Oracle Semantics

Aquinas Hobor

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

Advisor: Andrew W. Appel

November 2008

Abstract

We define a Concurrent Separation Logic with first-class locks and threads for the C language, and prove its soundness in Coq with respect to a compilable operataional semantics.

We define the language Concurrent C minor, an extension of the C minor language of Leroy. C minor was designed as the highest-level intermediate language in the CompCert certified ANSI C compiler, and we add to it lock, unlock, and fork statements to make Concurrent C minor, giving it a standard Pthreads style of concurrency. We define a Concurrent Separation Logic for Concurrent C minor, which extends the original Concurrent Separation Logic of O'Hearn to handle first-class locks and threads.

We then prove the soundness of the logic with respect to the operational semantics of the language. First, we define an erased concurrent operational semantics for Concurrent C minor that is a reasonable abstraction for concurrent execution on a real machine. Second, we define a new modal substructural logic which we use as the model for assertions in Concurrent Separation Logic. Third, we define an unerased

concurrent operational semantics for Concurrent C minor that keeps track of additional bookkeeping to demonstrate the programs are well-behaved. Fourth, we define an oracle semantics that is a thread-local semantics of Concurrent C minor; this allows us to separate the metatheory of sequential and concurrent computation from each other. Fifth, we give a new semantics to the Hoare tuple using our modal substructural logic that connects Concurrent Separation Logic to our oracle semantics and then connect our oracle semantics to our concurrent semantics.

Our soundness proofs are largely implemented in 60,000 lines of Coq; our modular proof design allows us to largely reuse a significant portion of the soundness proofs of the sequential subset of Concurrent Separation Logic developed by Appel and Blazy. Our ability to reuse those proofs gives us confidence that we will be able to modify the CompCert compiler proofs to handle Concurrent C minor in the future.

Acknowledgements

A PhD takes considerable effort, and I am very grateful to the many people who have provided friendship and support during my five years at Princeton. I would like to acknowledge the following people.

My advisor, Andrew Appel, for being extremely generous with his time and energy. I have very much enjoyed learning from and working with him and am very proud of what we have done. It has been fun.

My thesis readers, David Walker and Peter O'Hearn, provided helpful suggestions to improve the text and did a wonderful job reading a long thesis in a short time. I would also like to thank my nonreaders, Brian Kernighan and Vivek Pai, for their support.

My friends near and far, including Ian Adams, Stephen Adams, David Beazley, C.J. Bell, Christian Bienia, Elizabeth Cheehy, Manuvir Das, Rob Dockins, M. and Mme. Dubosc, R.J. Dragani, Gilles Goulas, Carl Haefling and Pam Johnson, Joe Helmer, Mark Kormes, Jason (Wen Di) Li, David MacQueen, Francesco Zappa Nardelli, Travis Pew, Christopher Richards, Ari Schwayder, Tushar and Rupa Shanbhag, Chris Sherman, Colleen Smith, Dan Suberviola, Charlie Trapp, Steve

Wilson, and Daniel Wang.

My in-laws, Steve and Fonde Werts, for being so kind and generous to their son in law. My brother in-law, Charlie Werts, deserves special mention, among other reasons, for his loan of the somewhat incomprehensible yet enjoyable anime series RahXephon. My three grandparents-in-law—Ernie, Joe, and Martha—have been very generous as well.

My family, including my mother, Nancy Hobor; my father, Mike Hobor; my Grandmother, Jane Coulson; and last (but probably not least) my brother, Justinian Hobor. Few things in life give me more pleasure than blowing up his wonders in Civ 4. My Aunt Nancy and Uncle Andy have been very generous in letting me and my wife visit them in Maine, which we have throughly enjoyed. My Aunt Jane and Uncle Jim helped make New Jersey more hospitable. I would also like to thank my cousin George Ellis for letting me stay with him when I visited Rochester.

Finally, my wonderful wife, Lucy Day. We made it!

This research was supported in part by National Science Foundation grant CCF-0540914. In addition, I gratefully acknowledge three summer internships and the people who made them possible: the Program Analysis Group at Microsoft Redmond (Stephen Adams), Lapides Asset Management (Steve Wilson), and the MOSCOVA project at INRIA Rocquencourt (Francesco Zappa Nardelli).

Contents

Abstract iii

Acknowledgements v

Contents vii

List of Figures xii

List of Tables xv

1 Introduction 1

1.1 Concurrency and Multiple Cores 1

1.2 Formal Methods 3

1.3 The Goal of Oracle Semantics 11

1.4 Structure of Thesis 13

2 Related Work 19

2.1 Hoare logic and separation logic 19

2.2 Concurrent Separation Logic 24

2.3 Rely/Guarantee 29

2.4 C minor and CompCert 31
2.5 Separation Logic for C minor 35
2.6 Generation of proofs in CSL 44
2.7 Gotsman et al.'s CSL 45
2.8 Miscellaneous related work 47

3 Concurrent C minor 49
3.1 The Concurrent C minor language 50
3.2 Programming with locks 53

4 Concurrent Separation Logic 59
4.1 Basic assertions 59
4.2 Sequential separation logic assertions 60
4.3 Fractional ownership 61
4.4 Stratified separation algebras 62
4.5 Additional properties for shares 65
4.6 Share models 67
4.7 New assertions for CSL 70
4.8 Concurrent separation logic Hoare rules 75
4.9 Applying CSL to the example program 77
4.10 Conclusions 89

5 Erased Concurrent Operational Semantics 91
5.1 Erased sequential step relation 93
5.2 Erased concurrent step relation 95
5.3 Reasonableness of interleaving model 99

5.4 Conclusion 100

6 Engineering Isolation 101

6.1 Modularization Goals 102

6.2 Shared definitions for glue 106

6.3 An Extensible Semantics 114

6.4 Building an extension 123

6.5 Gluing a core and extension together 127

6.6 Notation for the rest of the thesis 130

7 A Modal Substructural Logic 133

7.1 Modelling difficulties 134

7.2 A substructural modal model 139

7.3 Reasoning about the model with a modal substructural logic 158

7.4 Conclusions 186

8 Concurrent Operational Semantics 187

8.1 Architecture 188

8.2 Sequential submachine 189

8.3 Machine state 193

8.4 Consistent machines 196

8.5 Concurrent step relation 203

8.6 Reasonableness of the step relation 208

8.7 Conclusions 210

9 Oracle Semantics 211

9.1 Why an oracular semantics . . . 211

9.2 Concurrent oracle . . . 214

9.3 Concurrent consult relation . . . 216

9.4 The oracular step . . . 219

9.5 Conclusion . . . 220

10 A Modal Hoare Judgment and Oracular Soundness 221

10.1 A modal Hoare judgment . . . 222

10.2 Hoare judgments in CSL . . . 235

10.3 Soundness of the Oracular Approach . . . 246

10.4 Conclusion . . . 259

11 Conclusions and Future Work 261

11.1 State of the Coq Development . . . 263

11.2 Future work . . . 264

11.3 Concluding thoughts . . . 265

A A Miniature Model in Coq 267

A.1 Headers . . . 267

A.2 Stratified Model . . . 268

A.3 Dependent Model . . . 269

A.4 Private definitions relating public to private . . . 270

A.5 Public interface to acquire resource . . . 271

B Proofs 273

B.1 The unlock rule 273
B.2 Forking a child 276

Bibliography 279

List of Figures

2.1 Leroy's C minor expressions 31
2.2 Leroy's C minor statements 32
2.3 Appel and Blazy's C minor 37
2.4 Models of the Sequential Separation Logic Operators 39
2.5 Continuation-passing style definition of Hoare tuple 41
2.6 Axiomatic Semantics of Separation Logic (without call and return) 42

3.1 Sample concurrent program 54
3.2 Informal description of $R(l)$ 55

4.1 Concurrent Separation Logic 75
4.2 Resource invariant for example program 78

5.1 Erased sequential step relation 94
5.2 Erased concurrent step relation 97

6.1 An extensible language 103
6.2 Shared definitions in all CompCert languages 107

6.3 Axiomatic presentation of resources and resource maps . . . 111
6.4 Interface for extensible semantics . . . 115
6.5 Axioms for Extensible Semantics . . . 119
6.6 Axioms for C minor memory model . . . 121
6.7 Basic definitions for Extensible Semantics . . . 122
6.8 Interface for extension . . . 125
6.9 Coq oracular step relation . . . 128
6.10 Oracular step relation . . . 131

7.1 Naïve assertion model . . . 136
7.2 Sketch of substructural modal model . . . 140
7.3 Increasing approximation . . . 149
7.4 Interface to substructural modal model . . . 156
7.5 Models of logical assertions . . . 161
7.6 Models of modal assertions . . . 163
7.7 Models of substructural assertions . . . 168
7.8 Rules for reasoning in the modal substructural logic . . . 170
7.9 Models of Concurrent Separation Logic assertions . . . 172
7.10 Logical implication in the logic . . . 174
7.11 Extensionality, validity, precision, and tightness . . . 178
7.12 Recursion . . . 180

8.1 Simplified subset of sequential step relation . . . 190
8.2 The consult relation of the sequential submachine . . . 192
8.3 Sequential steps in the concurrent step relation . . . 204

8.4 Fully concurrent steps in the concurrent step relation 205

9.1 The oracle allows for reasoning after a concurrent instruction 212

9.2 Oracular projection 215

9.3 Running the other threads 216

9.4 The oracular consult relation 217

10.1 Oracular safety 223

10.2 Naïve continuation-passing style definition of Hoare tuple . . 225

10.3 A modal Hoare tuple 231

10.4 Building believe 239

10.5 Concurrent safety 246

10.6 Progress invariant 248

10.7 Preservation invariant 251

List of Tables

6.1 Naïve application of glue to compiler 104

6.2 Level independence 106

Chapter 1

Introduction

1.1 Concurrency and Multiple Cores

Since the 1960s, programmers have utilized concurrency to make their programs faster and to better interact with the external world; however, concurrent programs have been notoriously difficult to develop. In the 1970s, Dijkstra and Hoare proposed constructs such as semaphores to handle the resulting difficulties in [Dij68, Hoa74, Hoa78], but even with them writing concurrent software was extremely time-consuming and error-prone. Therefore people avoided writing concurrent programs as long as computer manufacturers were able to relentlessly crank out chips that could run sequential code faster and faster.

Unfortunately, starting in 2003, manufacturers found it difficult to continue to increase the clock speed, due to factors such as heat retention. To continue to improve the processing power of their chips,

manufacturers began to incorporate multiple CPU cores into a single chip. Simply stated, the idea was that a two-core chip would be able to execute as many instructions as a single-core chip executing at twice the clock speed. Since 2003 it appears to be easier to increase the number of cores than to increase the clock speed, multi-core processors are expected to be pervasive in the future.

However, there are significant problems involved in using a multi-core system as opposed to a single-core one. First, even in the best of cases, there is overhead for communication between the multiple cores, so that in fact, even under ideal conditions, a two-core system will perform worse than a single-core system with twice the clock speed. This problem, while annoying, is not fatal. A more severe problem is that to really take advantage of multiple cores, one must write highly concurrent programs.

Since historically programmers avoided concurrency, when computer manufacturers began to incorporate dual-core processors into their consumer product lines in 2005, the second core was noticably underutilized. Manufacturers' plans to offer chips with 256 or more cores seemed unjustified, given the low utilization of the multiple cores. For any large number of cores to be useful, programs will have to utilize substantially more concurrency.

What makes concurrent programs so hard to develop is that the threads of control interact with each other in unpredictable ways. This behavior makes it very hard for programmers to reason about how the

program will behave. Moreover, testing is very difficult, since typically concurrency bugs are not reproduceable due to the interaction with the scheduler, which is usually nondeterministic.

The first techniques to handle the difficulties of concurrent programming were various programming disciplines. The earliest of these were monitors, introduced by Hoare [Hoa74] and Hansen [Han93], and semaphores, introduced by Dijkstra [Dij68]. Other common programming disciplines and techniques include condition variables, channels, CSP, message passing, RPC, rendezvous, etc. These methods are good in practice as long as the discipline is maintained, but in large software systems, maintained over long periods by many different people, they tend to break down.

1.2 Formal Methods

One family of techniques to help programmers write better quality software is formal methods, such as model checking, programming languages, type theories, and verification. As compared to testing, these methods require more theoretical and engineering machinery, but they can provide stronger guarantees, ensuring that programs behave properly even in tricky corner cases.

Each method has strengths and weaknesses. A typical model-checking method enumerates all possible scenirios under a certain size to produce a bound on program behavior. To reach a conclusion in a reasonable

amount of time, a model-checker makes various kinds of simplifying assumptions that are unsound in certain, weakening the guarantees provided. New programming languages can be devised to help programmers write better code, but generally speaking, programmers are very reluctant to adopt new languages, or even new features in old ones. Type systems, in which the types produce a conservative approximation for the result of the program, can have the major advantage of being almost entirely automated (that is, the computer can apply the type system without significant help from the programmer). However, type systems seldom provide the strongest kinds of guarantees. Finally, verification, a technique whose end goal is a formal mathematical proof that a program meets its specification, can produce the most expressive and powerful guarantees about system behavior, but can be very labor-intensive to do.

1.2.1 By Hand or Machine

Reasoning about real code is very complex, frequently requiring dozens of special cases, exceptions, and assumptions to guarantee the desired behavior. Even for purely sequential programs, to apply a given formal method to a computer system can be quite difficult. There are two basic strategies for doing so.

The first strategy is to abstract the "core" of the system, such as the key algorithm, and produce a proof by hand that it meets its specification. If the core is not too complex, then this proof will be trustworthy.

One obvious problem is that features in the core can interact with other "noncore" features in surprising ways, meaning that properties proven to hold for the core might not hold for the whole system. This is a serious concern, although generally people are good at understanding which parts of the system are vital to the proof of the desired property. The real challange lies in implementing the system. Naturally, one takes as much care as possible to implement it correctly and avoid bugs. Unfortunately, if the system is of any significant size, it will have many bugs; one is hopeful, however, that they will not be too numerous or serious, and with careful testing one can have a program that is well-behaved most of the time.

The second strategy is to try to prove something about the actual code of the computer system. The challange is then to deal with all of the special cases, exceptions, assumptions, and other complexities involved in real code. The details can quickly become overwhelming if one is writing traditional pen-and-paper proofs. There is another possibility, however, which is to use a computer to help with the process.

Although there are many automated theorem checkers in existence, including Coq, Isabelle/HOL, and Twelf, using a computer to develop and check proofs is not easy. Generally speaking, a computer is better at proof checking than proof development (e.g., type checking is simpler than type inference). Some formal methods, such as the simpler type systems or model checking, have very significant automation possibilities, where the computer can do most or even all of the work. Other

methods, such as verification, tend to only use automation to check proofs; fully automatic generation of verification proofs tends to be too difficult.

The advantage of using a computer to check that a property has been demonstrated is that computers are very good at handling the bulk of detail associated with real code, so that one can apply formal techniques to the actual program and still be confident about the result. The disadvantage of automated techniques is that the parts that require human input can be very laborious. Indeed, the endeavor becomes similar to building a large software project, and quite unsuprisingly "proof engineering" becomes very important. For example, good machine-checked proofs, like good software, should be modular so that parts can be re-used in other systems.

1.2.2 Source vs. Target

One major problem for formal methods is that the analysis tends to be done on source code, whereas the code that actually runs on the chip is machine code. In between sits the compiler, a very large software engineering artifiact. Any bug in the compiler could cause the machine code to behave in a different way from the source code, invalidating any property guaranteed. Since compilers can easily be hundreds of thousands of lines long, most have many bugs, even when they have been extensively tested.

Until a few years ago, this problem was considered so difficult that

no-one knew how to handle it satisfactorially. One technique that works in limited cases is to analyze the machine code directly. The problem has been that most real systems are not developed in assembly language, making those techniques useful only in rare circumstances. However, in recent years people have been improving compiler technology, for example with Proof-Carrying Code (PCC), developed in [Nec97], in the hope of addressing this issue.

1.2.3 Certified Compilers, CompCert, and Sequential Separation Logic

Most proof-carrying code systems do not guarantee the correct behavior of the compiled code, but instead guarantee that certain incorrect behaviors can never occur. That is, they produced a safety guarantee, not a correctness guarantee. However, in order to soundly apply more general kinds of reasoning at the source level to code at the machine level, a correctness guarantee—that is, a guarantee that source code was correctly translated into machine code by the compiler—was required. Compilers with this guarantee are called *certified*. With certified compilers it becomes feasible to connect correctness proofs at the source language to guarantees about how the hardware will actually behave when running the compiled code.

One of the first certified optimizing compilers from a realistic source language to a realistic assembly language was the CompCert compiler, developed by Leroy in [Ler06], which translates a language called C

minor to the Power PC. CompCert has a correctness proof, machine-checked in Coq, that proves that the compiler translates programs correctly.

Leroy found that using operational semantics was the most convienent for proving his compiler correct. However, when verifying a concrete program, it is frequently more convienent to use an axiomatic semantics, such as Hoare logic [Hoa69]. Separation logic is a variant of Hoare logic that includes elegant rules for reasoning about pointers [Rey02]. Since the focus of this thesis is on concurrency, we will refer to a separation logic for a sequential language as *seqential separation logic* (SSL).

Appel and Blazy developed a version of SSL for C minor, and constructed a machine-checked *soundness proof* for it in Coq [AB07]. A soundness proof for an axiomatic semantics demonstrates the connection between that axiomatic semantics and the underlying operational semantics. In other words, if one has a program verified with SSL to have certain properties, the soundness proof demonstates that the program actually has those properties. Depending on the complexities of the desired axiomatic semantics, and the underlying operational semantics, it can be quite difficult to establish soundness.

The result is an end-to-end system: one can take a source program, prove properties about it using the axiomatic semantics, and then have a guarantee about how the actual machine code will execute. As significant as systems such as CompCert have been, however, one major

shortcoming has been the lack of support for concurrency—all certified compilers in existence have focused on purely sequential settings [LPP05, Ler06, Moo89].

1.2.4 Formal Methods for Concurrency

As previously discussed, verification even in a purely sequential setting is difficult enough; adding in concurrent language features usually requires significantly more effort on the part of the verifier. Classic techniques have required verifiers to reason about the state of all the other threads at each program point, which can easily require an exponentially-sized proof.

The proof becomes unmanageably large beacause in a concurrent program different sections of the code interact (typically via shared memory and synchronization operations) despite the lack of explicit connectives such as function calls. This kind of global, whole-program behavior has made it very difficult to reason locally or to use simpler, sequential-style techniques. This is particularly frusterating, since real concurrent systems tend to have large portions of purely sequential code, where the interleaving should not have any important effect on the result. Those sequential portions are often difficult to reason about on their own, and it is painful to be forced to reason simultaneously about some complex sequential feature such as function call and some concurrent feature such as context switching.

In fact, the typical situation is that there is an existing sequen-

tial system, comprising both implementation (code) and verification (proof), and the desire is to add concurrency to that system. However, not only does one have to modify the code to take advantage of concurrency, but also one must modify the proofs, even for those parts of the code that are unchanged and that "should" not be interacting with the concurrency additions.

1.2.5 Concurrent Separation Logic

Recently, O'Hearn developed *Concurrent Separation Logic* (CSL), a Hoare-style logic designed to verify concurrent programs [O'H07]. CSL has two major properties that help solve many of the problems typically associated with the verification of concurrent programs. First, it modularly isolates the different threads of execution from each other, so that when when reasoning about a given thread, one does not have to consider the states of all the other threads, i.e., one can use thread-local reasoning. Second, the rules of CSL are a proper superset of the rules of SSL. Thus if one has laboriously verified a program with SSL and wishes to add concurrency, all of the parts of the code that have not changed will have exactly the same verification.

As groundbreaking as CSL was, as originally proposed it had several drawbacks. First, some of the language features supported were nonstandard, e.g., for example the treatment of shared local variables. Second, and more worryingly, O'Hearn did not have a soundness proof.

The lack of standard language features prevents programming with

standard alogrithms or techniques. Some of these ommissions, such as the lack of functions, were done simply to make an attempt at a soundness proof simpler (that is, functions were not considered to be part of the "core" of the problem[1]). Other features, such as allowing for the dynamic creation and destruction of locks, were missing because it was quite unclear how to prove them sound.

Even with significant simplifications, it was quite difficult to prove CSL sound. Due to the complex nature of concurrency and to a number of subtle issues involving resource invariants, the soundness of CSL was doubted for some time until Brookes developed a soundness proof for it [Bro07][2].

