1']@i \left\{x_1''.\ Q'' \wedge \cdot \overset{\text{rely}}{\sqsupseteq} |W'|\right\}$ by Cor. 2.
21. Let $v_1'' \in \text{Val}$.
22. Let W'', η'' with $W'', \eta'' \vDash^{\rho} Q''[v_1''/x_1''] \wedge \cdot \overset{\text{rely}}{\sqsupseteq} |W'|$.
23. Suffices to show $W'', \eta'' \vDash^{\rho} [i \mapsto v_1'\ v_1'']@i \left\{x_1.\ Q\right\}$ by Lem. 20.
24. Suffices to show $|W''|, \eta'' \vDash^{\rho} [i \mapsto v_1'\ v_1'']@i \left\{x_1.\ Q\right\}$ by Cor. 1.
25. Suppose $W''.k > 0$ WLOG.
26. Pick v_2'' with
    $|W''| \vDash^{\rho} v_1'' \preceq^{\mathcal{V}} v_2'' : \tau_1$,
    $\eta'' = (\varnothing, \{\varnothing; [j \mapsto K[v_2'\ v_2'']]\})$,
    $|W''| \overset{\text{rely}}{\sqsupseteq} |W'|$.
27. Let $W''' \overset{\text{rely}}{\sqsupseteq} |W''|, \eta_f \# \eta''$.
28. Suppose $W'''.k > 0$ and $h, \Sigma : W''', \eta'' \otimes \eta_f$.
29. Write $v_1' = \textbf{rec}\ f(x).g_1'$ and $v_2' = \textbf{rec}\ f(x).g_2'$.
30. $h; [i \mapsto v_1'\ v_1''] \to h; [i \mapsto g_1'[v_1'/f, v_1''/x]]$.
31. Write $\Sigma = \{\varnothing; [j \mapsto K[v_2'\ v_2'']]\} \otimes \Sigma_0$.
32. Write $\Sigma' = \{\varnothing; [j \mapsto K[g_2'[v_2'/f, v_2''/x]]]\} \otimes \Sigma_0$.
33. $\Sigma \rightrightarrows \Sigma'$.
34. Write $\eta''' = (\varnothing, \{\varnothing; [j \mapsto K[g_2'[v_2'/f, v_2''/x]]]\})$.
35. $h, \Sigma : \triangleright W''', \eta'' \otimes \eta_f$ by Lem. 16.
36. $h, \Sigma' : \triangleright W''', \eta''' \otimes \eta_f$.
37. Suffices to show $\triangleright W''', \eta''' \vDash^{\rho} [i \mapsto g_1'[v_1'/f, v_1''/x]]@i \left\{x_1.\ Q\right\}$.
38. Suffices to show $|\triangleright W'''|, \eta''' \vDash^{\rho} [i \mapsto g_1'[v_1'/f, v_1''/x]]@i \left\{x_1.\ Q\right\}$ by Cor. 1.
39. Suppose $W'''.k > 0$ WLOG.
40. $|W'''| \vDash^{\rho} v_1' \preceq^{\mathcal{V}} v_2' : \tau_1 \to \tau_2$.
41. $|\triangleright W'''| = \triangleright |W'''|$.
42. $|\triangleright W'''| \vDash^{\rho} v_1'' \preceq^{\mathcal{V}} v_2'' : \tau_1$.
43. $|\triangleright W'''|, \eta''' \vDash^{\rho} [i \mapsto g_1'[v_1'/f, v_1''/x]]@i \left\{x_1.\ Q\right\}$ by Lem. 27.

□

Lemma 29.

$$\frac{\Delta; \Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vDash e_1 \preceq e_2 : \tau_2}{\Delta; \Gamma \vDash \mathbf{rec}\ f(x).e_1 \preceq \mathbf{rec}\ f(x).e_2 : \tau_1 \to \tau_2}$$

Proof.