1.3 The Goal of Oracle Semantics

The goal of this thesis is to support the verification of concurrent programs in a way that is natural to the end-user and allows for substantial re-use of existing sequential code and associated machine-checked proofs; using this semantics we prove the soundness in Coq of a new concurrent separation logic with first-class locks and threads. The key technical advance is to isolate proofs about sequential language features and proofs about concurrent language features from each other. This

[1] In the case of first-order functions, it is true that they are noncore; first-class functions, however, are another story.

[2] "At that time I had no model and was scared about soundness. Then John Reynolds showed surprising subtleties regarding soundness, and we were facing an extremely difficult problem in the semantics of concurrency. We needed expert help, and it arrived in Steve Brookes who rode in and saved the day (and the whole approach)." [O'H]

substantially simplifies the reasoning for each part, since when reasoning about complex sequential features one does not have to worry about the issues typically associated with concurrency, and when reasoning about complex concurrent features one does not have to worry about all of the messy details involved in pure sequential control flow. This isolation also makes the resulting system more robust, since changes to one part of the system do not require changes in the other.

The key technical advances are

1. A powerful new concurrent language, Concurrent C minor, which is related to to a certified compiler.

2. A new Concurrent Separation Logic with first-class locks and functions.

3. A series of modules and other engineering developments to add concurrency to a certified compiler.

4. A new modal substructural logic, used for the assertions of CSL.

5. A novel concurrent operational semantics that uses the logic in its operational semantics to guarantee lock invariants are obeyed.

6. A new pseudosequential oracle semantics that gives a sequential view of concurrent computation.

7. A new Hoare tuple that allows for assertions to be directly embedded into syntax.

8. A soundness proof of CSL with respect to the concurrent operational semantics.

Note about proofs and Coq. This thesis is about two related but distinct developments. First, it is about a series of mathematical ideas developed to prove a soundness result. Second, it is about the engineering work required to make a 60,000 line soundness proof check in the Coq theorem prover. From time to time, the two developments have influenced each other, sometimes leading to new mathematics or better engineering. When the presentation here diverges from the Coq development there are two likely causes. First, in general our mathematical understanding is ahead of our Coq implementation; therefore, from time to time in the text we present a cleaner or more elegant definition than the one in the Coq proof. Second, we frequently simplify some of the significant complexities from the Coq definitions so that the kernel of the mathematical idea is clear. When we diverge from the Coq implementation we try to mention this fact in the text.

1.4 Structure of Thesis

First, since the verification of concurrent programs is a fairly extensive goal, a major part of the research is learning about existing systems that support parts of that goal, so that they can be connected together. Since the ideal end-goal is to have guarantees about the machine code that actually executes, we develop our system in Coq for the C minor

language, to enable a connection to Leroy's CompCert certified compiler. This and other related work is introduced in chapter 2.

Second, since C minor is a sequential language, we add concurrent statements to create the new language Concurrent C minor. We add the standard concurrency primitives `lock` and `unlock`, and we allow the kind of dynamic lock creation and destruction that programmers are used to. We also add a `fork` statement to start a new thread of execution. Together, these statements allow for a familiar, well-understood concurrency setting in the style of Pthreads. The full details of Concurrent C minor language are discussed in chapter 3.

Third, we extend and modify O'Hearn's CSL to reason about our new language. First-class locks and functions (that is, locks that can guard function pointers and function pointers that can take locks as arguments) turn out to be a significant technical advance, and were one of the substantial challanges in building the model. Our new CSL is similar to the one independently developed by Gotsman et al, indicating that it is the natural way to extend CSL to handle Pthreads-style concurrency. Our version of CSL is developed in chapter 4.

Fourth, we define a simple concurrent operational semantics for Concurrent C minor called the erased concurrent operational semantics to demonstrate that we have a realistic operational semantics for concurrency. The erased concurrent operational semantics is presented in chapter 5.

Fifth, we explain the modularization developed to isolate the se-

quential and concurrent features of the language from each other. The modularization also isolates the concurrency system from being dependent on the C minor language, which will be helpful when extending the CompCert compiler. The modularization is explained in chapter 6.

Sixth, to develop the semantic model for invariants, we develop a modal logic. This logic allows us to reason elegantly about different requirements in an orthogonal, modular fashion, allowing us to build up our different invariants cleanly. Our modal logic is quite powerful, allowing for contravariant recursion, impredicative quantification over both values and predicates, function pointers, and other features required for handling programs with abstraction. After we model our invariants using our modal logic, we must next develop a model for our modal logic, which we do in the style of Appel, Mellies, Richards, and Vouillon [AMRV07]. Our modal logic and the associated semantic model is presented in chapter 7.

Seventh, we define a more complex concurrent operational semantics called the unerased concurrent operational semantics. The new semantics does substantial additional bookkeeping that guarantees that it gets stuck on ill-synchronized programs; if we can show that a program does not get stuck, then we know that it is well-synchronized. We prove an erasure theorem that guarantees that if the unerased machine is not stuck then the erased machine is not stuck. The unerased concurrent operational semantics of Concurrent C minor is presented in chapter 8.

Eigthth, to gain confidence that our new CSL is well-defined, we

must develop a soundness proof for it, connecting it to our concurrent operational semantics. However, connecting it directly is messy. Instead, we define a new *oracular* semantics, which looks and behaves sequentially, but is able to proceed at concurrent instructions. This semantics will be so sequential that we will be able to use definitions and proofs developed in purely sequential settings with it, and properties proven in the oracular setting will then carry over into the concurrent one. The oracle semantics and its soundness proof is developed in chapter 9.

Nineth, we connect our new CSL to our oracle semantics. To do this we must define the Hoare triple in terms of the oracular operational semantics, and then prove the CSL axioms as lemmas from that definition. We are greatly aided by being able to connect to the oracle semantics, which lets us largely re-use a definition for the Hoare triple first developed in the sequential setting by Appel and Blazy. Using this imported definition we are able to recycle their proofs of all of the rules of CSL that come from SSL (which constitute a majority of the rules of CSL). For the rules about the concurrent features, such as the `lock` rule, we develop the proof from scratch but are able to ignore the sequential features while arguing about concurrent behavior. Our definition of the Hoare triple and the proofs of the CSL rules is presented in chapter 10.

Tenth, we present areas for future research. These include modifying the CompCert compiler to handle concurrency, developing techniques to reason about lock-free algorithms of various kinds, and reason-

ing in the presence of weak memory models. We conclude by observing that this system would allow for end-to-end reasoning of programs, from the source code to the machine code, all in a machine-checked manner. Our view on future work and concluding thoughts are contained in chapter 11.

Chapter 2

Related Work

2.1 Hoare logic and separation logic

Hoare logic is a verification technique developed by Hoare and Floyd to verify the behavior of a program [Flo67, Hoa69]. A *Hoare triple* is a judgement of the form $\{P\}\ c\ \{Q\}$, where P and Q are *assertions* and c is a *command*. Assertions are formulas in some logic, and the command c is a statement in a programming language. P is usually called the *precondition*, and Q the *postcondition*. The informal meaning of a Hoare triple is "If P holds, then after one executes c, Q will hold."

A Hoare logic is a set of triples, called *axioms* or *rules*. Typically these contain free variables, which are universally quantified. For example:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}$$

This rule, the *skip axiom*, indicates that for any precondition P, if

one runs the command `skip`, then P will hold afterwards[1]. Another common rule is the *sequence axiom*:

$$\frac{\{P\}\ c_1\ \{R\} \qquad \{R\}\ c_2\ \{Q\}}{\{P\}\ c_1 ; c_2\ \{Q\}}$$

The sequence rule says that it is valid to chain together sequences of assertions, allowing for pre- and postconditions for groups of statements, functions, and whole programs. It certainly seems like a reasonable rule—however, it is usually not true in concurrent setings! Another thread could modify the state after c_1 has executed, invalidating the postcondition R, thus invalidating any guarantee about the state after c_2 executes. This is an indication of why moving from sequential to concurrent settings is difficult.

2.1.1 Soundness for a Hoare Logic

If a Hoare logic is simply a set of axioms, how does one determine which sets of axioms are reasonable? In other words, consider the following two possible axioms:

$$\text{axiom 1}\ \frac{}{\{\texttt{x} = 0\}\ \texttt{x++}\ \{\texttt{x} = 1\}}$$

$$\text{axiom 2}\ \frac{}{\{\texttt{x} = 0\}\ \texttt{x++}\ \{\texttt{x} = 2\}}$$

[1]The command `skip` does nothing.

Which one is correct? On first inspection, the first one seems much more likely to be correct than the second. However, the true answer depends on what it means for an assertion such as $\{\texttt{x} = 0\}$ to hold, what pre- and postconditions are, and the result of the command `x++`.

For example, many garbage-collected languages use a "tag bit": that is, if the lowest bit in an integer value is 0, then that value is regular data; if on the other hand the lowest bit is 1, then that value is a pointer. In that case, all integer calculations would be carried out using only the upper 31 bits. If the logic was designed to reason about this issue (which would be important if one was trying to verify the garbage collector in question), then in fact axiom 2 would be the correct rule. Perhaps for simple languages, one can simply "eyeball" the rules, but for languages with complex features more is required.

A *soundness proof* is needed to demonstate that a particular set of axioms is valid. A soundness proof of an axiomatic semantics like Hoare logic is a connection between the definitions of "assertion," "precondition," "postcondition," and the operational semantics of the language in question. To make this connection, a formal definition is given to the Hoare triple, and the Hoare axioms are proven as lemmas from that definition.

Despite the importance of establishing soundness, hereafter, for ease of presentation, statements have the "obvious" semantics unless otherwise specified.

2.1.2 Separation Logic

One problem with Hoare logic is that it is difficult to use it to reason about pointers, due to aliasing. Consider the judgment:

$$\frac{}{\{\texttt{x} \mapsto 0 \ \wedge \ \texttt{y} \mapsto 0\} \ \texttt{[x] := 1} \ \{???\}}$$

Here, the notation $a \mapsto b$ means that a is a pointer and is pointing to a memory location that contains the value b. The question is, what is the correct postcondition? One obvious candidate is $\texttt{x} \mapsto 1 \ \wedge \ \texttt{y} \mapsto 0$, but it is not the only candidate: there is also $\texttt{x} \mapsto 1 \ \wedge \ \texttt{y} \mapsto 1$, since x and y could be *aliased*—that is, they could point to the same location in memory. In that case, updating `[x]` would also update `[y]`. This ambiguity causes the logic to be unsound (that is, a property "proven" using the logic might not actually hold when the program is executed). This is exactly the kind of subtle problem a soundness proof prevents.

There are a variety of possible solutions to the problem of aliased pointers. For example, one could include the explicit requirement that `x` and `y` not be aliased:

$$\frac{\texttt{x} <> \texttt{y}}{\{\texttt{x} \mapsto 0 \ \wedge \ \texttt{y} \mapsto 0\} \ \texttt{[x] := 1} \ \{\texttt{x} \mapsto 1 \ \wedge \ \texttt{y} \mapsto 0\}}$$

The benefit of this approach is that with the restrictions on aliasing in place, the logic becomes sound once again. However, the number of aliasing constraints grows quadratically with the number of locations. In

some contexts this may not be a problem, but often the detail becomes overwhelming.

Separation logic, developed by Reynolds and O'Hearn, gives a more abstract, cleaner view of the problem[2] [Rey02, IO01]. The key idea is to add a new operator, the separating conjunction $*$, to the assertion logic. $P * Q$ means that P and Q hold on *disjoint* sections of memory. This means that updates to the part of memory described by P cannot affect the validity of Q, and vice versa. Using this new operator, we can phrase our initial axiom in a different way:

$$\frac{}{\{\texttt{x} \mapsto 0 \ * \ \texttt{y} \mapsto 0\} \ \texttt{[x] := 1} \ \{\texttt{x} \mapsto 1 \ * \ \texttt{y} \mapsto 0\}}$$

The advantage of presenting things this way is that the aliasing constraints are bundled up in the $*$ operator, and thus do not overwhelm the reasoning.

The key to understanding the power of the separating conjunction is the *frame rule*:

$$\frac{\{P\} \ c \ \{Q\}}{\{P * F\} \ c \ \{Q * F\}}$$

The frame rule says that if one has proven the validity of some statement "in a vacuum," then it is valid to add it to a larger program, even if the other parts of the program place restrictions on parts of memory untouched by the statement in question. It thus supports local reason-

[2] As mentioned in chapter 1, since this work focuses on concurrency, separation logic that does not handle concurrency will be called *sequential separation logic* (SSL).

ing by allowing verifiers to concentrate on the parts of the state being modified by the statements under consideration. One critical point is that the frame rule is not true if one substitutes the normal conjunction for the separating one:

$$\frac{\{P\}\ c\ \{Q\}}{\{P \wedge F\}\ c\ \{Q \wedge F\}}$$

This rule is unsound because F could be describing some of the same memory described by P and modified by c.

2.2 Concurrent Separation Logic

Another problem with Hoare logic is that it is difficult to use it to reason about concurrency. Consider again the sequence rule:

$$\frac{\{P\}\ c_1\ \{R\} \quad \{R\}\ c_2\ \{Q\}}{\{P\}\ c_1 ; c_2\ \{Q\}}$$

As mentioned above, this rule is usually false in a concurrent setting because the action of other threads executing between c_1 and c_2 can invalidate the postcondition R of c_1 before c_2 can execute. If R does not hold, then there is no gurantee that c_2's will result in the postcondition Q, making this rule unsound. A soundness proof would prevent subtle problems resulting from such a rule.

As with aliasing, there are several possible approaches to solve this issue. One possibility, somewhat analogous to the solution to the alias-

ing problem, is to add complicated premises that argue about the state of all of the other threads. The problem is that if done naïvely the proof grows rapidly with the size of the program.

O'Hearn developed an alternative approach, *concurrent separation logic* (CSL) [O'H07]. The key idea is embedded in the *parallel rule*:

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \qquad \{P_2\}\ c_2\ \{Q_2\}}{\{P_1 * P_2\}\ c_1\ \parallel\ c_2\ \{Q_1 * Q_2\}}$$

Here, $c_1 \parallel c_2$ indicates that the statements c_1 and c_2 are to be executed concurrently. If each thread is operating on its own part of memory (which is guaranteed by the separating conjunction), then it is valid to reason about them independently.

CSL includes all of the rules of SSL for verifying the individual threads—that is, the rules of SSL are a proper subset of the rules of CSL. This means that CSL includes the sequence rule, which typically is unsound in concurrent settings. Informally, the sequence rule is sound because all the threads are operating on disjoint parts of memory. Thus, when executing the sequence $c_1;c_2$, any postcondition R of c_1 cannot be invalidated by the action of any other thread, meaning that it will be valid to use R as the precondition of c_2.

Verifying a program with CSL not only allows local reasoning, thereby avoiding significant space blow-up, but also allows for the direct re-use of SSL verifications, since all of the rules of SSL are in CSL. This second benefit is a significant advantage, since verification can be extremely

labor-intensive.

2.2.1 Thread Communication

The parallel rule allows two processes to run independently, but says nothing about how processes collaborate. To achieve collaboration, which is an important part of concurrent programming, additional rules and semantics are needed.

The synchronization device used in CSL is the *lock*, also called a *resource*. In O'Hearn's CSL, resources are declared statically at the beginning of a program, and in the verification each resource r is associated with a *resource invariant* RI_r.

To use the resource r, O'Hearn provides the *critical section rule*:

$$\frac{\{(P * RI_r) \ \wedge \ B\}\ c\ \{Q * RI_r\} \qquad \text{No other process modifies variables free in P or Q}}{\{P\}\ \texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ c\ \texttt{endwith}\ \{Q\}}$$

When inside the critical section, the code may use not only the thread's private memory, but also the memory controlled by the resource. The critical region c can proceed only when it has exclusive access to the resource r and the boolean value B is true. The side condition involving other processes may seem very strong; it is necessary because in O'Hearn's CSL, local variables are shared between processes. The separating conjunction ensures that memory is divided appropriately, but does not say anything about the local variables.

2.2.2 Limitations to the Approach

Despite the significant advantages of the approach, as originally presented, CSL had several drawbacks. First, several of the features of the language for which CSL was developed were not standard. The parallel composition operator ‖ and the critical section construct are more static and less general than dynamic commands such as `fork` to start parallel computation in an open-ended manner, and `lock` and `unlock` for synchronization purposes. In other words, it is straightforward to use `fork`, `lock`, and `unlock` to implement ‖ and `with-when-do`; however it is not obvious how to do the opposite.

Second, the idea of sharing local variables between threads is problematic. Local variables are typically compiled to registers, and registers are not shared between threads by the context-switching mechanism. This makes shared local variables in a language questionable. In addition, one unfortunate result of sharing local variables is the second premise of the critical section rule, which places requirements on other processes and therefore has more of a global flavor than might be hoped.

Third, CSL lacked first-class locks. As mentioned above, in O'Hearn's CSL, all locks are statically declared at the beginning of the program, and cannot be created once the program has begun. Programming in this style is frequently unnatural, and in many real programming languges, particularlly in object-oriented languages, locks are created and destroyed constantly as programs execute. Dynamic first-class locks were not included in CSL because it was unclear how to prove them

sound; indeed they were regarded as a significant technical challenge, as indicated by Bornat et al. [BCOP05]:

> ...the idea of semaphores in the heap makes theoreticians wince. The semaphore has to be available to a shared resource bundle: that means a bundle will contain a bundle which contains resource, a notion which makes everybody's eyes water. None of it seems impossible, but it's a significant problem, and solving it will be a small triumph.

Fourth, CSL did not have function pointers. Actually, the original CSL did not even have functions! First order functions (*i.e.*, functions that could not be passed as arguments or protected by locks) would not cause any *fundamental* problems, but it was thought [O'H07, §4] that they would complicate the soundness proof, a good example of why a machine-checked proof is so important when reasoning about more realistic systems. However, first class functions and function pointers are a considerably harder problem for the same kinds or reasons that first-class locks are a harder problem than static locks.

2.2.3 Brookes's Soundness Result

When CSL was developed there were serious doubts about the soundness of the whole approach [O'H]. Reynolds concocted a clever example in which a typical rule of separation logic, the *conjunction rule*, did not hold [O'H07]. An additional technical requirement on resource invari-

ants called *precision* seemed to help, but it was unclear if other troubles were lurking.

Fortunately, Brookes was able to develop a soundness proof for CSL [Bro07], in which he uses a local interpretation of program traces to model the semantics of CSL. Brookes was aided by some of the simplifications of CSL, particularlly including the lack of first-class locks and functions, but it was still a very noteworthy development. Although we depart from Brookes's model in many respects, there are several ideas in common, including the idea of modeling a race condition as a fatal error in the program execution.

2.2.4 Summary of Concurrent Separation Logic

As mentioned, there were several limitations and drawbacks to the approach as originally stated. However, the basic idea—that threads usually execute on private data, and communicate via special synchronization primitives such as locks, which control access to shared resources—is both natural and powerful. Moreover, the ability to re-use verification results from sequential separation logic is a powerful advantage.

2.3 Rely/Guarantee

The idea of verifying concurrent programs by relying on noninterference was introduced by Owicki and Gries [OG76]. However, their method was not compositional. Jones introduced the rely/guarantee method

[Jon83], which has a more modular design. Each thread is given a pair of assertions, one of which is the *rely* for that thread and the other of which is the *guarantee*. The thread is allowed to assume that the rely always holds, and must never break the invariants of the guarantee. Other threads will then be able to rely on the current thread's guarantee, and will be responsible themselves for guaranteeing the current thread's rely.

Rely/guarantee allows for relatively modular reasoning at the thread level, but one major drawback is that the flavor of the rely/guarantee assertions is quite global. However, it is quite good at reasoning about interference between threads. In contrast, concurrent separation logic allows for much more local reasoning, but to reason about thread interference can require large numbers of auxiliary variables and unnatural proofs.

It is natural to hope to combine the two approaches, and Vafeiadis and Parkinson [VP07] demonstrate how to do so. Their combination allows a verification to contain the local reasoning typical of concurrent separation logic for most of the program, as well as the more global rely/guarantee reasoning for those parts of the program that require it, and thereby enjoy the benefits of both.

a	:=	id	local variable
	|	$id = a$	variable assignment
	|	$op(\vec{a})$	constants and arithmetic
	|	`load`$(chunk, a)$	memory load
	|	`store`$(chunk, a_1, a_2)$	memory store
	|	`call`$(sig, a, \vec{a})$	function call
	|	a_1 `&&` a_2	sequential boolean "and"
	|	a_1 `||` a_2	sequential boolean "or"
	|	a_1 `?` a_2 `:` a_3	conditional expression
	|	`let` a_1 `in` a_2	local binding
	|	n	reference to let-bound variable

Figure 2.1: Leroy's C minor expressions

2.4 C minor and CompCert

The CompCert system was one of the first optimizing compilers to be proved correct with a machine-checked proof [Ler06]. CompCert consists of a definition of the C minor language, a certifying compiler, and that compiler's correctness proofs.