1. Let $U, \rho : \Delta \to \mathrm{VRel}, \gamma_1, \gamma_2 : \mathrm{dom}(\Gamma) \to \mathrm{Val}$.
2. Suppose $\forall x \in \mathrm{dom}(\Gamma).\ U \vDash^\rho \gamma_1(x) \preceq^{\mathcal{V}} \gamma_2(x) : \Gamma(x)$.
3. Write $e_1' = e_1[\gamma_1/\Gamma], e_2' = e_2[\gamma_2/\Gamma]$.
4. Let K, i, j.
5. Write $\eta = (\varnothing, \{\varnothing; [j \mapsto K[\mathbf{rec}\ f(x).e_2']]\})$.
6. Write $Q = \exists x_2.\ x_1 \preceq^{\mathcal{V}} x_2 : \tau_1 \to \tau_2 \wedge j \rightarrowtail_{\mathrm{s}} K[x_2]$.
7. Suffices to show $U, \eta \vDash^\rho [i \mapsto \mathbf{rec}\ f(x).e_1']@i\ \{x_1.\ Q\}$. by Lem. 24.
8. Suffices to show $U, \eta \vDash^\rho Q[\mathbf{rec}\ f(x).e_1'/x_1]$.
9. Suffices to show $U \vDash^\rho \mathbf{rec}\ f(x).e_1' \preceq^{\mathcal{V}} \mathbf{rec}\ f(x).e_2' : \tau_1 \to \tau_2$.
10. Suffices to show $\forall U' \overset{\mathrm{rely}}{\sqsupseteq} U.\ U' \vDash^\rho \mathbf{rec}\ f(x).e_1' \preceq^{\mathcal{V}} \mathbf{rec}\ f(x).e_2' : \tau_1 \to \tau_2$.
11. Proceed by induction on $U'.k$.
12. Suppose $U'.k > 0$.
13. Let $w_1, w_2, U'' \overset{\mathrm{rely}}{\sqsupseteq} \triangleright U'$ with $U'' \vDash^\rho w_1 \preceq^{\mathcal{V}} w_2 : \tau_1$.
14. Let K, j, i.
15. Write $\eta' = (\varnothing, \{\varnothing; [j \mapsto K[e_2'[\mathbf{rec}\ f(x).e_2'/f, w_2/x]]]\})$.
16. Write $Q' = \exists x_2.\ x_1 \preceq^{\mathcal{V}} x_2 : \tau_2 \wedge j \rightarrowtail_{\mathrm{s}} K[x_2]$.
17. Suffices to show $U'', \eta' \vDash^\rho [i \mapsto e_1'[\mathbf{rec}\ f(x).e_1'/f, w_1/x]]@i\ \{x_1.\ Q'\}$ by Lem. 27.
18. $\forall x \in \mathrm{dom}(\Gamma).\ U'' \vDash^\rho \gamma_1(x) \preceq^{\mathcal{V}} \gamma_2(x) : \Gamma(x)$.
19. $U'' \vDash^\rho \mathbf{rec}\ f(x).e_1' \preceq^{\mathcal{V}} \mathbf{rec}\ f(x).e_2' : \tau_1 \to \tau_2$ by induction hypothesis.
20. $U'', \eta' \vDash^\rho [i \mapsto e_1'[\mathbf{rec}\ f(x).e_1'/f, w_1/x]]@i\ \{x_1.\ Q'\}$ by assumption and Lem. 24.

□

C.3.1.4 ▸ *CAS*

Lemma 30. For $U.k \neq 0$ we have that $U \vDash^\rho \ell_1 \preceq^{\mathcal{V}} \ell_2 : \mathbf{ref}(\overline{\tau})$ implies the existence of an $i \in \mathrm{dom}(U.\omega)$ such that we have

$$\mathcal{I}[\![U.\omega(i)]\!]_U = \left\{([\ell_1 \mapsto \overline{v_1}], \{[\ell_2 \mapsto \overline{v_2}]; \varnothing\}) \mid \bigwedge \overline{\triangleright U \vDash^\rho v_1 \preceq^{\mathcal{V}} v_2 : \tau}\right\}.$$

Lemma 31. Assume that we have $U \vDash^\rho v_1 \preceq^{\mathcal{V}} v_2 : \sigma$ and $U \vDash^\rho w_1 \preceq^{\mathcal{V}} w_2 : \sigma$. If $U.k \neq 0$ and there are η, h and Σ such that $h, \Sigma : U, \eta$ holds, then we have that

$$v_1 = w_1 \iff v_2 = w_2.$$

Lemma 32.

$$\frac{\Delta;\Gamma \vDash e_1 \preceq e_2 : \mathsf{ref}(\overline{\tau}) \qquad \tau_n = \sigma \qquad \Delta;\Gamma \vDash f_1 \preceq f_2 : \sigma \qquad \Delta;\Gamma \vDash g_1 \preceq g_2 : \sigma}{\Delta;\Gamma \vDash \mathsf{cas}(e_1[n], f_1, g_1) \preceq \mathsf{cas}(e_2[n], f_2, g_2) : \mathsf{bool}}$$

Proof.

1. Let $U, \rho : \Delta \to \mathrm{VRel}, \gamma_1, \gamma_2 : \mathrm{dom}(\Gamma) \to \mathrm{Val}$.
2. Suppose $\forall x \in \mathrm{dom}(\Gamma).\ U \vDash^{\rho} \gamma_1(x) \preceq^{\mathcal{V}} \gamma_2(x) : \Gamma(x)$.
3. Write $e_1' = e_1[\gamma_1/\Gamma], e_2' = e_2[\gamma_2/\Gamma]$.
4. Write $f_1' = f_1[\gamma_1/\Gamma], f_2' = f_2[\gamma_2/\Gamma]$.
5. Write $g_1' = g_1[\gamma_1/\Gamma], g_2' = g_2[\gamma_2/\Gamma]$.
6. Let K, i, j.
7. Write $\eta = (\varnothing, \{\varnothing; [j \mapsto K[\mathsf{cas}(e_2'[n], f_2', g_2')]]\})$.
8. Write $Q = \exists x_2.\ x_1 \preceq^{\mathcal{V}} x_2 : \mathsf{bool} \wedge j \rightarrowtail_{\mathrm{s}} K[x_2]$.
9. Suffices to show $U, \eta \vDash^{\rho} [i \mapsto \mathsf{cas}(e_1'[n], f_1', g_1')]@i \{x_1.\ Q\}$. by Lem. 24.
10. Write $Q' = \exists x_2'.\ x_1' \preceq^{\mathcal{V}} x_2' : \mathsf{ref}(\overline{\tau}) \wedge j \rightarrowtail_{\mathrm{s}} K[\mathsf{cas}(x_2'[n], f_2', g_2')]$.
11. $U, \eta \vDash^{\rho} [i \mapsto e_1']@i \{x_1'.\ Q'\}$. by assumption and Lem. 24.
12. $U, \eta \vDash^{\rho} [i \mapsto e_1']@i \{x_1'.\ Q' \wedge \cdot \overset{\mathrm{rely}}{\sqsupseteq} U\}$. by Cor. 2.
13. Let $v_1' \in \mathrm{Val}$.
14. Let W', η' with $W', \eta' \vDash^{\rho} Q'[v_1'/x_1'] \wedge \cdot \overset{\mathrm{rely}}{\sqsupseteq} U$.
15. Suffices to show $W', \eta' \vDash^{\rho} [i \mapsto \mathsf{cas}(v_1'[n], f_1', g_1')]@i \{x_1.\ Q\}$ by Lem. 20.
16. Suffices to show $|W'|, \eta' \vDash^{\rho} [i \mapsto \mathsf{cas}(v_1'[n], f_1', g_1')]@i \{x_1.\ Q\}$ by Cor. 1.
17. Suppose $W'.k > 0$ WLOG.
18. Pick v_2' with
    $|W'| \vDash^{\rho} v_1' \preceq^{\mathcal{V}} v_2' : \mathsf{ref}(\overline{\tau})$,
    $\eta' = (\varnothing, \{\varnothing; [j \mapsto K[\mathsf{cas}(v_2'[n], f_2', g_2')]]\})$,
    $|W'| \overset{\mathrm{rely}}{\sqsupseteq} U$.
19. Write $Q'' = \exists x_2''.\ x_1'' \preceq^{\mathcal{V}} x_2'' : \sigma \wedge j \rightarrowtail_{\mathrm{s}} K[\mathsf{cas}(v_2'[n], x_2'', g_2')]$.
20. $|W'|, \eta' \vDash^{\rho} [i \mapsto f_1']@i \{x_1''.\ Q''\}$ by assumption and Lem. 24.
21. $|W'|, \eta' \vDash^{\rho} [i \mapsto f_1']@i \{x_1''.\ Q'' \wedge \cdot \overset{\mathrm{rely}}{\sqsupseteq} |W'|\}$ by Cor. 2.
22. Let $v_1'' \in \mathrm{Val}$.
23. Let W'', η'' with $W'', \eta'' \vDash^{\rho} Q''[v_1''/x_1''] \wedge \cdot \overset{\mathrm{rely}}{\sqsupseteq} |W'|$.
24. Suffices to show $W'', \eta'' \vDash^{\rho} [i \mapsto \mathsf{cas}(v_1'[n], v_1'', g_1')]@i \{x_1.\ Q\}$ by Lem. 20.