2.4.1 The C minor language

C minor is a simple imperative language based on C. It is intended to be the highest intermediate-level language in a compiler, but is elegant enough that one could write programs directly in C minor (e.g., a garbage collector). C minor contains expressions, statements, functions, and programs, composed in the standard way. The full grammar for the original C minor expressions is given in figure 2.1. Like expressions in C, expressions in C minor can have side effects and can contain function calls. C minor supports the full memory model of ANSI C,

s	:=	a;	expression evaluation
	|	$s_1; s_2$	sequence
	|	`if` a $\{\vec{s}_1\}$ `else` $\{\vec{s}_2\}$	conditional
	|	`loop` $\{\vec{s}\}$	infinite loop
	|	`block` $\{\vec{s}\}$	delimited block
	|	`exit` n;	block exit
	|	`return`; | `return` a;	function return

Figure 2.2: Leroy's C minor statements

including such features as pointer arithmetic and loading and storing variously-sized data from memory (the *chunk* parameter specifies data size). Unlike C, C minor requires explicit conversions and does not overload arithmatical operators. C minor expressions also include some features more commonly found in functional languages, such as local variable binding.

The full grammar for the original C minor statements is given in figure 2.2. Since expressions have side effects, most computation is done with a sequence of expressions. Control flow is given by a combination `loop`, `if-else` constructs, and `block` constructs. The `exit` n statement prematurely terminates the n enclosing `block` constructs. There is no general `goto` statement.

As originally presented, C minor was given a big-step structured operational semantics. The semantics was completely deterministic. Expressions evaluate to values v, which can be 32-bit integers, 64-bit floats, pointers, or a special `undef` value, used for unitialized values. C minor has no statements for input or output, so all computation is done starting from the initial contents of memory, and the result of the program

is the final contents of memory, plus a value returned from the special function `main`. One negative consequence of the choice of big-step semantics is the inability to reason about nonterminating computations and the difficulty in extending the semantics to handle concurrency.

2.4.2 The CompCert system

The CompCert compiler translates C minor into PowerPC assembly code. The translation is via several intermediate languages, with numerous optimizations and transformations, e.g. constant propagation, common subexpression elimination, register allocation, etc. Generally speaking, performance of compiled code is adequate, and compilation time is reasonable.

What was novel about the CompCert compiler, however, was that the system includes a proof of correctness, stating that the compiled code was a faithful translation of the source code. In other words, the compiler has no bugs that compromise correctness of the generated code[3].

The compiler itself is only about 13% of the total code. The remaining code is the proof of correctness. This includes (1) the specification of compiler correctness, including the formal syntax and semantics of C minor and the PowerPC (8%), (2) statements of intermediate theorems, supporting lemmas, and associated definitions (22%), (3) proof scripts for generating the proofs (50%), and (4) custom tactics for the proof

[3]Unlike for correctness, there is no formal guarantee of the *performance* of the generated code. However, in tests CompCert code was competitive with `gcc -O1`.

scripts (7%). Note that the Trusted Computing Base of the system (1) is thus slightly over half of the size of the compiler itself. However, as the compiler becomes more complicated, for example by introducing new optimizations, the specification of the system should not grow significantly.

Still, there is an interesting observation to be made: the specification and proofs of correctness of the compiler were approximately eight times larger than the implementation of the compiler. Beyond illustrating the the importance of good proof engineering, it is clear that when one wishes to modify the CompCert compiler, it is very important to consider the effects the change will have on the associated proofs.

This reality means that some changes can be much easier to make than others, even if both require roughly the same amount of modification to the compiler. This is particuarlly noticable for those kinds of changes that break invariants assumed by the correctness proofs.

One shortcoming of the CompCert system is that it is not obvious how to extend it to handle concurrency. In fact, the extension to concurrency is exactly the kind of extension that is so difficult, since the concurrency breaks many of the assumptions used by the proof. Ironically, changing the compiler itself to handle concurrency is probably fairly simple[4] – almost all of the difficulty is expected to be in modifying the proof of correctness.

[4]Changing the compiler is relatively simple because although CompCert is an optimizing compiler it does not do the kinds of aggressive optimizations that would be unsound in a concurrent setting, and can simply compile purely concurrent instructions like `lock` as if they were function calls to an external library.

2.5 Separation Logic for C minor

Another issue with the CompCert system is that while the system does guarantee that the target code has the same behavior as the source code, it does not provide an easy way to determine exactly what the behavior of the source code is. As mentioned above, the CompCert compiler is specified and proven correct with respect to the operational semantics of C minor and PowerPC. However, when verifying concrete source programs, it is usually simpler to use some kind of axiomatic semantics, such as sequential separation logic.

This was the major problem that Appel and Blazy [AB07] solved when they defined a separation logic for C minor, and proved it sound in Coq. As a key part of that work, they made many modifications to the C minor language and its semantics to better support the separation logic and future developments involving the C minor universe. The key changes to C minor developed by Appel and Blazy, in consultation with Leroy, are:

1. Modifying C minor to be more machine-independent
2. Modifying C minor to be a better target language for ML and Object Oriented languages
3. Changing the operational semantics from big-step to small-step
4. Modifying the semantics to better match a clean axiomatic semantics (sequential separation logic)

The first two changes substantially enhanse the ability of C minor and CompCert to be used more broadly, connecting more source languages to more target machines. This is quite important for the system as a whole, but it is the second two changes that are of primary concern to people who wish to add concurrency to the CompCert system.

2.5.1 A small-step relation for C minor

The third change, moving from a big-step semantics to a small-step one, is obviously useful for reasoning about concurrency. With only a big-step semantics, it is impossible to reason about partial computations, and the key point about concurrent computation is that communication between interacting threads always happens at midway points in the computation.

The small-step C minor step relation developed by Appel and Blazy has the form:

$$(\sigma, \kappa) \longmapsto (\sigma', \kappa')$$

where, σ is a *state* and κ is a *control*. A state σ in Appel and Blazy's C minor is a tuple $\sigma = (\Psi, sp, \rho, \phi, m)$, where:

- Ψ is a pair of program (mapping of addresses to function bodies) and global variable bindings (mapping of identifiers to addresses)
- sp is the stack pointer
- ρ is the local environment (mapping from local names to values)

e	::=	$\mathsf{val}\ (v)$	values, including constants
	|	$\mathsf{var}\ (id)$	local variable
	|	$\mathsf{op}\ (op, \vec{e})$	arithmetic; op is the opcode
	|	$\mathsf{load}\ (ch, e)$	memory load; ch is the data size
	|	$\mathsf{condition}\, e\, e_t\, e_f$	conditional expression (`?:` operator)
	|	$\mathsf{let}\, e_1\, e_2$	local binding
	|	$\mathsf{letvar}\, n$	reference to let-bound variable
s	::=	$id := e$	local variable assignment
	|	$[e_1]_{ch} := e_2$	memory store; ch is the data size
	|	$\mathsf{loop}\, s$	infinite loop
	|	$\mathsf{block}\, s$	declared block
	|	$\mathsf{exit}\, n$	break out of nth enclosing block
	|	$\vec{id} := \mathsf{call}\, \Sigma\, e\, \vec{e}$	function call[6]
	|	$\mathsf{return}\ el$	function return
	|	$s_1; s_2$	sequence
	|	$\mathsf{if}\, e\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2$	conditional
	|	skip	do nothing

Figure 2.3: Appel and Blazy's C minor

- ϕ is a new component added by Appel and Blazy, the *footprint*, which is a mapping from addresses to permissions, explained below

- m is the memory (mapping from addresses to values)

A control κ is a stack of the current instructions to be executed, block exit points, and function return points. Using their small-step relation, Appel and Blazy define $\mathsf{safe}(\sigma, \kappa)$ in the usual way – that the continuation (σ, κ) will never get stuck[5].

[5] As is common in formal methods, when a program enters an error state or has undefined behavior we say that the program "gets stuck". The major goal of a safety proof is to guarantee that programs do not get stuck.

[6] Σ indicates which parameters are integers and which are floating-point, and helps the compiler allocate registers for function calls properly.

The fourth change, modifying the semantics to better match a clean separation logic, involved many changes; a grammer of the new language is given in figure 2.3. Some, such as changing expressions to be side-effect free, are relatively simple, but simplify reasoning by making side-effects more explicit. A much more complex change was the addition to the semantics of a new component, the footprint.

2.5.2 Footprints, introduced

As presented by Appel and Blazy, a footprint ϕ is a map from memory addresses to permissions. Memory access outside the footprint causes the semantics to get stuck. The simplest model for permissions is a boolean value, with true representing an address which is safe to access and false representing an address which is not. Footprints also come with a *join* operation, which states which permissions are compatible. The join operation on footprints is a kind of separation algebra, as explained in section 4.4. The existence of footprints in the semantics became quite important when developing the semantics of Concurrent C minor.

2.5.3 Separation logic for C minor

In addition to the changes to the C minor language and semantics, Appel and Blazy designed a separation logic that could handle the control flow and other complex features of C minor. Assertions in separation

$$
\begin{array}{rcl}
\mathbf{emp} & \stackrel{\text{sem}}{=} & \lambda\sigma.\ \phi_\sigma = \emptyset \\
P * Q & \stackrel{\text{sem}}{=} & \lambda\sigma.\ \exists\phi_1.\exists\phi_2. \\
 & & \qquad \phi_\sigma = \phi_1 \oplus \phi_2 \ \wedge\ P(\sigma[:=\phi_1]) \ \wedge\ Q(\sigma[:=\phi_2]) \\
P \vee Q & \stackrel{\text{sem}}{=} & \lambda\sigma.\ P\sigma \vee Q\sigma \\
P \wedge Q & \stackrel{\text{sem}}{=} & \lambda\sigma.\ P\sigma \wedge Q\sigma \\
P \Rightarrow Q & \stackrel{\text{sem}}{=} & \lambda\sigma.\ P\sigma \Rightarrow Q\sigma \\
\neg P & \stackrel{\text{sem}}{=} & \lambda\sigma.\ \neg(P\sigma) \\
\exists z.P & \stackrel{\text{sem}}{=} & \lambda\sigma.\ \exists z.\, P\sigma \\
[A]_{\mathsf{Coq}} & \stackrel{\text{sem}}{=} & \lambda\sigma.\ A \qquad\qquad \text{where } \sigma \text{ does not appear free in } A
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{true} & \stackrel{\text{syn}}{=} & [\mathbf{True}]_{\mathsf{Coq}} \\
\mathbf{false} & \stackrel{\text{syn}}{=} & [\mathbf{False}]_{\mathsf{Coq}} \\
e \Downarrow v & \stackrel{\text{syn}}{=} & \mathbf{emp} \wedge [\mathsf{pure}(e)]_{\mathsf{Coq}} \wedge (\lambda\sigma.\ \sigma \vdash e \Downarrow v) \\
\lceil e \rceil_{\mathsf{expr}} & \stackrel{\text{syn}}{=} & \exists v.\, e \Downarrow v \ \wedge\ [\mathsf{is\_true}\, v]_{\mathsf{Coq}} \\
\mathsf{defined}(e) & \stackrel{\text{syn}}{=} & \lceil e \stackrel{\text{int}}{==} e \rceil_{\mathsf{expr}} \vee \lceil e \stackrel{\text{float}}{==} e \rceil_{\mathsf{expr}} \\
e_1 \stackrel{ch}{\mapsto} e_2 & \stackrel{\text{syn}}{=} & \exists v_1.\ \exists v_2. \\
 & & \qquad (e_1 \Downarrow v_1) \ \wedge\ (e_2 \Downarrow v_2) \ \wedge\ \mathsf{defined}(v_2) \wedge \\
 & & \qquad (\lambda\sigma.\ m_\sigma \vdash v_1 \stackrel{ch}{\mapsto} v_2 \ \wedge\ v_1 \in^{ch}_{\mathsf{store}} \phi_\sigma)
\end{array}
$$

Figure 2.4: Models of the Sequential Separation Logic Operators

logic are modelled as predicates on state, written in Coq as:

```
Definition assert : Type := state -> Prop.
```

`Prop`, here, is the Coq-level proposition[7]. One major advantage of defining assertions this way is that one can develop a *shallow embedding*, which means that when reasoning about the assertions in the Coq theorem prover, one is able to use standard Coq tactics, greatly saving engineering effort. Appel and Blazy define the standard operators of separation logic, as well as several operators more specific to C minor, as given in figure 2.4.

To handle function calls, global variables, function return, and breaking out of multi-level `block` statements, the Hoare triple is augmented with additional parameters Γ (for globals and functions), R (for the assertion that must be satisfied on function return, and which is a function from the results of the function to assert), and B, a list of the assertions that must be satisfied to exit out of various levels of blocks (a function from the naturals to assert). Therefore, the Hoare judgement for C minor has the following form:

$$\Gamma; R; B \vdash \{P\}\ c\ \{Q\}$$

[7]As elegant as this definition is, there is one unfortunate problem with it: since assertions are predicated on states, and states include Ψ, and Ψ includes a function whose range is code (syntax), it is impossible to embed these assertions directly in program syntax. For sequential C minor this was only modestly limiting (for example, it prevents defining an `assert` statement that takes a real semantic assertion), but when concurrency was added this caused some severe difficulties. In chapter 10 we will show how to solve this problem.

$$
\begin{array}{rcll}
P\,\square_A\,\kappa & = & \forall\sigma. & (A * P)\ \sigma \ \Rightarrow\ \mathsf{safe}(\sigma,\kappa) \\
R\,\mathbb{R}_A\,\kappa & = & \forall\vec{v}. & R(\vec{v})\ \square_A\ \texttt{return}\ \vec{v}\cdot\kappa \\
B\,\mathbb{B}_A\,\kappa & = & \forall n. & B(n)\ \square_A\ \texttt{exit}\ n\cdot\kappa \\
\mathsf{frame}\ (\Gamma, A, s) & = & & \Gamma * \mathsf{closemod}\ (s, A) \\
\Gamma; R; B \vdash \{P\}\ c\ \{Q\} & = & \forall A, \kappa. & (Q\,\square_{\mathsf{frame}\ (\Gamma,A,s)}\,\kappa)\ \wedge \\
 & & & (R\,\mathbb{R}_{\mathsf{frame}\ (\Gamma,A,s)}\,\kappa)\ \wedge \\
 & & & (B\,\mathbb{B}_{\mathsf{frame}\ (\Gamma,A,s)}\,\kappa)\ \Rightarrow \\
 & & & \quad (P\,\square_{\mathsf{frame}\ (\Gamma,A,s)}\,c\cdot\kappa)
\end{array}
$$

Figure 2.5: Continuation-passing style definition of Hoare tuple

The Appel and Blazy definition of the Hoare tuple is given in figure 2.5. An assertion P *guards* a control κ in frame A, written $P\,\square_A\,\kappa$, if whenever $A * P$ holds, it is safe to execute κ. Using the basic guard, they define derivative guards for function return and block exit. Then the Hoare tuple is defined using the various guards in a continuation-passing style[8]. There are many details here, which is common when building machine-checked proofs of large, realistic systems, but in simple terms, the Hoare tuple says that for any continuation κ, if the post-condition Q ensures the safety of κ, then the precondition P ensures the safety of c followed by κ.

Using this definition, Appel and Blazy prove, in Coq, soundness of the rules of sequential separation logic given in figure 2.6. The rules for `skip`, sequenceb, assignment, and `loop` are standard. The rule for storing the the heap requires that e and e_2 be pure—that is, not read from the heap. Appel and Blazy demonstrate how to accommodate impure expressions by applying local program transformations. The `if-then-else` rule also requires that e be pure, which can be handled in

[8] $\mathsf{closemod}\ (s, A)$ closes the assertion A over any variables modified by s.

$$\frac{}{\Gamma; R; B \vdash \{P\}\ \mathsf{skip}\ \{P\}}$$

$$\frac{\Gamma; R; B \vdash \{P\}\ s_1\ \{P'\} \qquad \Gamma; R; B \vdash \{P'\}\ s_2\ \{Q\}}{\Gamma; R; B \vdash \{P\}\ s_1; s_2\ \{Q\}}$$

$$\frac{\rho' = \rho_\sigma[x := v] \qquad P = (\exists v.\, e \Downarrow v\ \wedge\ \lambda\sigma.\, Q\,\sigma[:= \rho'])}{\Gamma; R; B \vdash \{P\}\ x\ :=\ e\ \{Q\}}$$

$$\frac{\mathsf{pure}\,(e) \qquad \mathsf{pure}\,(e_2) \qquad P = (e \overset{ch}{\mapsto} e_2\ \wedge\ \mathsf{defined}(e_1))}{\Gamma; R; B \vdash \{P\}\ [e]_{ch} := e_1\ \{e \overset{ch}{\mapsto} e_1\}}$$

$$\frac{\mathsf{pure}\,(e) \qquad \Gamma; R; B \vdash \{P \wedge e\}\ s_1\ \{Q\} \qquad \Gamma; R; B \vdash \{P \wedge \neg e\}\ s_2\ \{Q\}}{\Gamma; R; B \vdash \{P\}\ \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \{Q\}}$$

$$\frac{\Gamma; R; B \vdash \{I\}\ s\ \{I\}}{\Gamma; R; B \vdash \{I\}\ \mathsf{loop}\, s\ \{\mathbf{false}\}} \qquad \frac{\Gamma; R; Q \cdot B \vdash \{P\}\ s\ \{\mathbf{false}\}}{\Gamma; R; B \vdash \{P\}\ \mathsf{block}\, s\ \{Q\}}$$

$$\frac{}{\Gamma; R; B \vdash \{B(n)\}\ \mathsf{exit}\, n\ \{\mathbf{false}\}}$$

Figure 2.6: Axiomatic Semantics of Separation Logic (without call and return)

the same way. The `block` and `exit` rules enable reasoning for nonlocal control flow. Not pictured are the standard rules of separation logic such as the rule of consequence and the frame rule; also missing are the rules for call and return. For a full description of all of the rules and the consequences for sequential control flow we refer the reader to [AB07].

The proofs of these rules with respect to the definition of the Hoare tuple is done in Coq [AB07]. The most difficult rule to prove was the `loop` rule, which interacted complexly with the `block` and `exit` rules.

The full proof base for Appel and Blazy is 23,557 lines. 37% is

the memory model, and basic definitions, all of which are shared with CompCert. 19% is the definition of the C minor language and the model for permissions. 2% is the operators and rules of sequential separation logic, and the final 42% is the proofs of soundness for those rules.

Like the correctness proofs of the compiler, the proofs of the soundness of the separation logic rules make a number of assumptions about the underlying semantics. Unfortunately, just as in the compiler correctness proofs, several of the assumptions used, such as determinancy of the step relation, are typically false in a concurrent setting. The challenge when adding concurrency will be in a large part how to adapt these sequential-language proofs to a concurrent language.

In the remainder of this thesis, since the focus is not on sequential control flow, both the R and B parameters will be elided from the presentation. However, their existance forms a part of the *motivation* to design the concurrency system cleanly, which is why they were presented in full detail here: the Hoare tuple is complex enough to begin with, and there is every reason to avoid making it any more complicated than it already is.

2.5.4 End to end

With the combination of Appel and Blazy with the work of Leroy, it is possible to create an end-to-end guarantee: properties proven of the source code using separation logic will hold on the code that executes on the machine. Moreover, since the entire system is machine-checked end-

to-end, there is very high assurance that the *actual* program written, when run through the *actual* compiler used, will *actually behave as specified in the source code.*

2.6 Generation of proofs in CSL

One issue not yet discussed is where the proofs in CSL come from. One obvious possibility is that they are generated by humans, ideally in some machine-checked setting, as the code is being written. However, this can involve a great deal of additional effort if it is done directly, even considering that the amount of debugging time is dramatically less.

Appel [App06] presents one way to make the process simpler by giving a series of custom Coq tactics for reasoning about C minor programs in Coq. Using these tactics it is possible for a human to generate a proof more easily.

Another approach is given by Gotsman et al. [GBCS07]. Gotsman develops a separation-logic-based shape-analysis algorithm that can automatically generate correctness proofs for certain classes of simple concurrent programs.

Mansky [Man08] uses both approaches. First he takes a relatively complex example and proves it correct in Coq by hand, and then he implements the algorithm described in [GBCS07] in Coq, and uses it to examine simpler programs.

In general humans are able to verify very complex programs, but it takes an enormous amount of effort. In contrast, automatic techniques can only handle relatively simple cases, but are able to do so with great efficiency. Clearly a reasonable approach is to combine the two, letting humans handle the difficult parts of a program while using powerful inference techniques to automate the analysis of the simpler parts, but exactly how to do make everything fit together is an open problem.