25. Suffices to show $|W''|, \eta'' \vDash^{\rho} [i \mapsto \mathbf{cas}(v_1'[n], v_1'', g_1')]@i \{x_1.\ Q\}$ by Cor. 1.
26. Suppose $W''.k > 0$ WLOG.
27. Pick v_2'' with
    $|W''| \vDash^{\rho} v_1'' \preceq^{\mathcal{V}} v_2'' : \sigma$,
    $\eta'' = (\varnothing, \{\varnothing; [j \mapsto K[\mathbf{cas}(v_2'[n], v_2'', g_2')]]\})$,
    $|W''| \overset{\text{rely}}{\sqsupseteq} |W'|$.
28. Write $Q''' = \exists x_2'''.\ x_1''' \preceq^{\mathcal{V}} x_2''' : \sigma \wedge j \rightarrowtail_{\mathrm{s}} K[\mathbf{cas}(v_2'[n], v_2'', x_2''')]$.
29. $|W''|, \eta' \vDash^{\rho} [i \mapsto g_1']@i \{x_1'''.\ Q'''\}$ by assumption and Lem. 24.
30. $|W''|, \eta' \vDash^{\rho} [i \mapsto g_1']@i \{x_1'''.\ Q'' \wedge \cdot \overset{\text{rely}}{\sqsupseteq} |W'|\}$ by Cor. 2.
31. Let $v_1''' \in \mathrm{Val}$.
32. Let W''', η''' with $W''', \eta''' \vDash^{\rho} Q'''[v_1'''/x_1'''] \wedge \cdot \overset{\text{rely}}{\sqsupseteq} |W''|$.
33. Suffices to show $W''', \eta''' \vDash^{\rho} [i \mapsto \mathbf{cas}(v_1'[n], v_1'', v_1''')]@i \{x_1.\ Q\}$ by Lem. 20.
34. Suffices to show $|W'''|, \eta''' \vDash^{\rho} [i \mapsto \mathbf{cas}(v_1'[n], v_1'', v''')]@i \{x_1.\ Q\}$ by Cor. 1.
35. Suppose $W'''.k > 0$ WLOG.
36. Pick v_2''' with
    $|W'''| \vDash^{\rho} v_1''' \preceq^{\mathcal{V}} v_2''' : \sigma$,
    $\eta''' = (\varnothing, \{\varnothing; [j \mapsto K[\mathbf{cas}(v_2'[n], v_2'', v_2''')]]\})$,
    $|W'''| \overset{\text{rely}}{\sqsupseteq} |W''|$.
37. Let $W'''' \overset{\text{rely}}{\sqsupseteq} W''', \eta_f \# \eta'''$.
38. Suppose $W''''.k > 0$ and $h, \Sigma : W'''', \eta''' \otimes \eta_f$.
39. Suppose $h; [i \mapsto \mathbf{cas}(v_1'[n], v_1'', v''')] \to h'; T'$.
40. Suppose $W''''.k > 1$ WLOG.
41. Write $v_1' = \ell_1$ and $v_2' = \ell_2$.
42. Pick $\overline{v}, \overline{w}, h_0, \Sigma_0$ with
    $\bigwedge \overline{\triangleright |W''''| \vDash^{\rho} v \preceq^{\mathcal{V}} w : \tau}$,
    $h = [\ell_1 \mapsto \overline{v}] \uplus h_0$,
    $\Sigma = \{[\ell_2 \mapsto \overline{w}]; \varnothing\} \otimes \{\varnothing; [j \mapsto K[\mathbf{cas}(v_2'[n], v_2'', v_2''')]]\} \otimes \Sigma_0$ by Lem. 30.
43. $\triangleright |W''''| \vDash^{\rho} v_n \preceq^{\mathcal{V}} w_n : \sigma$.
44. $\triangleright |W''''| \vDash^{\rho} v_1'' \preceq^{\mathcal{V}} v_2'' : \sigma$.
45. $h, \Sigma : \triangleright |W''''|, \eta''' \otimes \eta_f$ by Lem. 16.
46. $v_n = v_1'' \Leftrightarrow w_n = v_2''$ by Lem. 31.