2.7 Gotsman et al.'s CSL

Gotsman et al. independently developed a version of concurrent separation logic [GBC+07]. Their CSL is similar to ours, indicating that we have independently discovered *the* natural extension of CSL to first-order locks. Key differences between their work and ours include:

1. Our assertion language is more powerful, providing support for features such as impredicative quantification, higher-order recursion, and first-class function pointers. We are able to use this power to define very powerful assertions, such as the Hoare tuple, and check programs in a more modular way.

2. We can embed our assertions directly into the syntax of programs, while they must utilize a global table on the side.

3. We have a 60k line machine-checked soundness proof in Coq. The realities of engineering such a large system have forced us to de-

velop our system in a highly modular way: the sequential reasoning is isolated from the concurrent reasoning in the model, and the isolation is strong enough to make the proof in Coq tractable. This modularization leads to a very different flavor of soundness proof: while they argue via predicate transformers, we establish soundness in a more operational style.

4. We have a strategy for connecting to an existing certified compiler, and a thesis for why our results should hold on a machine with a weak memory model, thereby achieving an end-to-end system.

5. Gotsman's system is designed to connect to the shape analysis tool presented in [GBCS07].

6. Gotsman also presents a method for guaranteeing that all locks have been freed by the end of the program.

We suspect that it would not be difficult to extend our work to support (6). One interesting avenue for future work would be to combine the positive aspects of the two approches, thereby achieving both (4) and (5) and an even larger end-to-end system.

2.8 Miscellaneous related work

2.8.1 A very modal model

Our semantic model for assertions was built on top of ideas first developed for the Princeton Foundational Proof-Carrying Code project for modeling mutable references [AM01, AAV02, AAV03, Ahm04]. Appel et al. [AMRV07] re-engineered this model by developing a modal logic with an underlying Kripke model to achieve greater modularity.

2.8.2 The C0 compiler

Leinenbach et al. [LPP05] developed the C0 certified compiler. C0, like C minor, is a simplified version of the C language, and the C0 compiler translated C0 into the DLX machine language. This compiler was part of the larger Verisoft project, which built an entire machine-verified hardware/software stack, from the gate level through a TCP/IP stack. The verification was done in Isabelle/HOL. In comparison to the CompCert compiler, the C0 compiler is simpler, supporting very few optimizations.

2.8.3 Release Consistency

Release consistency, proposed by Gharachorloo et al. [GLL+90], is a memory consistency model first proposed in the systems/architecture community in 1990. This work was one of the first to demonstrate the

benefits of distinguishing lock/unlock memory accesses from ordinary loads and stores; we follow them in this regard.

Synchronization actions are guaranteed to be sequentially consistent, while normal accesses are not, with the exception that they must execute with respect to synchronization actions in program order. Assuming proper synchronization (similar in nature to that required by CSL), release consistency is consistent with sequential consistency; our thesis for why our system should be sound in the presence of weak memory models is similar.

Chapter 3

Concurrent C minor

Concurrent C minor is a new high-level intermediate language suitable for writing concurrent programs. Here we present the new language, give an informal semantics, and present an example. Later we give three different formal semanics for Concurrent C minor. First, in chapter 5, we give a simple concurrent operational semantics to demonstrate that we support a standard model for concurrent programming. Second, in chapter 8, we present a more complicated concurrent operational semantics that does additional bookkeeping to allow us to prove strong properties. Last, in chapter 9, we give an oracular semantics, which is a thread-local semantics for concurrent execution[1].

[1]Portions of this chapter have been published before as [HAZ08a] and [HAZ08b].

3.1 The Concurrent C minor language

The Concurrent C minor language is built on top of the (sequential) C minor language first developed by Leroy [Ler06] and then modified by Appel and Blazy [AB07]. The sequential features include: complex control flow, such as function calls and multi-exit loops; support for the full ANSI C memory model, including pointer arithmeric and the ability to work with variable-sized data[2]; and certain other features useful in an intermediate language such as locally nameless variables. In figure 2.3 we gave the grammar of Appel and Blazy's version of C minor, which we extend here.

We extend C minor with five statements to construct *Concurrent C minor*:

s	::=	...	
	|	$\mathsf{lock}\, e$	lock e
	|	$\mathsf{unlock}\, e$	unlock e
	|	$\mathsf{fork}\, e\, (\vec{e})$	start a child process
	|	$\mathsf{make\_lock}\, e\, R$	create the lock e with invariant R
	|	$\mathsf{free\_lock}\, e$	free the lock e

Unlike function calls, loops, and so forth, each of these statements is *straightline*, meaning that none of them result in nonlocal control flow in the thread that employs them; instead, after they complete, control

[2]We use "variable-sized data" to mean data that can be 8 bits long (byte), 32 long bits (word), etc.

always passes to the next statement in the program.

The $\mathsf{lock}\,(e)$ statement evaluates e to an address v, then waits until it acquires lock v. The $\mathsf{unlock}\,(e)$ statement evaluates e to an address v and then releases the lock v. A lock at location v is *locked* when the memory contains a 0 at v and *unlocked* when memory contains a 1 at v.

Each lock comes with a *resource invariant* R which is an assertion that explains how the lock should be used and which addresses in memory it owns. R must hold before unlocking a lock, which means the next thread to grab the lock will know that R holds. This is standard in CSL [O'H07], but we go further and use the invariants at a crucial point in our unerased concurrent operational semantics to guarantee the absence of race conditions.

The unerased concurrent operational semantics checks the truth of lock invariants when unlocking a lock; failure of this check causes the operational semantics to get stuck, as explained in in chapter 8. The language of these assertions contains the full power of logical propositions (Coq's `Prop`), so the operational semantics is nonconstructive, i.e., it is given by a classical relation[3].

The set of memory locations controlled by a lock need not be static; it can change over time depending on the state of memory (e.g., one can say, "this lock controls that variable-sized linked list"). As explained in section 2.5.2, if a thread tries to access memory it does not own, it gets

[3]We use a small, consistent set of classical axioms in Coq: extensionality, proposition extensionality, dependent unique choice, and relational choice.

stuck. This protocol ensures the absence of read/write or write/write race conditions.

The statement make_lock e R takes an address e and a lock invariant R, and declares e to be a lock with the associated invariant. make_lock will get stuck if memory location e does not contain a 0, meaning that the lock is created in the locked state. The address is turned back into an ordinary location by free_lock e, which also requires the location e to contain a 0. Both instructions are thread-local, i.e., they don't synchronize with other threads or the scheduler. It is illegal to apply lock or unlock to nonlock addresses, or to apply ordinary load or store to locks.

The fork statement spawns a new thread, which calls function e on arguments $\vec{e}$. No variables are shared between the caller and callee except through the function parameters. All functions in Concurrent C minor are given pre- and postconditions, and the parent passes to the child a portion of the memory it controls, specified by the precondition of the child. This portion typically contains visibility of some locks so that the two threads can communicate. Just as the operational semantics gets stuck at unlock if the lock invariant does not hold, it will get stuck at fork if the child's precondition does not hold. A thread exits by returning from its top-level function call.

We have not added a join operator since the Concurrent C minor programmer can implement join using a lock that is passed from parent to child and unlocked by the child just before exiting.

Although the syntax of Concurrent C minor requires the user to specify lock invariants and function pre- and postconditions, these can be taken directly from a program proof in concurrent separation logic. Therefore, if the author of a Concurrent C minor program wishes to develop a verified program, it is not an extra burden to augment the source code with the appropriate invariants.

3.2 Programming with locks

To illustrate programming in Concurrent C minor, we give an example program in figure 3.1. Here we explain the program's behavior and present an informal proof that it is bug-free. For cross-referencing purposes we number both the lines of the program in figure 3.1 and the informal proof.

i. Every program in Concurrent C minor has a special function `main`, where the program begins. In Concurrent C minor, at function declaration it is necessary to give pre- and postconditions. In the example program, `main` is declared in line 1, and both the pre- and postconditions are declared to be **emp**. The assertion language for pre- and postconditions is explained in section 4; the **emp** precondition indicates that `main` can be run at any time, and the **emp** postcondition indicates that when `main` terminates it has freed up all of its resources[4].

[4]This understanding of the postcondition is not quite accurate; a fuller explanation of this issue will be given in section 4.9.2.

```
1  void main() with pre:{emp} and post:{emp} {
2   L := call malloc_and_zero(4);
3   i := 0;
4   make_lock L R(L);
5   fork f(L);
6   *(L+3) := 1;
7   block {loop {
8            if (*(L+3)==0) exit 0 else skip;
9            *(L+1) := i; *(L+2) := i;
10           unlock L;
11           i := i+1;
12           lock L;
13  }}
14  free_lock L;
15  if (*(L+1)==*(L+2)) skip
16    else get_stuck;
17  call free(L,4);
18  return ();
19 }
20
21 void f(l) with pre:{l ⇝ R(l)} and post:{emp} {
22  loop {
23   lock l;
24   *(l+1) := *(l+1) * 2;
25   *(l+2) := *(l+2) * 2;
26   if (*(l+1) > 10) {
27     *(l+3) := 0;
28     unlock l;
29     return ();
30   } else skip;
31   unlock l;
32 }}
```

Figure 3.1: Sample concurrent program

l	*lock*
$l+1$	*data*
$l+2$	*data*
$l+3$	*continue?*

Figure 3.2: Informal description of $R(l)$

ii. In line 2, the program allocates and initializes a 4-word block of memory[5]. The local variable `i` is initialzied to 0 in line 3.

iii. In line 4, the bookkeeping instruction make_lock turns memory location L from a normal memory location into a lock with an invariant $R(\mathsf{L})$; see figure 3.2 for a schematic of the invariant. Informally, the invariant says that the lock L guards the next three locations in memory. The first two locations contain data values which are supposed to be equal. The last location contains a "continue" flag; a 1 indicates that the child thread is still computing, whereas a 0 indicates that it has completed. In section 4.9 we will formally discuss the exact invariant used. Like all locks, the lock is created in the locked state.

iv. In line 5, the function `f` is spawned as a child thread, and is passed the parameter `L`, the address of the still-locked lock.

v. In line 6, the continue field is set to 1, meaning that the child has not finished computing. Even though the child has already been started, this is safe because the child is not able to see the

[5]In the implemented Concurrent C minor, memory is byte-addressed; locks are word-sized and must be word-aligned. We simplify the presentation by assuming word addressing.

continue field until it grabs the lock L, which is currently being held by the parent.

vi. In lines 7–13, the main thread loops. First, in line 8, it checks to see whether the continue flag has been reset to 0; if so, it breaks out of the loop. Then, in line 9, it modifies the data cells by setting them equal to the local variable `i`, before unlocking the lock in line 10. Then it increments `i` in line 11, grabs the lock again in line 12, and loops back around.

vii. Control reaches line 14 after the child process has set the continue flag to 0, indicating that it has completed. Accordingly, it should be safe to convert the location L from a lock back into a regular piece of memory, so we use the bookkeeping instruction free_lock .

viii. In lines 15–16, there is a test that checks whether the two data cells contain the same value; if they do not, then the program gets stuck. Since both threads write to both data cells, it is not obvious whether this test will succeed. However, lines 9 and 24–25, where the data fields are mutated, shows that as long as as the values are identical before they are modified, they will be identical afterwards. Since they both start as 0, we can informally conclude that they will be equal at this point in the program. In 4.9.2 we will formally prove that they are equal.

ix. In lines 17–18 `main` cleans up. First, in line 17 the program frees the memory used, and then, in line 18, it exits by returning from

`main`.

x. Line 21 declares the child process `f`. The precondition of the function `f` is that the argument l is a lock with invariant $R(l)$, while the postcondition **emp** indicates that when the child terminates it will have given up all of its resources (including the lock passed as an argument).

xi. In lines 22–29, the child loops. It grabs the lock on line 23, and then modifies the protected data in lines 24–25. On line 26, the child tests to see if the first data cell is larger than 10; if so, it sets the continue flag to 0 in line 27, unlocks the lock in line 28, and exits the thread by returning from the function in line 29. Otherwise, the thread unlocks the lock in line 31, and loops back around.

The explanation given above indicates what the example program does and why it is correct. In section 4.9 we give parts of the formal proof of correctness. Mansky provides a fully systematic proof of correctness in Coq using our Concurrent Separation Logic in [Man08].

Chapter 4

Concurrent Separation Logic

Concurrent Separation Logic for Concurrent C minor is a new program logic for reasoning about the correctness of programs written in Concurrent C minor. Here we present the assertions of the logic, give the Hoare rules using those assertions to reason about the concurrent language features, and demonstrate the power of the system by verifying the example program presented in section 3.2. The semantics of assertions and the Hoare tuple are quite complicated and so are deferred until chapters 7 and 10, respectively[1].

4.1 Basic assertions

To support reasoning in Concurrent Separation Logic, we have many of the usual assertions of Hoare logic: **true** and **false**; conjunction $\wedge$ and disjunction $\vee$; universal and existential quantifiers $\forall$ and $\exists$; equire-

[1]Portions of this chapter have been published before as [HAZ08a] and [HAZ08b].

cursive assertions μF; assertions in the base (Coq) logic $[A]_{\mathsf{Coq}}$; and assertions that a C minor expression e evaluates to a value v, $e \Downarrow v$. C minor expressions can read the heap but not write to it. Most of our separation logic operators take only pure expressions, i.e., those that do not read the heap. Appel and Blazy [AB07, §4] show how to bridge the pure-impure gap in Separation Logic by local program transformations.

We support *impredicative* quantification—that is, our universal and existential quantifiers range over all types, including assertions themselves. This powerful form of quantification is useful for reasoning about complex language features such as objects, higher-order functional programs, and generics.

We do not expose full implication to the user, since it interacts complexly with the underlying model. We do support many kinds of limited implication, however. Similarily, we do not expose full negation. The problems surrounding implication are explained in section 7.3.7.

4.2 Sequential separation logic assertions

To these Hoare logic assertions we add the standard assertions of separation logic: the empty heap **emp**, which gives no permission to access memory; the singleton "maps-to" $e \mapsto v$, which gives permission to access the location e, and asserts that location e currently contains the value v[2]; and the separating conjunction $P * Q$, which says that the

[2]Since C minor allows for variable-sized data, the actual relation used in the proofs also handles the size of the data, but this detail has been elided for the

memory can be divided (not necessarily uniquely) into two parts, the first of which satisfies P and the second of which satisfies Q. We augment the maps-to assertion with the concept of *fractional ownership*.

4.3 Fractional ownership

First-class locks are only useful if more than one thread has the ability to try to grab the lock at the same time. Fractional ownership allows the the ownership of an address to be split into pieces. With partial ownership of a standard location, reading from that location is allowed but writing to it is not. With partial ownership of a lock location, trying to acquire the lock is allowed.

Fractional ownerships also enable the verification of concurrent algorithms that utilize a multiple-reader, single-writer protocol; moreover, they can also simplify sequential reasoning. For example, at a function call, a caller can pass just the read permission for an address to the callee. After the callee returns, the caller will know that the callee did not modify the address (but could have read it)[3].

We augment the maps-to relation with the additional parameter π, which is a *share*, resulting in $e \overset{\pi}{\mapsto} v$. A share indicates how much of the location is owned; we require that any share in a maps-to assertion be nonempty.

---

presentation.

[3]Further applications of fractional ownership can be found in Boyland [Boy03] and Bornat et al. [BCOP05].

Shares come with a *join relation* $\pi_1 \oplus \pi_2 = \pi$, which indicates that the shares π_1 and π_2 combine to form the new share π. Not every pair of shares π_1 and π_2 join—often there is no π that will serve. We write $\pi_a \perp \pi_b$ to mean that π_a and π_b join; that is, $\exists \pi_c.\ \pi_a \oplus \pi_b = \pi_c$. Conversely, we write $\pi_a \not\perp \pi_b$ to mean that π_a and π_b do not join; that is, $\pi_a \not\perp \pi_b \equiv \neg(\pi_a \perp \pi_b)$. We write $\pi_a \subset \pi_c$ to mean that π_a joins with some other share π_b to equal π_c; that is, $\pi_a \subset \pi_c \equiv \exists \pi_b.\ \pi_a \oplus \pi_b = \pi_c$.

4.4 Stratified separation algebras

Our shares and the join relation are similar to the notion of a *separation algebra* as defined by Calcagno et al. [COY07]. A separation algebra is a partial commutative monoid with a cancellation property. That is, a separation algebra has the following properties:

1. Determinism: $(\pi_a \oplus \pi_b = \pi_1) \Rightarrow (\pi_a \oplus \pi_b = \pi_2) \Rightarrow \pi_1 = \pi_2$
2. Associativity: $(\pi_1 \oplus \pi_2 = \pi_a) \Rightarrow (\pi_a \oplus \pi_3 = \pi) \Rightarrow$
   $\exists \pi_b.\ (\pi_1 \oplus \pi_b = \pi) \wedge (\pi_2 \oplus \pi_3 = \pi_b)$
3. Commutativity: $(\pi_1 \oplus \pi_2 = \pi) \Rightarrow (\pi_2 \oplus \pi_1 = \pi)$
4. Cancellation: $(\pi_1 \oplus \pi_a = \pi) \Rightarrow (\pi_2 \oplus \pi_a = \pi) \Rightarrow \pi_1 = \pi_2$
5. Identity: $\exists \pi_e.\ \forall \pi.\ \pi_e \oplus \pi = \pi$

(1)–(5) gives a clean, standard definition, and it is possible to develop a model for shares with it (the identity is the empty share). However,

it will not be sufficient when the join relation is extened to resource maps in chapter 7, since the model presented there has multiple distinct empty resource maps. From an engineering perspective the generalization to multiple identity elements is simpler if it is done uniformly, since it means a lemma library needs to be developed only once. Accordingly, we replace (5) with (6), which swaps the order of the quantifiers:

6. Multiple identities: $\forall \pi.\ \exists \pi_{e(\pi)}.\ \pi_{e(\pi)} \oplus \pi = \pi$

The idea of multiple identity elements in a commutative setting is unusual, since there is a standard proof that in such a setting, if two identity elements join with each other then they must be equal.

> **Lemma 4.1**. If π_{e1} and π_{e2} are identity elements, and if $\pi_{e1} \perp \pi_{e2}$ then $\pi_{e1} = \pi_{e2}$.
>
> **Proof**. $\pi_{e1} \perp \pi_{e2}$ means that there exists π such that $\pi_{e1} \oplus \pi_{e2} = \pi$. Since π_{e1} is an identity, we know that $\pi_{e1} \oplus \pi_{e2} = \pi_{e2}$. By cancellation, $\pi = \pi_{e2}$. By commutativity, we know $\pi_{e2} \oplus \pi_{e1} = \pi$. Since π_{e2} is an identity, $\pi_{e2} \oplus \pi_{e1} = \pi_{e1}$. By cancellation, we know that $\pi = \pi_{e1}$. Thus, by transitivity of equality, $\pi_{e1} = \pi_{e2}$. Proved in Coq. □

However, join is a partial function, which means that not every element must join with a given identity. Therefore, it is reasonable to imagine a set of identity elements, each of which can join with certain other elements. Since if two identity elements join they must be the same

element, this means that the sets of elements that join to particular empty elements form an equivalence class.

Lemma 4.2. If $\pi \oplus \pi = \pi$, then π is an identity.

Proof. Let π_e be the identity for π; thus $\pi_e \oplus \pi = \pi$. By cancellation, $\pi_e = \pi$. Therefore, π is an identity. Proved in Coq. □

This idea leads to a clean definition of identity: $\mathsf{identity}\ \pi \equiv \pi \oplus \pi = \pi$.

Lemma 4.3. If $\mathsf{identity}\ \pi$, and $\pi \perp \pi'$, then $\pi \oplus \pi' = \pi'$.

Proof. By lemma 4.2, π is an identity. Let π'_e be the identity for π'; thus $\pi'_e \oplus \pi' = \pi'$. By commutativity, $\pi' \oplus \pi'_e = \pi'$. By $\pi \perp \pi'$, there exists π'' such that $\pi \oplus \pi' = \pi''$. By commutativity, $\pi' \oplus \pi = \pi''$. By transivitity, $\pi \perp \pi'_e$. By lemma 4.1, $\pi = \pi'_e$. By substitution, $\pi \oplus \pi' = \pi'$. Proved in Coq. □

One additional property that we want for our join relation is

7. Split identities: $\forall \pi_a, \pi_b, \pi_e. (\pi_a \oplus \pi_b = \pi_e) \Rightarrow \mathsf{identity}\ \pi_e \Rightarrow \mathsf{identity}\ \pi_a$

This property says that if two elements join to an identity element, then both must be an identity element as well; it is a kind of monotonicity property. Property (7) is not true for all separation algebras (e.g., $\mathbb{Z}$ with addition); however, it is true for common examples of separation algebras (e.g., sets of $\mathbb{N}$ with the disjoint union operation), and does not

pose serious difficulties in a model. Accordingly, for us, a join relation has properties (1)–(4) and (6)–(7); it is a separation algebra with the pathological[4] combination of multiple and split identities, which we call a *stratified separation algebra*[5]. We call a type with a join relation that obeys the stratified separation algebra axioms a *joinable type*. Our Coq development has many different examples of joinable types.