Case $\boxed{v_n = v_1'' \wedge w_n = v_2''}$

47. Write $v_n^\dagger = v_1'''$ and $v_m^\dagger = v_m, m \neq n$.
48. $h' = h[\ell_1 \mapsto \overline{v^\dagger}]$.
49. $T' = [i \mapsto \mathsf{true}]$.
50. Write $w_n^\dagger = v_2'''$ and $w_m^\dagger = w_m, m \neq n$.
51. Write $\Sigma' = \{[\ell_2 \mapsto \overline{w^\dagger}]; \varnothing\} \otimes \{\varnothing; [j \mapsto K[\mathsf{true}]]\} \otimes \Sigma_0$.
52. $\Sigma \rightrightarrows \Sigma'$.
53. Write $\eta'''' = (\varnothing, \{\varnothing; [j \mapsto K[\mathsf{true}]]\})$.
54. $\bigwedge \overline{\triangleright |W''''| \vDash^\rho v^\dagger \preceq^{\mathcal{V}} w^\dagger : \tau}$.
55. $h', \Sigma' : W'''', \eta'''' \otimes \eta_f$.
56. $h', \Sigma' : \triangleright W'''', \eta'''' \otimes \eta_f$ by Lem. 16.
57. $\triangleright W'''', \eta'''' \vDash^\rho [i \mapsto \mathsf{true}]@i \{x_1.\ Q\}$.

Case $\boxed{v_n \neq v_1'' \wedge w_n \neq v_2''}$

58. $h' = h$.
59. $T' = [i \mapsto \mathsf{false}]$.
60. Write $\Sigma' = \{[\ell_2 \mapsto \overline{w}]; \varnothing\} \otimes \{\varnothing; [j \mapsto K[\mathsf{false}]]\} \otimes \Sigma_0$.
61. $\Sigma \rightrightarrows \Sigma'$.
62. Write $\eta'''' = (\varnothing, \{\varnothing; [j \mapsto K[\mathsf{false}]]\})$.
63. $h, \Sigma' : W'''', \eta'''' \otimes \eta_f$.
64. $h, \Sigma' : \triangleright W'''', \eta'''' \otimes \eta_f$ by Lem. 16 .
65. $\triangleright W'''', \eta'''' \vDash^\rho [i \mapsto \mathsf{false}]@i \{x_1.\ Q\}$.

□

C.3.2 *May-refinement*

Theorem 4 (May-refinement)**.** Suppose $\cdot;\cdot \vDash e_1 \preceq e_2 : \mathbf{nat}$ holds. Let h_1, h_2, i, j and n be arbitrary. If we have

$$\exists h_1', T_1.\ h_1; [i \mapsto e_1] \to^* h_1'; [i \mapsto n] \uplus T_1$$

then we also have

$$\exists h_2', T_2.\ h_2; [j \mapsto e_2] \to^* h_2'; [j \mapsto n] \uplus T_2.$$

Proof. Let M be the number of steps in the assumed reduction. Write

$$h_1; [i \mapsto e_1] = h_1^0; T_1^0 \to h_1^1; T_1^1 \to \cdots \to h_1^M; T_1^M = h_1'; [i \mapsto n] \uplus T_1.$$

We proceed to prove by induction the claim that for all $0 \le m \le M$ there are W_m, $\eta_m \# \eta$, and Σ_m with the following properties, where $\eta = (h_1, \{h_2; \varnothing\})$ and $\Sigma = \{h_2; [j \mapsto e_2]\}$:

- $W_m, \eta_m \vDash^\rho T_1^m @ i \left\{x_1.\ \exists x_2.\ x_1 \preceq^{\mathcal{V}} x_2 : \mathbf{nat} \wedge j \rightarrowtail_{\mathrm{s}} x_2\right\}$.
- $h_1^m, \Sigma_m : W_m, \eta_m \otimes \eta$.
- $W_m.k = 1 + M - m$.
- $\Sigma \rightrightarrows \Sigma_m$.