4.5 Additional properties for shares

A share and its join operation must satisfy the properties of a joinable type from section 4.4. In addition a share must have some additional properties that are not always true for all separation algebras or separation logics:

8. Nonoverlapping: $\forall \pi_a, \pi_b, \pi_c.\ (\pi_a \oplus \pi_b = \pi_c) \Rightarrow \neg\mathsf{identity}\ \pi_a \Rightarrow \pi_a \not\searrow \pi_c$

9. Splittability: there is a named total function split, which takes a share and returns a pair of shares, and obeys the following rule:

$$\begin{aligned} \forall \pi, \pi_1, \pi_2.\quad & \mathsf{split}(\pi) = (\pi_1, \pi_2) \Rightarrow \\ & \pi_1 \oplus \pi_2 = \pi\ \wedge \\ & \neg\mathsf{identity}\ \pi \Rightarrow (\neg\mathsf{identity}\ \pi_1\ \wedge\ \neg\mathsf{identity}\ \pi_2) \end{aligned}$$

[4]This term is a mathematician's joke.

[5]In the Coq development, an uglier set of axoims is used. The equivalence of the presented axioms has been proven in Coq.

10. Intersection: $\pi_a \oplus \pi_b = \pi \ \wedge \ \pi_c \oplus \pi_d = \pi \ \Rightarrow$

    $\exists \pi_{ac}, \pi_{ad}, \pi_{bc}, \pi_{bd}.$

    $\pi_{ac} \oplus \pi_{ad} = \pi_a$

    $\pi_{bc} \oplus \pi_{bd} = \pi_b$

    $\pi_{ac} \oplus \pi_{bc} = \pi_c$

    $\pi_{ad} \oplus \pi_{bd} = \pi_d$

π	π_a	π_b
π_c	π_{ac}	π_{bc}
π_d	π_{ad}	π_{bd}

These axioms are provided so that the user of CSL is able to develop more modular proofs. Property (8) gives the user a finer ability to distinguish shares from one another[6]. Property (9) allows the user to develop proofs of modules without the need to handle a share that cannot be further split. Property (10), illustrated in the diagram above, guarantees a more subtle kind of splitting, and helps the user develop more modular proofs.

Properties (8)–(10) could probably be included in the definition of a joinable type instead of being guaranteed only for shares[7]. However, shares have two fundemental properties that distinguish them from other joinable types:

11. Full share: $\exists \pi_f. \ \forall \pi. \ \pi \subset \pi_f$

12. Nontrivial: $\exists \pi. \ \neg \mathsf{identity} \ \pi$

[6]The usefulness of this ability is discussed in section 4.6. Property (8) is similar in some ways to property (7) in that both in some sense constrain the existance of inverses, but the precise relationship of property (7) to property (8) is not fully clear.

[7]That (8)–(10) are defined only for shares, while (7) is defined for all joinable types could very well be historical accident relating to how the proof was developed; for some future work it might be useful to determine whether (8)–(10) should be moved to joinable, or perhaps whether (7) should be moved to shares.

Property (11) implies that shares have a partial order, with a (unique) top element, the full share. The partial ordering also guarantees the existence of a unique bottom element, the empty share[8]. Finally, property (12) guarantees that the system is nontrivial and completes the axiomitization. Accordingly, in our system, shares have properties (1)–(4) and (6)–(12)[9].

Lifting joinability from shares to assertions. Given the above properties for shares, we can model the separating conjunction as a join relation in a more general way than Appel and Blazy. Two maps-to assertions $e \overset{\pi_1}{\mapsto} v$ and $e \overset{\pi_2}{\mapsto} v$ separate if and only if $\pi_1 \perp \pi_2$. Two maps-to assertions about different locations always join. The assertion **emp** is the identity element.

4.6 Share models

The simplest model for fractional permissions is a rational number in $[0, 1]$. A 0 means that neither reading nor writing are allowed; any nonzero means that reading is allowed; and a 1 gives permission to write—so permission for writing implies permission for reading. In this simple model, the join relation is addition if the total is less than or equal to 1, and two shares which sum to more than 1 do not join.

[8]The existence of the unique identity element has been proven in Coq.

[9]Just as with joinable types, in the Coq development, an uglier set of axoims than (8)–(12) is used. It has been proven in Coq that the presented axioms imply the ones used in the Coq development, with one minor exception: in the Coq development, the full and empty shares are explicitly constructed, instead of being shown to exist existentially. This explicit construction is useful for engineering purposes.

As Parkinson explains in [Par05, ch. 5], this simple share model is not really sufficient for many applications. Among other problems, we cannot distinguish *which half* of the interval $[0, 1]$ is owned because two shares both equal to the same rational number are indistinguishable.

To model this finer distinction, Parkinson proposed that a share be a subset of the natural numbers. The full share is then $\mathbb{N}$, and the empty share the empty set. The join relation becomes the disjoint union operation[10]. Using this model, one can easily distinguish partial shares—two shares are equal only if the underlying sets are equal.

One drawback to using Parkinson's model for our work is that it does not support the splitting axiom, since Parkinson allows finite subsets. The natural strategy for modifying Parkinson's model to support the splitting axiom is to require that shares be infinite subsets of the naturals. Unfortunately, infinite subsets of the naturals do not support the intersection axiom, since the intersection of two infinite sets can be finite.

One mathematically elegant solution is to define a share to be an equivalence class of sets of real numbers taken from the interval $[0, 1]$, where two sets are in the same equivalence class if their symmetric difference has Lebesgue measure of 0. The join relation of two shares π_1 and π_2 is defined as long as

$$\exists S_1 \in \pi_1, S_2 \in \pi_2.\ S_1 \cap S_2 \text{ has measure } 0.$$

[10] That is, two shares only join if they are disjoint; the result of a successful join is their union.

If we write π to be the equivalence class containing $S_1 \cup S_2$, then $\pi_1 \oplus \pi_2 = \pi$. The join relation defined in this way will have all of the desired properties.

The problem with this model is that although it is mathematically clean, it would require developing measure theory in Coq, which is a sizable amount of work. A somewhat mathematically uglier technique is to define a share as the disjoint union of half-open intervals (open at the top) with rational endpoints drawn from the set $[0, 1)$, and to define the join operation as the disjoint union operation on sets. One advantage of this model is that it only uses the reals implicitly, since it manipulates only the rational endpoints. However, it still proved difficult to implement in Coq[11].

A third similar idea to use equivalence classes of reals where two sets are equivalent if their symmetric difference is countable. This is probably sufficient, but it was never implemented in Coq due to a general desire to avoid utilizing the real numbers unless absolutely necessary. Using equivalence classes of naturals where two sets are equivalent if their symmetric difference is finite might also work, but has not been extensively developed.

Dockins [Doc08] developed a share model in which a share is modeled by a binary-tree data structure. Models developed from a computer science perspective are sometimes easier to model in Coq than models

[11]In fact, it was so painful that for most of the development time shares were simply infinite subsets of the naturals, even though this meant that the intersection property was false.

developed from a mathematical perspective. Dockins's model satisfies all of our axioms, and is currently being used in the Coq development.

To simplify the remainder of the presentation we will use the following convention. ◆ indicates the full share[12], and ◇ indicates an empty share. Given a share π, we write $\lfloor\pi\rfloor$ to indicate the "left half" of $\mathsf{split}(\pi)$ and $\lceil\pi\rceil$ to indicate the "right half"; by the splitting rule, as long as π is not the empty share, then neither are $\lfloor\pi\rfloor$ or $\lceil\pi\rceil$. The left and right halves of the empty share are the empty share[13]: $\lfloor\diamond\rfloor = \lceil\diamond\rceil = \diamond$. For the convienece of notation, we will write ◀ for $\lfloor◆\rfloor$ and ▶ for $\lceil◆\rceil$. Therefore,

$$◀ \oplus ▶ = ◆ .$$

4.7 New assertions for CSL

To reason about first-class locks, we introduce the new assertion $e \overset{\pi}{\rightsquigarrow} R$, read "$e$ is a lock with share π and invariant R". It means that the expression e evaluates to a memory location l containing a lock that is associated with an assertion (in CSL) R, called the *resource invariant* of l. Just as with a maps-to assertion, we require that the share π be nonempty. A location cannot be both a normal memory cell and a lock. This restriction is enforced by not allowing a lock assertion and a maps-to assertion about the same address to separate from each other.

[12]That is, any share that satisfies property (11); since the top element is unique this is sufficient. As noted in footnote 9, in the Coq development the full share is given a name.

[13]This is guaranteed by property (7).

4.7.1 Resource invariants

The resource invariant R of a lock l must hold before a thread can unlock the lock l, or the machine will get stuck. This means that the next thread to lock l will know that R holds. Locks are an exclusive form of access control, meaning that only one thread can hold a lock at a given time. Therefore, the thread that grabs the lock will know that no other thread will be able to touch the resource until the lock is unlocked again.

Resource invariants for locks must be *closed to local variables*, *precise*, and *valid*. An invariant R is closed to local variables when R does not depend on the values of the C minor locals. Since local variables are not usually shared in concurrency, this restriction is natural[14].

An invariant R is precise when for all Q, the separating conjunction can only divide $R * Q$ in one way. An example of a precise predicate is a maps-to predicate. Requiring precision substantially simplifies the reasoning and lets the system be deterministic[15], and moreover is part of the standard presentation of CSL [O'H07].

The notion of validity is technical and will be deferred until chapter 7. However, in practice validity is not a difficult requirement to satisfy

[14]On the other hand, Concurrent C minor *can* express the idea of memory-resident local variables, by permitting each function invocation to have a stack-allocated memory block. Locations in these blocks can be addressed using load/store, and can be the subject of "maps-to" assertions. Stack-allocated addresses can be shared via the resource invariants of CSL. Before returning from a function, the thread must reclaim exclusive ownership of its stack block or the `return` command will get stuck.

[15]It is unclear what kinds of trade-offs are possible if one wishes to relax the precision requirement; Gotsman has recently been working in this area [Got08].

because all the operators mentioned above as being exposed to the user, as well as all the operators presented in section 4.7 are valid.

If an assertion R is closed, precise, and valid, then we say that R is *tight*. But how should we ensure that the user of our concurrent separation logic will always provide tight predicates? One choice is having side conditions, but this turns out to be annoying in the proofs. Instead, we utilize the new operators close, precisely, and validly. For all R, close R is closed, and moreover if R is closed, then close $R = R$. If R is not closed, then the meaning of the assertion close R depends on R and can be difficult to predict; in many cases it becomes **false**. precisely R and validly R are defined similarly with respect to precision and validity. The formal definitions of close, precisely, and validly are technical and so are deferred until section 7.3.8.

We define tightly as the composition of validly, close, and precisely: tightly R = validly (close (precisely R)). As expected, for all R, tightly R is tight, and tightly $R = R$ if and only if R is tight[16].

Using these definitions, it is possible to avoid stating explicit side conditions in the following way: when a lock with invariant R is unlocked, tightly R must hold; the next locking thread will then know tightly R holds after sucessfully acquiring the lock. If R is not tight, then the meaning of tightly R depends on R and is difficult to predict. Frequently it will be equal to **false**; in this case it will be impossible to

[16]The definition of tightly is somewhat subtle; for example, exchanging the order of the composition leads to a bad definition, because precision is a less orthogonal property than one might hope for.

unlock the lock.

4.7.2 Holds

Partial ownership of a lock is sufficient to attempt to lock it. A thread that succeeds in locking a lock also gains the additional permission $\mathsf{hold}\ e\ R$, which means that the expression e evaluates to a lock location l, with associated invariant R, and that the lock l is held by the thread in whose proof the hold assertion appears[17].

The simplest way to guarantee that the permission $\mathsf{hold}\ e\ R$ is required to unlock e is to require that all lock invariants R satisfy the relation $R = (\mathsf{hold}\ e\ R) * S$ for some additional assertion S. The simplest way to get this property is via the recursion operator μ:

$$R \ = \ \mu\alpha.\ (\mathsf{hold}\ e\ \alpha) \ * \ S.$$

Because $\mathsf{hold}\ e\ R$ is always closed and precise, as long as S is closed and precise then $(\mathsf{hold}\ e\ R) * S$ is as well. In other words, S is tight if and only if $(\mathsf{hold}\ e\ R) * S$ is tight. One advantage of this approach is that lock invariants cannot be **emp**, a fact that is useful when establishing soundness[18]. See section 4.9.1 for a sample resource invariant created with the recursion μ operator.

[17] Fractional ownership of the hold permission is possible, but does not seem to be very useful.

[18] In fact, an alternative way of getting soundness is to do away with hold altogether, and simply to require that lock invariants be nonempty. The problem with allowing lock invariants to be empty is that it allows a thread to unlock a lock twice.

4.7.3 Functions

We reason about first-class functions with the assertion $f : \{P\}\{Q\}$, which means that the expression f evaluates to a function pointer that points to a function with precondition P and postcondition Q. A precondition P of a function is actually a map from function arguments $\vec{e}$ to an assertion; similarly, the postcondition Q is a map from the values returned by the function to an assertion[19]. To allow a function's pre- and postconditions to be related to each other (e.g., function f returns an integer twice as large as its input), there is also an additional parameter for both P and Q. The ability to relate postconditions to preconditions is not so interesting for spawnable functions since the spawned functions never return to their parent. Therefore in this presentation we elide this extra parameter; see Appel and Blazy [AB07] for a discussion of how it is used for reasoning about sequential function call.

Since C minor separates the "data memory" from the "program memory," functions do not have shares, and every function is visible to every thread[20]. A function can either be called (within the current thread) or spawned (as a new thread). We require that the preconditions of functions that will be spawned be precise. We enforce this requirement with the validly and precisely operators analogously to our use of tightly for lock invariants. This way it is simple to determine

[19] Our version of C minor supports multiple return values.

[20] If one wished to verify self-modifying code, for example, functions would need shares.

$$\textsf{make\_lock}\ \frac{}{\Gamma \vdash \{e \overset{\blacklozenge}{\mapsto} 0\}\ \textsf{make\_lock}\, e\, R\ \{e \overset{\blacklozenge}{\rightsquigarrow} R\ *\ \textsf{hold}\, e\, R\}}$$

$$\textsf{free\_lock}\ \frac{}{\Gamma \vdash \{e \overset{\blacklozenge}{\rightsquigarrow} R\ *\ \textsf{hold}\, e\, R\}\ \textsf{free\_lock}\, e\ \{e \overset{\blacklozenge}{\mapsto} 0\}}$$

$$\textsf{lock}\ \frac{}{\Gamma \vdash \{e \overset{\pi}{\rightsquigarrow} R\}\ \textsf{lock}\, e\ \{e \overset{\pi}{\rightsquigarrow} R\ *\ \textsf{tightly}\ R\}}$$

$$\textsf{unlock}\ \frac{R = (\textsf{hold}\, e\, R\ *\ S)}{\Gamma \vdash \{\textsf{tightly}\ R\}\ \textsf{unlock}\, e\ \{\mathbf{emp}\}}$$

$$\textsf{fork}\ \frac{}{\Gamma \vdash \{f : \{P\}\{Q\}\ *\ \textsf{validly precisely}\ P(\vec{e})\}\ \textsf{fork}\, f\, \vec{e}\ \{f : \{P\}\{Q\}\}}$$

Figure 4.1: Concurrent Separation Logic

which resources the parent process gives to its new child.

4.8 Concurrent separation logic Hoare rules

One of the most important aspects of O'Hearn's CSL is that it includes all of the rules of standard separation logic [O'H07]. Similarily, our CSL rules are a superset of the rules of Appel & Blazy presented in figure 2.6. Our new concurrent rules are presented in figure 4.1.

To use the make_lock rule to reason about making a lock with invariant R at location e, full ownership of e is required. Moreover, that location must contain a 0; the operational semantics described in sec-

tion **??** will get stuck if these conditions are not met. Since a lock is considered to be locked when its location contains a 0, it is therefore created in the locked state, and the postcondition includes both the full visibility of the lock $e \overset{\frown}{\rightsquigarrow} R$ and the holding of the lock $\mathsf{hold}\ e\ R$.

Both $e \overset{\frown}{\rightsquigarrow} R$ and $\mathsf{hold}\ e\ R$ are required to reason about freeing a lock using the free_lock rule. Since a thread can have a hold permission only if the lock is locked, this means that the location associated with the lock contains a 0.

To use the lock rule to reason about locking a lock e with associated invariant R, a thread needs to own some (typically partial) share π of e. After locking, the thread still has the same visibility of the lock as before, and in addition gains the assertion $\mathsf{tightly}\ R$. If R is tight, then $\mathsf{tightly}\ R$ is equal to R.

The precondition for using the unlock rule for reasoning about unlocking a lock is $\mathsf{tightly}\ R$. We also require as a side condition that the lock invariant R be equal to $(\mathsf{hold}\ e\ R) * S$ for some S[21]. Again, as long as R is tight (which is true if and only if S is tight) then the precondition is equal to R. If R is not tight, then $\mathsf{tightly}\ R$ is equal to **false**, meaning that the unlock statement cannot be proven to be safe using CSL.

The rule for fork is somewhat analogous to the rule for unlock. This is natural since in both cases resources are being transferred from the

[21] In the Coq development, this property is enforced not by a side condition but instead by using another operator like tightly; the presentation here is equivalent and cleaner.

current thread into the lock for unlock or into the child thread for fork. The precondition requires a function e with precondition P and postcondition Q. Just as for unlock, we apply an operator to force the precondition to have the desired property, in this case only precision[22].

Since the precondition P is a map from the arguments $\vec{e}$ to an assertion, the precondition of the fork rule requires that the assertion precisely $P(\vec{e})$ hold. Just as in the case of unlock, if $P(\vec{e})$ is precise, precisely $P(\vec{e}) = P(\vec{e})$; otherwise it will be equal to **false**, meaning that the fork statement in question cannot be proven to be safe using CSL. After the fork statement, the parent thread loses the assertion precisely $P(\vec{e})$. The child process will then start with the assertion precisely $P(\vec{e})$ when it starts.

4.9 Applying CSL to the example program

We demonstrate the usefulness of our CSL by demonstrating how to apply it to critical parts of the example program from section 3.2.

4.9.1 The resource invariant $R(l)$

The key resource invariant is given in figure 4.2. It is divided into two parts: $S(l, P)$ and $R(l)$. In effect, $S(l, P)$ is specialized to each lock,

[22]In the Coq development, precision is guaranteed with a side condition; the presentation here is equivalent and more consistent with the other rules.

l	*lock*
$l+1$	*data*
$l+2$	*data*
$l+3$	*continue?*

1 $\quad S(l,P) \quad = \quad (\exists v. l+1 \overset{\blacklozenge}{\mapsto} v \ * \ l+2 \overset{\blacklozenge}{\mapsto} v) *$

2 $\qquad\qquad\qquad ((l+3 \overset{\blacklozenge}{\mapsto} 1) \vee ((l+3 \overset{\blacklozenge}{\mapsto} 0) * l \overset{\blacklozenge}{\rightsquigarrow} P))$

3 $\quad R(l) \quad = \quad \mu P.\ (\mathsf{hold}\ l\ P) * S(l,P)$

Figure 4.2: Resource invariant for example program

while the $R(l)$ definition is boilerplate. The convention used when defining $S(l,P)$ is that l is the address of the lock and P is the lock's invariant, which will include $S(l,P)$. Because we are using the recursion operator μ to form resource invariants, $S(l,P)$ must be contractive in P; as long as P is used inside a lock or hold assertion then this requirement is easily satisfied. The formal definition of contractiveness is technical and is deferred until section 7.3.9.

In line 1, $S(l,P)$ implies that the locations $l+1$ and $l+2$ are fully owned and point to the same value. In line 2, $S(l,P)$ also implies that the location $l+3$ is owned, and is being used as a signal to communicate that the child thread is done processing. While the child is working, $l+3$ is set to 1, to indicate that it is continuing to compute. When the child is done, it sets $l+3$ to 0, and gives up the location l, which is a lock with invariant P (fufilling the convention for the boilerplate), and which is partially owned with the left half of the full share. This allows the child to return all of its resources to its parent cleanly, and thus exit holding

no resources.

$S(l, P)$ is contractive in P since P is only used inside a lock invariant. $S(l, P)$ is precise since the separating conjunction of two precise predicates is precise and since exactly one half of the disjunction on line 2 is true at any time. Moreover, $S(l, P)$ is valid since it is built from the operators presented in sections 4.1 and 4.7. By inspection, $S(l, P)$ is closed to local variables. Therefore $S(l, P)$ is tight.

The boilerplate definition for $R(l)$ is given on line 3. It uses the recursive operator μ to tie the knot and ensures that the lock invariant implies that the lock is held, as required for the unlock rule. Since $S(l, P)$ is contractive, μ is well-defined. Since $S(l, P)$ is tight, so is $R(l)$.

4.9.2 Verification of selected instructions

Here we show how to apply the rules of CSL to verify parts of the example program. For cross-referencing purposes, we use the same numbering here as in section 3.2. For clarity we omit the use of Γ (the map from functions to specifications) in the presentation of our rules.

i. The pre- and postconditions of `main` are **emp**. Since **emp** is the identity element, a precondition of **emp** indicates that main can be called at any time. The postcondition of **emp** indicates that when it exits, `main` will no longer own any resources. In traditional separation logic this would mean that all of the resources allocated by `main` had been freed. However, in CSL it is possible for resources to be allocated and then given away in a lock;

this means that a postcondition of **emp** is not strong enough to guarantee that all resources have been freed on program termination. There are several approaches to guaranteeing this stronger property, such as requiring certain programming disciplines, or, alternatively, forcing lock invariants to have certain properties, an approach utilized by Gotsman et al. in [GBC+07].

ii. Lines 2 and 3 of the example program require only standard separation logic. It is important that L be initialized to 0 for (iii).

iii. The make_lock in main turns location L into a lock with invariant $R(\mathsf{L})$. The postcondition of line 3 is

$$\{\ \mathsf{i} = 0\ *\ \mathsf{L} \overset{\blacktriangle}{\mapsto} 0\ *\ \mathsf{L}+1 \overset{\blacktriangle}{\mapsto} 0\ *\ \mathsf{L}+2 \overset{\blacktriangle}{\mapsto} 0\ *\ \mathsf{L}+3 \overset{\blacktriangle}{\mapsto} 0\ \}.$$

Using the frame rule, we abstact away everything except

$$\{\ \mathsf{L} \overset{\blacktriangle}{\mapsto} 0\ \}.$$

Then we are able to use the rule for make_lock:

$$\{\ \mathsf{L} \overset{\blacktriangle}{\mapsto} 0\ \}$$
$$\mathsf{make\_lock}\ \mathsf{L}\ R(\mathsf{L})$$
$$\{\ \mathsf{L} \overset{\blacktriangle}{\rightsquigarrow} R(\mathsf{L})\ *\ \mathsf{hold}\ \mathsf{L}\ R(\mathsf{L})\ \}$$

The lock invariant is not yet satisfied, but it need not be satisfied until the lock is unlocked.

iv. When function f is spawned, main passes to f half of the visibility of lock L, as stated in f's precondition on line 21. The relevant postcondition from the previous line (omitting the rest of the frame) is

$$\{\ \mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\ \}.$$

Function assertions $f : \{P\}\{Q\}$ are kept in the global environment Γ. We are allowd to borrow them from Γ as long as we return them[23]. We borrow from Γ the specification of f, leading to

$$\{\ \mathsf{f} : \{\mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\}\{\mathbf{emp}\}\ *\ \mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\ \},$$

which is equal to

$$\{\ \mathsf{f} : \{\mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\}\{\mathbf{emp}\}\ *\ \mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\ *\ \mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\ \}.$$

Again using the frame rule, we are able to abstract to

$$\{\ \mathsf{f} : \{\mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\}\{\mathbf{emp}\}\ *\ \mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\ \}.$$

Since $\mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})$ is valid and precise[24], this is equal to

$$\{\ \mathsf{f} : \{\mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\}\{\mathbf{emp}\}\ *\ \mathsf{precisely}\ \mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\ \},$$

[23] For full details on the rules for borrowing from Γ, see [AB07].

[24] Like a maps-to assertion, the "is a lock with invariant R" assertion is valid and precise.

which allows us to use the fork rule as follows:

$$\{\ \mathsf{f} : \{\mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\}\{\mathbf{emp}\}\ *\ \mathsf{precisely}\ \mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\ \}$$
$$\mathsf{fork}\ \mathsf{f}(\mathsf{L})$$
$$\{\ \mathsf{f} : \{\mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\}\{\mathbf{emp}\}\ \}$$

Therefore, the spawned child thread is given the left half of lock L, and the parent retains the right half. We then return the specification of `f` to Γ.

v. Although the assignment on line 6 was worrisome in the informal presentation, in the formalism it is quite straightforward. A part of the frame from (iv) that comes from (iii) is

$$\{\ \mathsf{L}+3 \overset{\blacktriangleleft}{\mapsto} 0\ \},$$

which the standard separation logic rule for store instructions relates to postcondition

$$\{\ \mathsf{L}+3 \overset{\blacktriangleleft}{\mapsto} 1\ \}.$$

vi. The full precondition of the loop is

$$\{\ \mathsf{i}=0\ *\ \mathsf{L}+1 \overset{\blacktriangleleft}{\mapsto} 0\ *\ \mathsf{L}+2 \overset{\blacktriangleleft}{\mapsto} 0\ *\ \mathsf{L}+3 \overset{\blacktriangleleft}{\mapsto} 1\ *$$
$$\mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\ *\ \mathsf{hold}\ \mathsf{L}\ R(\mathsf{L})\ \},$$

which by fold-unfold is equal to

$$\{\ \mathsf{i} = 0\ *\ \mathsf{L} \overset{\blacktriangle}{\rightsquigarrow} R(\mathsf{L})\ *\ R(\mathsf{L})\ \},$$

which implies the loop invariant of

$$\{\ \mathsf{i} = \_\ *\ \mathsf{L} \overset{\blacktriangle}{\rightsquigarrow} R(\mathsf{L})\ *\ R(\mathsf{L})\ \}.$$

We write "_" to indicate that `i` has some arbitrary value. The postcondition of the loop is

$$\{\ \mathsf{i} = \_\ *\ (\exists k.\ (\mathsf{L}+1) \overset{\blacktriangle}{\mapsto} k\ *\ (\mathsf{L}+2) \overset{\blacktriangle}{\mapsto} k)\ *\ (\mathsf{L}+3) \overset{\blacktriangle}{\mapsto} 0\ *$$
$$\mathsf{L} \overset{\blacktriangle}{\rightsquigarrow} R(\mathsf{L})\ *\ \mathsf{hold}\ \mathsf{L}\ R(\mathsf{L})\ \}.$$

On line 8, the test `*(L+3) == 0` is done to break out of the loop. Given that the precondition of line 8 is the loop invariant, if the test succeeds, then the postcondition of the loop will hold by fold-unfold. If the test fails, then the state will not change, and the loop invariant will hold after line 8. If $R(\mathsf{L})$ holds before line 9, then it will hold afterwards. Therefore the precondition to the unlock statement on line 10 is

$$\{\ \mathsf{i} = \_\ *\ \mathsf{L} \overset{\blacktriangle}{\rightsquigarrow} R(\mathsf{L})\ *\ R(\mathsf{L})\ \}.$$

Using the frame rule, we abstract to

$$\{\ R(\mathsf{L})\ \}.$$

Since $R(\mathsf{L})$ is tight, this is equal to

$$\{\ \mathsf{tightly}\ R(\mathsf{L})\ \}.$$

Since the side condition of the unlock rule proceeds from the definition of $R(\mathsf{L})$, we can apply the rule as follows:

$$\begin{array}{c}\{\ \mathsf{tightly}\ R(\mathsf{L})\ \}\\ \mathsf{unlock}\,\mathsf{L}\\ \{\ \mathbf{emp}\ \}\end{array}$$

Line 11 is straightforward, and has the postcondition

$$\{\ \mathsf{i} = \_ \ *\ \mathsf{L} \overset{\blacktriangleright}{\rightsquigarrow} R(\mathsf{L})\ \}.$$

After applying the frame rule to get

$$\{\ \mathsf{L} \overset{\blacktriangleright}{\rightsquigarrow} R(\mathsf{L})\ \},$$

we can apply the lock rule for line 12:

$$\{\ \mathsf{L} \overset{\blacktriangle}{\rightsquigarrow} R(\mathsf{L})\ \}$$
$$\mathsf{lock}\,\mathsf{L}$$
$$\{\ \mathsf{L} \overset{\blacktriangle}{\rightsquigarrow} R(\mathsf{L})\ *\ \mathsf{tightly}\ R(\mathsf{L})\ \}$$

By the tightness of $R(\mathsf{L})$ this is equal to

$$\{\ R(\mathsf{L})\ \},$$

and by adding the frame back in we can prove the loop invariant.

vii. The postcondition of the loop given in (vi) implies that `main` has reacquired the full ownership of lock L. Accordingly, after abstracting away the frame we can apply the free_lock rule:

$$\{\ \mathsf{L} \overset{\blacktriangle}{\rightsquigarrow} R(\mathsf{L})\ *\ \mathsf{hold}\ \mathsf{L}\ R(\mathsf{L})\ \}$$
$$\mathsf{free\_lock}\ L$$
$$\{\ \mathsf{L} \overset{\blacktriangle}{\mapsto} 0\ \}$$

viii. The postcondition of the loop given in (vi) also implies that $\mathtt{L}+1$ and $\mathtt{L}+2$ point to the same value, so the test on 15 will always succeed and the program will not get stuck on line 16.

ix. The cleanup on lines 17–18 follows from the standard rules of separation logic, and implies the postcondition **emp** of `main`.

x. The precondition of `f` was discussed above in (iv), and the mean-

ing of **emp** as a postcondition was discussed in (i).

xi. The precondition and loop invariant of the loop starting on line 22 is

$$\{\ \mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\ \}.$$

The postcondition of the loop is **emp**. The lock on line 23 is straightforward:

$$\{\ \mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\ \}$$
$$\mathsf{lock}\,\mathsf{L}$$
$$\{\ \mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\ *\ \mathsf{tightly}\ R(\mathsf{L})\ \}$$

By the tightness of $R(\mathsf{L})$, this is equal to

$$\{\ \mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\ *\ R(\mathsf{L})\ \}.$$

On line 24, the loop invariant $R(\mathsf{L})$ is temporarily broken; $\mathsf{L} + 1$ does not point to the same value as $\mathsf{L} + 2$. However, this is permitted since the lock invariant does not have to hold while the lock is held. On line 25, the invariant is restored, leading to a postcondition of

$$\{\ \mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\ *\ R(\mathsf{L})\ \}.$$

If the test on line 26 succeeds, we reach line 27 with precondition

$$\{\ \mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\ *\ R(\mathsf{L})\ \}.$$

Unfolding $R(\mathsf{L})$, we get

$$\{\ \mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\ *\ (\exists k.\ (\mathsf{L}+1) \overset{\blacktriangleleft}{\mapsto} k\ *\ (\mathsf{L}+2) \overset{\blacktriangleleft}{\mapsto} k) *$$
$$((\mathsf{L}+3 \overset{\blacktriangleleft}{\mapsto} 1) \vee ((\mathsf{L}+3 \overset{\blacktriangleleft}{\mapsto} 0) * l \overset{\blacktriangleleft}{\rightsquigarrow} P))\ *\ \mathsf{hold}\ \mathsf{L}\ R(\mathsf{L})\ \}.$$

Since $\mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})$ does not join with itself, we know that we have the left half of the disjunct, yielding

$$\{\ \mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\ *\ (\exists k.\ (\mathsf{L}+1) \overset{\blacktriangleleft}{\mapsto} k\ *\ (\mathsf{L}+2) \overset{\blacktriangleleft}{\mapsto} k) *$$
$$(\mathsf{L}+3 \overset{\blacktriangleleft}{\mapsto} 1)\ *\ \mathsf{hold}\ \mathsf{L}\ R(\mathsf{L})\ \}.$$

The assignment on line 27 then results in postcondition

$$\{\ \mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L})\ *\ (\exists k.\ (\mathsf{L}+1) \overset{\blacktriangleleft}{\mapsto} k\ *\ (\mathsf{L}+2) \overset{\blacktriangleleft}{\mapsto} k) *$$
$$(\mathsf{L}+3 \overset{\blacktriangleleft}{\mapsto} 0)\ *\ \mathsf{hold}\ \mathsf{L}\ R(\mathsf{L})\ \},$$

which via fold is equal to

$$\{\ R(\mathsf{L})\ \}.$$

By the tightness of $R(\mathsf{L})$, this is equal to

$$\{\ \mathsf{tightly}\ R(\mathsf{L})\ \},$$

which lets us use the unlock rule for line 28:

$$\{ \text{ tightly } R(\mathsf{L}) \}$$
$$\text{unlock}\,\mathsf{L}$$
$$\{ \mathbf{emp} \}$$

This gives us the postcondition **emp**, which satisfies the postcondition given in line 21, so we can safely return on line 29.

Notice that the child thread gave up *all* its resources before returning. It did so by unlocking the lock L. However, one of the resources it gave up by unlocking L was the visibility of L itself! In general we want every thread to give up all resources before exiting, and in general it must do so by unlocking a lock. We accomplish this by taking advantage of the recursion operator as demonstrated above.

Now the main thread can safely dispose the lock L, as explained in (vii).

If the test on line 26 does not succeed, state is not modified on line 30, giving a precondition of

$$\{ \mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L}) \ * \ R(\mathsf{L}) \}$$

for the unlock on line 31. Applying the frame rule followed by the

unlock rule results in a postcondition of

$$\{ \mathsf{L} \overset{\blacktriangleleft}{\rightsquigarrow} R(\mathsf{L}) \},$$

which implies the loop invariant.

In [Man08], Mansky verifies this program with our CSL in Coq; in addition to filling in various details omitted above, he gives a sense for the process of verifying a program in Coq. Furthermore, taking a lesson from Knuth, he actually implemented the algorithm in C (with POSIX threads) and tested it.

4.10 Conclusions

We have presented a new concurrent separation logic. The logic includes many powerful assertions, including impredicative quantification, recursive invariants, fractional permissions, and first-class locks and function pointers.

We have given Hoare rules for using these assertions to reason about Concurrent C minor programs, including rules for `lock`, `unlock`, and `fork`. We have developed new operators such as `tightly` to express those rules cleanly. Finally we have demonstrated how to use these rules and operators to verify the example program from section 3.2.

In the remainder of this thesis we will present a soundness proof of our concurrent separation logic with respect to Concurrent C minor.

Chapter 5

Erased Concurrent Operational Semantics

In chapter 3 we introduced Concurrent C minor, and in chapter 4 we gave it an axiomatic semantics, Concurrent Separation Logic. This chapter gives Concurrent C minor a formal concurrent operational semantics, which we call an *erased semantics*. In chapters 8, 9, and 10, we will prove the soundness of CSL with respect to the erased semantics[1].

The erased concurrent operational semantics justifies the claim that Concurrent C minor has a reasonable model of conventional concurrency. It is not easy to prove properties about the erased semantics. It is easier to reason about an *unerased semantics*, which tracks additional bookkeeping information (see chapter 8). In section 8.6 we prove an erasure theorem that says that the unerased semantics is a con-

[1]Portions of this chapter have been published before as [HAZ08a] and [HAZ08b].

servative approximation to the erased semantics given here. Then in chapter 9 we define an oracular semantics that is suitable for reasoning about one thread at a time. In chapter 10 we model our CSL on that oracular semantics, and prove that properties proved with the oracular semantics holds on the unerased concurrent operational semantics. The erasure theorem then guarantees that the properties hold on our erased semantics.

One problem with the erased semantics given here is that it does not compose well with the compiler correctness proofs. If we erase too soon, at the source language, then the compiler will not be able to rely on the bookkeeping features of the unerased semantics in its proofs; as discussed further in section 8.6, the correct time to erase is at the very end of compilation, on the machine code.

In section 5.1 we give a vastly simplified semantics for sequential C minor because the complexities of the sequential language are not the focus of this thesis; in the Coq development we utilize the full C minor of Appel and Blazy [AB07]. In section 5.2, we give the erased concurrent step relation to show that we have a reasonable model for concurrency. Our concurrent semantics has an unusual interleaving model; in section 5.3 we discuss why the interleaving model is reasonable.

5.1 Erased sequential step relation

Here we present a highly simplified version of sequential C minor; the Coq development uses the full sequential C minor of Appel and Blazy, which is given in figure 2.3.

The syntax of simplified C minor is given by the following grammar:

$$
\begin{array}{lcll}
v & : \quad \mathsf{value} & := & n \\
s & : \quad \mathsf{statement} & := & [v_1]{:=}v_2 \\
 & & & \mathsf{call}\ f\ \vec{v} \\
 & & & s_1; s_2 \\
 & & & \mathsf{make\_lock}\ v\ P \\
 & & & \mathsf{free\_lock}\ v \\
 & & & \mathsf{lock}\ v \\
 & & & \mathsf{unlock}\ v \\
 & & & \mathsf{fork}\ f\ \vec{v}
\end{array}
$$

A value v is a natural number n. There are eight different kinds of statements: three sequential statements, given as simplified examples of the kinds of statements in C minor, and five concurrent statements. The store statement $[v_1]{:=}v_2$ updates the memory at location v_1 to have value v_2. The call statement $\mathsf{call}\ f\ \vec{v}$ starts a subroutine function f, passing arguments v. The sequence statement $s_1; s_2$ runs the statement s_1 followed by the statement s_2. The make_lock statement and the free_lock statement are bookkeeping statements and do not change

$$\text{er-sstep-update}\ \frac{m' = [v_1 \mapsto v_2]\ m}{\Psi \vdash ((\rho, m), [v_1]{:=}v_2 \cdot \kappa) \overset{\mathsf{e}}{\mapsto} ((\rho, m'), \kappa)}$$

$$\text{er-sstep-call}\ \frac{}{\Psi \vdash ((\rho, m), \mathsf{call}\, f\, \vec{v} \cdot \kappa) \overset{\mathsf{e}}{\mapsto} ((\rho, m), (\Psi(f)\ \vec{v}) \cdot \kappa)}$$

$$\text{er-sstep-seq}\ \frac{}{\Psi \vdash ((\rho, m), s_1; s_2 \cdot \kappa) \overset{\mathsf{e}}{\mapsto} ((\rho, m), s_1 \cdot s_2 \cdot \kappa)}$$

$$\text{er-sstep-makelock}\ \frac{}{\Psi \vdash ((\rho, m), \mathsf{make\_lock}\, v\, P \cdot \kappa) \overset{\mathsf{e}}{\mapsto} ((\rho, m), \kappa)}$$

$$\text{er-sstep-freelock}\ \frac{}{\Psi \vdash ((\rho, m), \mathsf{free\_lock}\, v \cdot \kappa) \overset{\mathsf{e}}{\mapsto} ((\rho, m), \kappa)}$$

Figure 5.1: Erased sequential step relation

the state. The lock statement waits until the lock v is unlocked, and then locks the lock. The unlock statement unlocks the lock v. The fork statement starts a new thread with a function call to f.

A sequential computation state, also called an erased world w, is a pair of locals ρ (a map from identifiers to values) and memory m; in our simplified syntax, ρ is unused, but it is used in the full C minor and so is included here.

A control contains program syntax and has the following grammar:

$$\begin{array}{rcll} \kappa & : & \mathsf{control} & := \quad s \cdot \kappa \\ & & & \quad\ \mathsf{Kstop} \end{array}$$

A function body is a map from arguments $\vec{v}$ to statement s. A program Ψ is a function from addresses to function bodies.

In figure 5.1, we define our erased sequential step relation with five

cases. The first case, er-sstep-update, handles the store statement by updating the memory at location v_1 to have value v_2 and advancing to the next statement κ. The second case, er-sstep-call, performs a function call by replacing the call statement with the body of the function $\Psi(f)$ and applying the arguments $\vec{v}$. The third case, er-sstep-seq, takes apart a sequence statement. The fourth case, er-sstep-makelock, and fifth case, er-sstep-freelock, do nothing other than advance to the next statement κ.

For the fully concurrent instructions lock, unlock, and fork, the erased sequential step relation gets stuck.

5.2 Erased concurrent step relation

A schedule $\mho$ is a **finite** list of natural numbers that act as thread-IDs. The number at the head of the schedule tells the concurrent operational semantics which thread to execute next. When the schedule runs out, the concurrent machine safely halts computation.

With a fixed schedule, our semantics is fully deterministic, which simplifies the proofs, particularly proofs about sequential features that assume determinacy. Of course, concurrent systems are not usually considered to be deterministic. We ensure that the proved properties hold regardless of the way threads interleave by universally quantifying over all schedules.