Let us initially consider the base case $m = 0$. We choose $W_0 = (1 + M, \varnothing)$, $\eta_0 = (\varnothing, \{\varnothing; [j \mapsto e_2]\})$ and $\Sigma_0 = \{h_2; [j \mapsto e_2]\}$. The different properties are easily verified; the only nontrivial is the first and that follows from the initial assumption $\cdot;\cdot \vDash e_1 \preceq e_2 : \mathbf{nat}$. The induction step comes down to unrolling the definition of threadpool triples; we omit the details.

Instantiating our claim at $m = M$ now gives us W_M, $\eta_M \# \eta$, and Σ_M such that:

- $W_M, \eta_M \vDash^\rho [i \mapsto n] \uplus T_1 @ i \left\{x_1.\ \exists x_2.\ x_1 \preceq^{\mathcal{V}} x_2 : \mathbf{nat} \wedge j \rightarrowtail_{\mathrm{s}} x_2\right\}$.
- $h_1', \Sigma_M : W_M, \eta_M \otimes \eta$.
- $W_M = 1$.
- $\Sigma \rightrightarrows \Sigma_M$.

A final unrolling of the definition of threadpool triples and a few calculations gives us Σ' with $\Sigma \rightrightarrows \Sigma'$ and $\Sigma' = \{\varnothing; [j \mapsto n]\} \otimes \Sigma_0$. All that remains is to pick an element from the nonempty set Σ'. □

D

Reference: the Joins library API

A new `Join` instance *j* is allocated by calling an overload of factory method `Join.Create`:

`Join` *j* = `Join.Create`(); *or*

`Join` *j* = `Join.Create`(*size*);

The optional integer *size* is used to explicitly bound the number of channels supported by `Join` instance *j*. An omitted *size* argument defaults to 32; *size* initializes the constant, read-only property *j*`.Size`.

A `Join` object notionally owns a set *channels*, each obtained by calling an overload of method `Init`, passing the location, *channel*(*s*), of a channel or array of channels using an `out` argument:

j`.Init`(`out` *channel*);

j`.Init`(`out` *channels*, *length*);

The second form takes a *length* argument to initialize location *channels* with an *array* of *length* distinct channels.

Channels are instances of delegate types. In all, the library provides six channel flavors:

```
// void-returning asynchronous channels
delegate void Asynchronous.Channel();
delegate void Asynchronous.Channel<A>(A a);
// void-returning synchronous channels
delegate void Synchronous.Channel();
delegate void Synchronous.Channel<A>(A a);
// value-returning synchronous channels
delegate R Synchronous<R>.Channel();
delegate R Synchronous<R>.Channel<A>(A a);
```

The outer class of a channel `Asynchronous`, `Synchronous` or `Synchronous<`*R*`>` should be read as a modifier that specifies its blocking behaviour and optional return type.

When a synchronous channel is invoked, the caller must wait until the delegate returns (void or some value). When an asynchronous channel is invoked, there is no result and the caller proceeds immediately without waiting. Waiting may, but need not, involve blocking.

Apart from its channels, a `Join` object notionally owns a set of *join patterns*. Each pattern is defined by invoking an overload of the instance method `When`

followed by zero or more invocations of instance method `And` followed by a final invocation of instance method `Do`. Thus a pattern definition typically takes the form:

j`.When(`c_1`).And(`c_2`)`$\cdots$`.And(`c_n`).Do(`d`)`

Each argument c to `When(`c`)` or `And(`c`)` can be a single channel or an array of channels. All synchronous channels that appear in a pattern must *agree* on their return type.

The argument d to `Do(`d`)` is a *continuation* delegate that defines the body of the pattern. Although it varies with the pattern, the type of the continuation is always an instance of one of the following delegate types:

```
delegate R Func<P1, ..., Pm, R>(P1 p1, ..., Pm pm);
delegate void Action<P1, ..., Pm>(P1 p1, ..., Pm pm);
```

The precise type of the continuation d, including its number of arguments, is determined by the sequence of channels guarding it. If the first channel, c_1, in the pattern is a synchronous channel with return type R, then the continuation's return type is R; otherwise the return type is `void`.