We use a finite schedule because it allows us to do induction easily.

We do not restrict the length of a schedule, so when we quantify over all schedules we include schedules of arbitrary length. Therefore our proved properties will be true for any finite amount of time.

A thread θ is a pair of local state (locals ρ) and concurrent control $\hat{\kappa}$. We do not put the memory in a thread because all the threads must share the same memory. A concurrent control $\hat{\kappa}$ contains the code (*i.e.*, the program syntax) of the thread, defined as follows:

$$\begin{array}{rcl} \hat{\kappa} : \mathsf{concurrent\ control} & = & \mathsf{Krun}\,\kappa \\ & | & \mathsf{Klock}\,v\,\kappa. \end{array}$$

$\mathsf{Krun}\,\kappa$ means that the thread is in a runnable state, with κ as the next control to execute. $\mathsf{Klock}\,v\,\kappa$ means that the thread is waiting on a lock at address v; after acquiring the lock it will continue with κ.

We denote a thread list by $\vec{\theta}$, and the ith thread by $\vec{\theta}_i$. A concurrent state S is a tuple of schedule $\mho$, thread list $\vec{\theta}$, and memory m.

In figure 5.2 we present our erased concurrent step relation with seven cases.

The first case, er-cstep-seq, covers running a sequential statement (including make_lock and free_lock). We select thread i, add the memory m, and run the sequential step relation; afterwards we put the thread back into place with new locals ρ' and control κ' and continue stepping with new memory m'. We do not change the schedule, and so do not context switch.

The second case, er-cstep-texit, covers what happens when we run

$$\text{er-cstep-seq}\ \frac{\begin{array}{c}\vec{\theta}_i = (\rho, \mathsf{Krun}\,\kappa) \\ \Psi \vdash ((\rho, m), \kappa) \overset{\mathsf{e}}{\longmapsto} ((\rho', m'), \kappa') \\ \vec{\theta'} = [i \mapsto (\rho', \mathsf{Krun}\,\kappa')]\ \vec{\theta}\end{array}}{\Psi \vdash (i :: \mho, \vec{\theta}, m) \overset{\mathsf{e}}{\Longmapsto} (i :: \mho, \vec{\theta'}, m')}$$

$$\text{er-cstep-texit}\ \frac{\vec{\theta}_i = (\rho, \mathsf{Krun}\,(\mathsf{Kstop}))}{\Psi \vdash (i :: \mho, \vec{\theta}, m) \overset{\mathsf{e}}{\Longmapsto} (\mho, \vec{\theta}, m)}$$

$$\text{er-cstep-prelock}\ \frac{\begin{array}{c}\vec{\theta}_i = (\rho, \mathsf{Krun}\,\mathsf{lock}\,v \cdot \kappa) \\ \vec{\theta'} = [i \mapsto (\rho, \mathsf{Klock}\,v\,\kappa)]\ \vec{\theta}\end{array}}{\Psi \vdash (i :: \mho, \vec{\theta}, m) \overset{\mathsf{e}}{\Longmapsto} (\mho, \vec{\theta'}, m)}$$

$$\text{er-cstep-nolock}\ \frac{\vec{\theta}_i = (\rho, \mathsf{Klock}\,v\,\kappa) \qquad m(v) = 0}{\Psi \vdash (i :: \mho, \vec{\theta}, m) \overset{\mathsf{e}}{\Longmapsto} (\mho, \vec{\theta}, m)}$$

$$\text{er-cstep-lock}\ \frac{\begin{array}{c}\vec{\theta}_i = (\rho, \mathsf{Klock}\,v\,\kappa) \\ m(v) = 1 \qquad m' = [v \mapsto 0]\ m \\ \vec{\theta'} = [i \mapsto (\rho, \mathsf{Krun}\,\kappa)]\ \vec{\theta}\end{array}}{\Psi \vdash (i :: \mho, \vec{\theta}, m) \overset{\mathsf{e}}{\Longmapsto} (i :: \mho, \vec{\theta'}, m')}$$

$$\text{er-cstep-unlock}\ \frac{\begin{array}{c}\vec{\theta}_i = (\rho, \mathsf{Krun}\,\mathsf{unlock}\,v \cdot \kappa) \\ m(v) = 0 \qquad m' = [v \mapsto 1]\ m \\ \vec{\theta'} = [i \mapsto (\rho, \mathsf{Krun}\,\kappa)]\ \vec{\theta}\end{array}}{\Psi \vdash (i :: \mho, \vec{\theta}, m) \overset{\mathsf{e}}{\Longmapsto} (\mho, \vec{\theta'}, m')}$$

$$\text{er-cstep-fork}\ \frac{\begin{array}{c}\vec{\theta}_i = (\rho, \phi, \mathsf{Krun}\,\mathsf{fork}\,v\,\vec{v} \cdot \kappa) \\ \vec{\theta'} = [i \mapsto (\rho, \mathsf{Krun}\,\kappa)]\ \vec{\theta} \\ \vec{\theta''} = \vec{\theta'} + ((\rho_0, \mathsf{Krun}\,(\mathsf{call}\,v\,\vec{v} \cdot \mathsf{Kstop})) :: \mathtt{nil})\end{array}}{\Psi \vdash (i :: \mho, \vec{\theta}, m) \overset{\mathsf{e}}{\Longmapsto} (\mho, \vec{\theta''}, m)}$$

Figure 5.2: Erased concurrent step relation

out of statements in a thread; in this case we switch to another thread by removing the head of the scheduler. We do not delete thread θ_i from $\vec{\theta}$, so threads never change position in line. If the thread is selected again, we will just context switch again.

The third case, er-cstep-prelock, covers what happens when a thread wants to start grabbing a lock. We move to the $\mathsf{Klock}\, v\, ctl$ state and context switch so that other threads have a chance to grab the lock.

The fourth case, er-cstep-nolock, covers the "spin" case of the spin lock. In the memory m lock value is 0, indicating that the lock is locked by some other thread. We context switch and keep waiting for the lock.

The fifth case, er-step-lock, covers the case when we grab the lock. In the memory m the lock has value 1, indicating that it is unlocked, and so we switch it to value 0, so that no other thread can grab the lock. At a lower level of the compiler this will become a test-and-set or compare-and-swap instruction, making the load from m and the store to m' atomic at all intermediate compiler levels. We then transition to the runnable $\mathsf{Krun}\, \kappa$ state and start running the thread.

The sixth case, er-step-unlock, covers the case when we release a lock. In the memory m we test to make sure that the lock is currently locked, and then update it so that it is unlocked. We context switch after an unlock so that other threads can tell that we have released it.

The seventh case, er-step-fork, covers the case when we fork a child thread. In that case we add the new thread to the thread list and context switch so that it has a chance to run.

5.3 Reasonableness of interleaving model

Our semantics has a nonpreemptive thread model. We chose this concurrency model because a sequentially consistent interleaving model is inappropriate for our context.

Sequentially consistent interleaving is too weak because actual processors go beyond interleaving at every step. They have *weak memory models*, where instructions are dynamically reordered by the processor in a way that is sequentially undetectable. Unfortunately, in the context of concurrency, the weak memory models can change the behavior of a program and thereby cause particularly wicked bugs.

On the other hand, sequentially consistent interleaving models make us do more work than required. The key is that we are not considering all programs, but only programs that are data-race free, *i.e.*, well-synchronized and verifiable in CSL. For well-synchronized programs interleaving only at concurrent instructions is equivalent to full interleaving; this was the key idea behind Dijkstra's invention of semaphores. In fact, we interleave the minimum number of times that preserves the semantics of a full-interleaving semantics. For example, in case er-step-prelock, we context switch even if the lock was already unlocked to let the other threads have a chance to grab it.

In future work we plan to extend our proofs to cover weak memory models. However, it is a mistake to prove that code at the C minor level obeys weak memory models. Instead, we will preserve our "rarely interleaving" semantics all the way through the compiler until we reach

machine code; only then will we prove the correctness of our semantics for well-synchronized programs executing on a machine with a weak memory model.

5.4 Conclusion

In chapter 3 we introduced Concurrent C minor. Here we gave Concurrent C minor a formal erased concurrent operational semantics. Our erased semantics is a reasonable model for conventional concurrency.

It is not easy to prove properties about our erased semantics. It is easier to reason about the unerased semantics defined in chapter 8. In section 8.6 we prove an erasure theorem that says that the unerased semantics is a conservative approximation to the erased semantics given here. In chapter 9 we define an oracular semantics that is suitable for reasoning about one thread at a time. In chapter 10 we model our CSL on that oracular semantics, and prove that properties proved with the oracular semantics hold on the unerased concurrent operational semantics. The erasure theorem then guarantees that the properties hold on our erased semantics.

Chapter 6

Engineering Isolation

In chapter 3 we presented the Concurrent C minor language and gave an example program in 3.2. Then in chapter 4 we developed a new concurrent separation logic and demonstrated its power by using it to verify an example program.

In the remainder of this thesis we will present a soundness proof of our Concurrent Separation Logic with respect to the operational semantics of Concurrent C minor. Since the soundness proof is developed in Coq it has aspects of both a mathematical proof and also an engineered software artifact. These aspects have influenced each other: part of the difficulty in developing the mathematical ideas was that they had to integrate cleanly into existing Coq proofs; conversely, at times the engineering process became too difficult and new mathematical techniques had to be developed to simplify the engineering task.

In this chapter, we focus particularly on engineering design choices

that allow for greater modularity in the mechanized Coq proof[1]. The material in this chapter has not been fully integrated into our Coq development, which uses a somewhat less modular design; work is ongoing to bring the Coq development into closer agreement. In chapter 7, we will develop the modal substructural logic underlying CSL assertions. In chapter 8 we will explain the operational semantics of Concurrent C minor. In chapter 9 we will introduce the oracle semantics for Concurrent C minor. Finally, in chapter 10, we will construct a modular proof of the soundness of Concurrent Separation Logic using the oracle semantics.

Note to readers uninterested in engineering a large proof in Coq. Readers who are largely uninterested in or unfamiliar with Coq engineering efforts should read section 6.1, and then may skip this chapter until section 6.6, where we define the notation and briefly cover the ideas we will use in the rest of the thesis.

6.1 Modularization Goals

Modules are an engineering technique that isolate parts of a software system from each other. By dividing up the system into isolated components, it becomes easier to understand, maintain, upgrade, and re-use. In our work, we use modules to isolate the sequential and concurrent parts of the system from each other. In general, reasoning about sequential features or concurrent features is complicated enough; to reason

[1]This chapter has not been previously published.

about both simultaneously would be very difficult. Also, if our isolation is strong enough, it will be easier to modify the CompCert compiler to compile Concurrent C minor since the existing sequential proofs will not have to be changed very much.

6.1.1 Core, glue, and extension

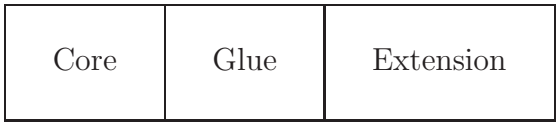

Figure 6.1: An extensible language

Languages like C minor have many complex features. At the level of syntax, adding new features is generally done by directly adding new statements into the set of statements already in the language. However, this is not very modular, as many proofs are done using case analysis, and all such proofs must be redone when new statements are added. Moreover, the semantics of the new language cannot be directly built from the semantics of the old language.

Instead of directly mixing features, we design a "glue" interface layer that allows a core language to be extended with new features. The glue combines the syntax and semantics of the core and extension to build a new language. The relationship between the core, glue, and extension is illustrated in figure 6.1. Since the glue isolates the core and extension from each other, the glue allows us to re-use proofs about the component parts as lemmas in proofs about the combined language.

Language level	Language	Glue	Extension
Source	C minor	$\text{Glue}_{\text{C minor}}$	$\text{Extension}_{\text{C minor}}$
Intermediate 1	RTL	Glue_{RTL}	$\text{Extension}_{\text{RTL}}$
Intermediate 2	LTL	Glue_{LTL}	$\text{Extension}_{\text{LTL}}$
Intermediate 3	Mach	$\text{Glue}_{\text{Mach}}$	$\text{Extension}_{\text{Mach}}$
Target	PowerPC	$\text{Glue}_{\text{PowerPC}}$	$\text{Extension}_{\text{PowerPC}}$

Table 6.1: Naïve application of glue to compiler

We will take as the core the C minor of Appel and Blazy [AB07] and as the extension a semantics of the concurrent statements make_lock, free_lock, lock, unlock, and fork. We glue these together to make Concurrent C minor.

6.1.2 Level independence

A major goal of our Concurrent C minor project is to extend the correctness proofs of the CompCert compiler [Ler06] from the sequential case to the conccurent one. The CompCert compiler translates C minor through several intermediate languages to PowerPC. A naïve way to apply the glue and extension technique to such a system is illus-

trated in table 6.1. From the source language of C minor to the target language of PowerPC, each sequential language is attached to some semantics of concurrency specialized to that language with its own language-dependent glue.

While this method would work, it would require much more substantial modifications to the CompCert compiler and correctness proofs than ideal, since the reasoning about the statements would have to be done differently for each intermediate language. This would also mean that a change to the new statements would require changing all of the proofs about the intermediate languages. One question is how well the glue isolates the core and extension from each other, and also how much the glue depends on the two of them.

Ideally, the core and extension could be so isolated from each other that each language could use the same extension to add new behavior, as illustrated in table 6.2. To do this, the new statements must fit uniformly into all of the intermediate languages of the CompCert compiler. While this does make the initial engineering work harder, the benefit should be a substantially easier time modifying the compiler to handle new extensions[2].

[2]Since the compiler has not yet been modified, it is quite likely that some additional engineering work beyond what is presented in this chapter will be necessary for that process. In particular, there are concerns that the "oracle" used to build the concurrency extension is not fully level independent. Hopefully, however, this material will provide a good start in the proper direction.

Language level	Language	Glue	Extension
Source	C minor	$\text{Glue}_\forall$	$\text{Extension}_\forall$
Intermediate 1	RTL	$\text{Glue}_\forall$	$\text{Extension}_\forall$
Intermediate 2	LTL	$\text{Glue}_\forall$	$\text{Extension}_\forall$
Intermediate 3	Mach	$\text{Glue}_\forall$	$\text{Extension}_\forall$
Target	PowerPC	$\text{Glue}_\forall$	$\text{Extension}_\forall$

Table 6.2: Level independence

6.2 Shared definitions for glue

We now present an extension system which takes a "core" small-step semantics and extends with certain additional instructions. We will then use this extensible semantics to build Concurrent C minor on top of the C minor of Appel and Blazy[AB07], and then later use it again build an oracular semantics for Concurrent C minor.

6.2.1 Basic definitions

All of the languages of the CompCert compiler share certain common definitions, including those given in figure 6.2[3].

[3]Coq code shown in this chapter has been simplified for the presentation.

```
1   (* Definitions shared in all CompCert languages *)
2
3   (* Memory addresses *)
4   Definition addr      := Z * Z
5
6   (* Identifiers *)
7   Definition ident     := nat.
8
9   (* Values *)
10  Inductive val        :=
11    | Vundef: val
12    | Vint: int -> val
13    | Vfloat: float -> val
14    | Vptr: addr -> val.
15
16  Definition globals   := ident -> option addr.
17
18  Definition locals    := ident ->  option val.
19
20  (* Lines 21 - 53 are explained in section 5.2.2 *)
21  (* Permissions *)
22  Parameter resource   : Type.
23
24  (* A map from address to permissions *)
25  Parameter rmap       : Type.
26-53 (see figure 5.5)
54
55  Definition mem       := address -> val.
56
57  Definition world     :=
58    globals * locals * rmap * mem.
59  Definition predicate := world -> Prop.
60
61  Definition ext_fun   := ident.
```

Figure 6.2: Shared definitions in all CompCert languages

Addresses are pairs of integers, which give a block and offset; this allows for greater modularity in the compiler. **Identifiers** (names of variables) are natural numbers. There are four kinds of **values**: int, float, pointer, and undefined[4]. A mapping from identifiers to addresses allows global variables to be used.

Locals. Local variables are tricky to handle uniformly, since each CompCert language has its own representation. To allow reasoning to operate more smoothly with the languages at each level in the compiler, we define a level-independent definition of locals as a map from $\mathbb{N}$ to value. Then, each language's glue will be responsible for converting its own private representation of local variables into the shared one when required.

Resource map. A *resource map* is a generalization of the footprint defined by Appel and Blazy [AB07]. Recall from section 2.5.2 that a footprint is a map from addresses to permissions. For Appel and Blazy, a permission was simply a binary value, where **true** meant that memory access to that address was allowed and **false** meant that memory access was forbidden. In our setting, resource maps and permissions are more complex and are explained in section 6.2.2. Like the rest of the shared definitions, the model of resource maps does not depend on any particular program syntax, meaning that every language in CompCert can use the same definition.

Memory. One of the major triumphs of the CompCert system is

[4]For example, storing a word and then reading a byte from the same address results in an undefined value, per the ANSI C standard.

that the same memory model is used for the source, target, and all intermediate languages. For this presentation we simplify the CompCert memory model so that memory is a map from addresses to values; Leroy et al. [LB08, Ler06] explain the CompCert memory model in more detail.

World, Predicate. We call the tuple of globals, locals, resource map, and memory a *world*. A *predicate* is a function from a world to a proposition in the calculus of co-inductive constructions (Coq's `Prop`). All of the assertions of separation logic developed in chapter 4 are predicates. A major advantage of defining things in this way is that since all of the components in a world are level-independent, so are predicates. Therefore, all of our separation logic assertions are well-defined and indeed have the same definition for all of the languages in CompCert.

External functions. Finally, one of the few statement types that is shared by all of the CompCert languages is the idea of an external function call, such as a library call, trap to the operating system, or other similar situation. Each external function is given a unique identifier, and a global table maps identifiers to the proper calling convention at each level in the compiler. There is therefore a uniform way to represent external function calls for all CompCert languages, and we will use this infrastructure to build our extension system.

6.2.2 Resource maps

Informally, a resource map, like the footprint of Appel and Blazy, is a map from an address to a permission, which we call a *resource*. As in Appel and Blazy's setting, using memory without the correct permission causes the machine to get stuck.

The formal definition of resource maps is technical and will be covered in chapter 7. In figure 6.3 we present an "axiomatic" view of the core definitions.

As explained above, a resource is a kind of permission, and a resource map is a function from addresses to resources. Given a resource map `rm` one uses `resource_at` to determine the resource at a given address `addr`. Since this is the primary use for a resource map, we define the notation `@`; this means:

```
rm @ addr  =  resource_at rm addr
```

A nonempty fractional share, as defined in section 4.5, has type `pshare`.

Resources have a kind, which is an inductive type that has five constructors: `kVAL` (regular data), `kFUN` (functions), `kLK` (locks), `kRES`, and `kCT`. Both `kRES`, and `kCT` are related to the C minor memory model and are explained in section 6.3.2.

We provide two pseudoconstructors for building resources: `NO` and `YES`. Both take a resource map as the first parameter; the reason for this parameter will be explained in chapter 7. For the purposes of this

```
21  (* Permissions *)
22  Parameter resource   : Type.
23
24  (* A map from address to permissions *)
25  Parameter rmap       : Type.
26
27  (* How to get from an rmap to a resource *)
28  Parameter resource_at: rmap -> address -> resource.
29  Infix "@" := resource_at (at level 50,no associativity).
30
31  (* Fractional ownership - nonempty share *)
32  Parameter pshare     : Type.
33
34  (* Resource kinds *)
35  Inductive kind       :=
36    kVAL | kFUN | kLK | kRES | kCT.
37
38  (* Resource abstract pseudoconstructors *)
39  Parameter NO         : rmap -> resource.
40  Parameter YES        : rmap -> kind -> pshare ->
41                           (list predicate) -> resource.
42
43  (* Resource pseudoconstructors *)
44  Definition VAL (rm: rmap) (sh: pshare) :=
45    YES rm sh kVAL nil.
46  Definition FUN (rm: rmap)(sh: pshare)(P Q:predicate) :=
47    YES rm fullshare kFUN (P :: Q :: nil).
48  Definition LK (rm: rmap) (sh: pshare) (R: predicate) :=
49    YES rm sh kLK (R :: nil).
50  Definition RESERVED (rm: rmap) :=
51    YES rm fullshare kRES nil.
52  Definition CT (rm: rmap) (sh: pshare) :=
53    YES rm sh kCT nil.
```

Figure 6.3: Axiomatic presentation of resources and resource maps

chapter, this parameter is not important.

`NO rm` indicates that all forms of access to that memory location are forbidden[5]. `YES rm k sh LP` indicates that the memory location permits certain kinds of access, the exact nature of which is determined by the kind `k`, share `sh` and list of predicates `LP`.