The continuation receives the arguments of channel invocations as delegate parameters $P_1\ p_1, \ldots, P_m\ p_m$, for $m \leq n$. The presence and types of any additional parameters $P_1\ p_1, \ldots, P_m\ p_m$ is dictated by the type of each channel c_i:

- If c_i is of non-generic type `Channel` or `Channel[]` then `When(`c_i`)`/`And(`c_i`)` adds no parameter to delegate d.
- If c_i is of generic type `Channel<`P`>`, for some type P then `When(`c_i`)`/`And(`c_i`)` adds one parameter p_j of type $P_j = P$ to delegate d.
- If c_i is an array of type `Channel<`P`>[]` for some type P then `When(`c_i`)`/`And(`c_i`)` adds one parameter p_j of array type $P_j = P[]$ to delegate d.

Parameters are added to d from left to right, in increasing order of i. In the current implementation, a continuation can receive at most $m \leq 16$ parameters.

A join pattern associates a set of channels with a body d. A body can execute only once all the channels guarding it have been invoked. Invoking a channel may enable zero, one or more patterns:

- If no pattern is enabled then the channel invocation is queued up. If the channel is asynchronous, then the argument is added to an internal bag. If the channel is synchronous, then the calling thread is blocked, joining a notional bag of threads waiting on this channel.
- If there is a single enabled join pattern, then the arguments of the invocations involved in the match are consumed, any blocked thread involved in the match is awakened, and the body of the pattern is executed in that thread. Its result—a value or an exception—is broadcast to all other

waiting threads, awakening them. If the pattern contains no synchronous channels, then its body runs in a new thread.

- If there are several enabled patterns, then an unspecified one is chosen to run.
- Similarly, if there are multiple invocations of a particular channel pending, which invocation will be consumed when there is a match is unspecified.

The current number of channels initialized on j is available as read-only property j`.Count`; its value is bounded by j`.Size`. Any invocation of j`.Init` that would cause j`.Count` to exceed j`.Size` throws `JoinException`.

Join patterns must be well-formed, both individually and collectively. Executing `Do`(d) to complete a join pattern will throw `JoinException` if d is **`null`**, the pattern repeats a channel (and the implementation requires linear patterns), a channel is **`null`** or *foreign* to this pattern's `Join` instance, or the join pattern is *empty*. A channel is foreign to a `Join` instance j if it was not allocated by some call to j`.Init`. A pattern is empty when its set of channels is empty (this can only arise through array arguments).

Array patterns are useful for defining dynamically sized joins, e.g. an n-way exchanger:

```
class NWayExchanger<T> {
  public Synchronous<T[]>.Channel<T>[] Values;
  public NWayExchanger(int n) {
    var j = Join.Create(n); j.Init(out Values, n);
    j.When(Values).Do(vs ⇒ vs);
  }
}
```

E

Reference: the Reagents library API

```
// Isolated updates on refs (shared state)
upd:  Ref[A] ⇒ (A × B ⇀ A × C) ⇒ Reagent[B,C]

// Low-level shared state combinators
read: Ref[A] ⇒ Reagent[Unit, A]
cas:  Ref[A] × A × A ⇒ Reagent[Unit, Unit]

// Interaction on channels (message passing)
swap: Endpoint[A,B] ⇒ Reagent[A,B]

// Composition
+  : Reagent[A,B] × Reagent[A,B] ⇒ Reagent[A,B]
>> : Reagent[A,B] × Reagent[B,C] ⇒ Reagent[A,C]
*  : Reagent[A,B] × Reagent[A,C] ⇒ Reagent[A, B × C]

// Liftings
lift:   (A ⇀ B) ⇒ Reagent[A,B]
first:  Reagent[A,B] ⇒ Reagent[A×C, B×C]
second: Reagent[A,B] ⇒ Reagent[C×A, C×B]

// Computed reagents
computed: (A ⇀ Reagent[Unit, B]) ⇒ Reagent[A,B]

// Post-commit actions
postCommit: (A ⇒ Unit) ⇒ Reagent[A,A]

// Invoking a reagent:
dissolve: Reagent[Unit,Unit] ⇒ Unit // as a catalyst
react:    Reagent[A,B] ⇒ A ⇒ B     // as a reactant,
                                   //    same as the ! method
```