Using `YES` we define five different resources: `VAL`, `FUN`, `LK`, `RESERVED`, and `CT`. `VAL rm sh` indicates that the location is owned with (positive) fractional ownership `sh`. Partial ownership will give permission to read from memory; full ownership will give permission to both read from and write, and so we define `readable` and `writable` as:

```
Definition readable res := exists phi, exists sh,
 res = VAL phi sh.
Definition writable res := exists phi,
 res = VAL phi fullshare.
```

`FUN rm P Q` indicates that the address contains a function with precondition `P` and postcondition `Q`; in other words, if `P` is satisfied just before jumping to the address, then `Q` will be satisfied after the function returns. `FUN` uses `fullshare` for its share; we will use a special resource map called the function pool to make sure that all threads have access to the functions[6].

`LK rm sh R` indicates that the address is partially owned with share

[5]Another way to think about resource maps is that they are partial functions; in this case `NO rm` indicates that the location is not in the domain.

[6]Recall that in section 4.7.3 the CSL function assertion does not have a share.

`sh`, and contains a lock with associated invariant `R`; in other words, locking the lock will result in gaining the invariant `R`. `RESERVED rm` and `RESERVED rm sh` are related to the C minor memory model and are explained in section 6.3.2.

There are three important points to note. First, these "constructors" are *semantic*, meaning that new kinds of resource can be defined at any time; accordingly, if one has a resource, it does not follow that it is one of these five. In fact, the underlying model is strong enough to allow for an open-ended number of resource kinds, although in the Coq development we only have the five explained above.

Second, these pseudoconstructors are not fully invertible; *i.e.*, from:

```
YES rm k sh LP   =   YES rm' k' sh' LP'
```

we can conclude that `k` = `k'` and that `sh` = `sh'`; however it is **invalid** to conclude that `rm` = `rm'` or that `LP` = `LP'`; in chapter 7 we will explain exactly what can be concluded about these parameters.

Third, eagle-eyed readers may have noticed that the definition of `YES` on lines 40–41 takes a list of `predicate`s as parameters, but `predicate` is not defined until line 59 (in figure 6.2). In general, this is a problem.

The definition of predicate given on line 59 is correct. In chapter 7 we will show that in fact `resource`, `rmap`, and `predicate` are all defined simultainously in a noncircular way. For the purposes of this chapter, we will ignore this issue and therefore remain slightly informal.

6.3 An Extensible Semantics

Consider a small-step operational semantics of the form

$$\Psi \vdash \sigma_1 \longmapsto \sigma_2,$$

where Ψ is a global environment (mostly the syntax of the running program) and σ is a state. We require that σ be isomorphic to a tuple of (ϕ, m, ς), where ϕ is a resource map, m is the memory, and ς is other kinds of state, such as in C minor the stack pointer, local variables, and control (code, *i.e.*, program syntax). We distinguish the memory and resource map because the types of these are the same in every language in the CompCert stack. Other components of the state are unique to each language and are bundled into the core state ς.

We show the Coq interface axiomatizing these ideas in figure 6.4[7]. Of course, each language has its own type of syntax, which in the module is called the *program*. Each language also has its own state, denoted σ, which is isomorphic to a triple of `rmap` ϕ, `mem` m, and `core` ς. Requiring an isomorphism instead of requiring state to be a triple gives the languages additional flexibility in their definitions. We require that the core contain some represention of local variables.

Extensibility is given by the parameter `at_external`. While the program is executing normal instructions, `at_external` returns `None`. When control reaches an external function call it transforms its private

[7]Proven in Coq: C minor satisfies the interface.

```
62  Module Type EXTENSIBLE_SEMANTICS.
63
64  (* Syntax *)
65  Parameter program    : Type.
66
67  (* Level-dependent representation *)
68  Parameter state      : Type.
69
70  (* Level-dependent data *)
71  Parameter core       : Type.
72
73  Parameter from_state : state -> rmap * mem * core.
74  Parameter to_state   : rmap * mem * core -> state.
75  Parameter core_locals: core -> locals.
76
77  Parameter at_external: core ->
78    option (ext_fun * locals * predicate).
79  Parameter after_external: locals-> core-> option core.
80
81  Definition genv: Type := (program * globals).
82
83  Definition filter (ge: genv) (st: state) : world :=
84    match (from_state st) with (rm, m, c) =>
85      (snd ge, core_locals c, rm, m)
86    end.
87
88  (* call main *)
89  Parameter initial_core: val -> list val -> core.
90
91  Parameter step: genv -> state -> state -> Prop.
92
93  (* Axioms about the step relation *)
94-152 (* See figures 5.7, 5.8, and 5.9 *)
153
154 End EXTENSIBLE_SEMANTICS.
```

Figure 6.4: Interface for extensible semantics

representation of that call into a language-independent representation.

In a language-independent representation, an external function call has three components. First, an identifier that specifies which function call, for example a 0 for lock or a 1 for unlock. Second, a set of locals, which become the parameters to the external function. Finally, external function calls are also able to take a predicate; this is required so that we can handle make_lock, which takes a resource invariant (predicate) as a parameter; it would also be useful if one wished to define an `assert` statement that took a real separation logic assertion instead of an expression.

Once the external function call is in the language-independent representation, an extension is able to provide the proper semantics. The extension is allowed to modify both the resource map and the memory as it executes, but is not allowed to modify the language-dependent core state. When the extension wishes to return control to the core, it places the return value of the function into a `locals` type and uses the `after_external` parameter to resume the core semantics.

Global data `genv`, denoted Ψ, is a tuple of two components: a language-depenent program, which contains the syntax of the code, and a language-independent global map, which maps global names to addresses.

Since each language has a different representation of state, the definition `filter` is provided so that the same `predicate` can be applied regardless of the details of the current language in the compilation pro-

cess. To apply the predicate `Q` in the context of language `L1` to the state σ_1 of type `L1.state`, one writes in Coq:

$$\texttt{Q (L1.filter } \Psi_{\texttt{L1}} \ \sigma_1\texttt{)}$$

Then if one later wants to apply the same predicate in the context of a different language `L2`, to the state σ_2 of type `L2.state`, one writes in Coq:

$$\texttt{Q (L2.filter } \Psi_{\texttt{L2}} \ \sigma_2\texttt{)}$$

The ability to use the same predicate `Q` with different languages should significantly simplify modifications to the compiler, since it means that the compiler will not have to manipulate predicates as it compiles code from one language to the next.

Although the internals of `core` vary significantly between the languages, all of the languages support the idea of creating an initial, simple `core`, typically for program start with `call main()`. The parameter `initial_core` is provided for that purpose.

Most importantly, a language has a `step` relation (*i.e.*, $\longmapsto$), which gives its actual small-step semantics.

For a given state, there does not always exist a subsequent state that the given state steps to. Typically this means that the semantics has gotten stuck, but in this case there is another possibility: the control in question is not part of the core language, and should be handled by the extension system. A simple example of a non-core instruction would be

a trap to the operating system; in Concurrent C minor the extension instructions are the concurrent instructions make_lock, free_lock, lock, unlock, and fork.

6.3.1 Axioms and basic definitions

The `EXTENSIBLE_SEMANTICS` module type also contains certain axioms and basic definitions about the step relation that must be satisfied; these are largely given in figure 6.5. Axioms are part of the module type to isolate and centralize the dependencies between the core and extension[8].

The axiom `step_obeys_rmap` on lines 96–102 expresses the major reason to include resource maps (or footprints) in the semantics. The key is on lines 100–101: for each address, either the address is writable (*i.e.*, `VAL _ fullshare`) or the value in memory before and after steping are **equal**[9].

The axiom `step_preserves_resource` on lines 104–121 details how the step relation is allowed to change the resource map. In fact, very little is allowed to change. If the value was empty before stepping, then it must be empty afterwards (line 117). If not, then there are two possibilities; by far the most common is that if the value was a `YES` value, then it is preserved with the same kind, share, and predicate list

[8]While it has been shown in Coq that C minor satisfies all of the axioms presented; it has not yet been shown that these are all of the axioms required by the concurrency proofs, since the process of rebuilding the proof using this cleaner design is ongoing.

[9]In the existance of byte-addressed memory, this property is more complicated.

```
93  (* Axioms about the step relation *)
94
95  (* Memory does not change except for fully-owned data *)
96  Axiom step_obeys_rmap: forall ge st st',
97    step ge st st' ->
98    match (from_state st, from_state st') with
99      ((rmap, m, _),(_, m', _)) =>
100       forall addr, (writable (rmap @ addr)) \/
101                    (m addr = m' addr)
102   end.
103
104 (* Required for CompCert memory model *)
105 Definition rmap_memory_model_rmap_ok (phi phi': rmap) :=
106-111 (* See figure 5.8 *)
112
113 Axiom step_preserves_resource: forall ge st st',
114   step ge st st' ->
115   match (from_state st, from_state st') with
116    ((rmap, _, _),(rmap', _, _)) => forall addr,
117      (rmap @ addr = NO rmap -> rmap' @ addr = NO rmap') /\
118     ((forall sh k LP, rmap  @ addr = YES rmap  k sh LP ->
119                       rmap' @ addr = YES rmap' k sh LP) \/
120      (rmap_memory_model_rmap_ok rmap rmap'))
121   end.
122
123 (* Determinism *)
124 Axiom step_fun: forall ge st st1 st2,
125   step ge st st1 -> step ge st st2 -> st1 = st2.
126
127 (* Required for model of resource maps *)
128 Axiom step_not_any_younger:
129   (* See section 6.XX *)
130
131 Axiom step_allocpool: forall ge st st',
132-134 (* See figure 5.8 *)
```

Figure 6.5: Axioms for Extensible Semantics

(lines 118–119); the other possibility (lines 104–111 and 120) is related to the C minor memory model and is covered in section 6.3.2.

The axiom `step_fun` on lines 124–125 says that the step relation must be deterministic. This property is heavily used in both the CompCert compiler proofs and the soundness proofs of sequential separation logic developed by Appel and Blazy.

The axiom `step_not_any_younger` is a technical axiom required by the model of resources, resource maps, and predicates, and will be explained in chapter 7.

6.3.2 Definitions and axioms for the C minor memory model

Here we give a *brief* explanation of some definitions and axioms required by the CompCert memory model; Leroy et al. [LB08, Ler06] give a fuller explanation of the memory model and the reasons behind its design decisions.

The resource `RESERVED` is used to indicate memory locations that have not been allocated, but can be allocated in the future. The resource `CT` is used because C minor is byte-addressed; accordingly, since locks are four bytes long, the first byte has resource `LK` and the next three have resource `CT`.

There are two major additions to the step relation axioms due to the memory model; these are given in figure 6.6.

```
104 (* Required for CompCert memory model *)
105 Definition rmap_memory_model_rmap_ok (phi phi':rmap) :=
106  forall addr,
107   (phi @ addr = RESERVED phi pfullshare /\
108      ((phi' @ addr = VAL phi' pfullshare) \/
109       (phi' @ addr = NO phi'))) \/
110   (phi @ addr = VAL phi pfullshare /\
111        phi' @ addr = NO phi').

...

131 Axiom step_allocpool: forall ge st st',
132   step ge st st' ->
133   ((allocpool * TT) (filter ge st) ->
134    (allocpool * TT) (filter ge st')).
```

Figure 6.6: Axioms for C minor memory model

The definition `rmap_memory_model_rmap_ok` on lines 104–111 gives two additional ways the resource map is able to change. In the first, on lines 107–109, resource is allowed to change from `RESERVED` to `VAL` or `NO`; this happens when memory is allocated. In the second, on lines 110–111, resource is allowed to change from `VAL` to `NO`; this happens when memory is deallocated.

The axiom `step_allocpool` on lines 131–134 is an elegant way of expressing that one never "runs out" of allocatable memory. Instead of expressing this fact directly, the axiom is defined in terms of a predicate `allocpool`, which must be a satisfied by some portion of the resource map (`TT` is the predicate **true**). The definition of `allocpool` is both technical and not relevant to concurrency, and so we do not present it here.

```
135 Inductive stepstar (ge:genv): state-> state-> Prop :=
136  | stepstar_0: forall st,
137    stepstar ge st st
138  | stepstar_S: forall st1 st2 st3,
139    step ge st1 st2 ->
140    stepstar ge st2 st3 ->
141    stepstar ge st1 st3.
142
143 Inductive immed_safe (ge : genv) (st: state) : Prop :=
144 |  immed_safe_step:  forall st',
145    step ge st st' ->
146    immed_safe ge st
147 |  immed_safe_rmap:
148    ... level of resource map is 0; see chapter 6 ... ->
149    immed_safe ge st.
150
151 Definition safe (ge: genv) (st: state) : Prop :=
152   forall st', stepstar ge st st' -> immed_safe ge st'.
```

Figure 6.7: Basic definitions for Extensible Semantics

6.3.3 Common definitions for core semantics

Finally, there are some common definitions for all core semantics which are given in figure 6.7. `stepstar` is defined in the standard inductive way to allow for finite composed applications of `step`. On lines 144–146, `immed_safe` says that a state is immediately safe if it can take a step; on lines 147–149 there is an additional way to be immediately safe, which is if the "level" of the resource map in the state is 0; this is required for the model of resource maps and will be explained in chapter 7.

Lastly, `safe` is defined in the standard way: a state σ is safe if, for any reachable state σ', σ' is immediately safe.

6.4 Building an extension

Once we have a core language, such as the C minor of Appel and Blazy, we want to extend it, for example with concurrency primatives. By using a "bolt-on" model for extensions, we can ensure that changes to the core semantics do not affect the extensions and vise versa.

6.4.1 Examples of extensions

Here we discuss different examples of extensions definable in our setting.

The null extension

One very simple extension is the *null extension*, which is the extension that does not implement any functionality. In this case the combination of the core language and the extension is equivalent to just the core language.

The operating system extension

A more complex extension is the *simple operating system extension*. This extension implements three external functions: `read`, which gets input from the user; `write`, which sends output to the user; and `exit`, which terminates the program and returns control to the operating system.

The sequential sub-machine extension

To define the semantics of Concurrent C minor we will build two different concurrency extensions. In chapter 8, we will build a small extension called the *sequential sub-machine extension* that only knows how to execute the concurrent instructions make_lock and free_lock, getting stuck on lock, unlock, and fork. We will use this extension to build the concurrent operational semantics of Concurrent C minor.

The oracular concurrency extension

Later, in chapter 9, we will build a more powerful extension using the concurrent operational semantics, called the *oracular concurrency extension*. That extension will be able to execute all five of the concurrent instructions, and we will use it to prove our CSL sound.

6.4.2 The `EXTENSION` module type

An extension must satisfy the module type given in figure 6.8. Unlike the core, extensions are quite simple. There is a type `oracle` (in mathematical notation Ω), which acts as "private data" for the extension and is hidden from the core semantics. The oracle type must be inhabited, so we provide a witness, `an_oracle`.

The point of an extension is to handle external function calls. However, any given extension need not handle all external functions; it is useful to be able to query the extension to determine which external functions it handles. The parameter `handles` gives an easy test for this.

```
155 Module Type EXTENSION.
156 Parameter oracle: Type.
157 Parameter an_oracle: oracle.
158
159 Parameter handles: external_function -> bool.
160
161 Parameter consult: external_function ->
162                    (oracle * world * predicate) ->
163                    option (oracle * world) -> Prop.
164
165 Axiom consult_obeys_rmap: forall c o1 w1 P o2 w2,
166   consult c (o1, w1, P) (o2, w2) ->
167   match (w1,w2) with ((_,_,rmap,m),(_,_,rmap',m')) =>
168     forall addr, (readable (rmap  @ addr) /\
169                   readable (rmap' @ addr)) ->
170                     (m addr = m' addr))
171   end.
172
173 (* Determinism *)
174 Axiom consult_fun: forall c a b1 b2,
175   consult c a b1 -> consult c a b2 -> b1=b2.
176
177 (* Required for model of resource maps *)
178 Axiom consult_not_any_younger:
179   (* See section 6.XX *)
180
181 (* Required for CompCert memory model *)
182 Axiom consult_allocpool: forall c ora w P ora' w',
183   consult c (ora, w, P) (Some (ora', w')) ->
184   ((allocpool * TT) w ->
185    (allocpool * TT) w').
186
187 End EXTENSION.
```

Figure 6.8: Interface for extension

To actually handle the external function call, the extension provides a `consult` relation, which takes an external function (*i.e.* a natural number that encodes which function is being called), a tuple (Ω, w, P), which are the arguments to the external call, and an option of tuple (Ω, w), which are the results.

In general, `consult` only modifies `world`, since world is exactly the part of the state that is shared between all of the languages of the CompCert compiler. However, the extension is allowed to use some auxiliary private state in the oracle. In the operating system extension this would encapsulate the state of the operating system; in the oracular concurrency extension it encapsulates the state of the other threads.

The predicate P in the `consult` function is used to implement make_lock. It would also be useful to implement an `assert` function that took a predicate instead of an expression.

Like `step`, the `consult` relation is partial and can get stuck. When `consult` returns `None`, the result is not that the consult failed, but that control never returns. Two examples of this result would be modeling the `exit` system call in the operating system extension and modeling the lock instruction, where *e.g.* the call can deadlock, in the oracular concurrency extension. When `consult` returns a new oracle and world, the world is passed back to the core semantics and the oracle is provided as a parameter on the next `consult`, thereby allowing the extension to use the oracle as shared state. In general the oracle is able to return a completely different world (memory, resource map, etc.) than the one

it was given.

In addition, like the core semantics, an extension must satisfy a variety of axioms[10]. The key axiom is `consult_obeys_rmap`, which is given on lines 165–171. This axiom is somewhat different from the one defined for the core on lines 96–102 in figure 6.5. Instead, this one guarantees that as long as an address is readable both before and after consulting then the memory at that address is unchanged by the consult.

All of the other axioms are very similar to the ones given for the core in figures 6.5 and 6.6.

6.5 Gluing a core and extension together

Once one has built a core semantics and an extension, one would like to glue them together to have a complete semantics. We define a functor `Oraclize` in Coq that takes two modules as arguments, the first of module type `EXTENSIBLE_SEMANTICS`, and the second of module type `EXTENSION`. This functor then builds a step relation of the following form:

$$\Psi \vdash (\Omega, \sigma) \nmapsto (\Omega', \sigma')$$

The Coq implementation of the *oracular step relation* is in figure 6.9.

The oracular step has three cases. Case one is given on lines 4–8. First, in line 5 we ensure that the state is not at an external function

[10]Just as with the core axioms, it has been shown in Coq that the concurrency extension meets these axioms, but it is unclear if other axioms are required as the proof is being re-engineered into this format.

```
Module Oraclize (Core: EXTENSIBLE_SEMANTICS) (Ext: EXTENSION).
...
1  Inductive oracle_step
2    (ge: genv) (ora1 : oracle) (st1 : state)
3                (ora2: oracle) (st2: state) : Prop :=
4    | step_core:
5        at_external (snd (snd (from_state st1))) = None ->
6        ora1 = ora2 ->
7        step ge st1 st2 ->
8        oracle_step ge ora1 st1 ora2 st2
9    | step_external_Some:
10     forall phi m core f vargs P rho' rho' core',
11       st1 = to_state (phi, m, core) ->
12       at_external core = Some (f,vargs,P) ->
13       consult f (ora1,(snd ge,vargs,phi,m), P)
14                 (Some (ora2,(_,rho',phi',m'))) ->
15       after_external rho' core = Some core' ->
16       st2 = to_state (phi',m',core') ->
17       oracle_step ge ora1 st1 ora2 st2
18   | step_external_None:
19     forall phi m core f vargs P,
20       st1 = to_state (phi, m, core) ->
21       at_external core = Some (f,vargs,P) ->
22       consult f (ora1,(snd ge,vargs,phi,m),P) None ->
23       ora1 = ora2 ->
24       st1 = st2 ->
25       oracle_step ge ora1 st1 ora2 st2.
...
End Oraclize.
```

Figure 6.9: Coq oracular step relation

call; in this case line 6 ensures that the oracle is constant since the core semantics is not allowed to modify the it. On line 7, the core semantics steps, and on line 8 the parts are put back together.

Both cases two and three deal with external function calls. Case two is given on lines 9–17, and handles the case when the external function returns. We verify that the state is at an external function call on line 12. On lines 13–14, we consult the oracle using the extension. After consulting we reconstruct the state on lines 15–16, and on line 17 the parts are put back together.

Case three is given on lines 18–25 and handles the case when the external function never returns control. We verify that the state is at an external function call on line 21. On line 22, we consult the oracle using the extension, and receive `None` back as the return, indicating that the extension never releases control back to the core. In this case, the semantics loops around forever by setting the next state and oracle to be equal to the previous one on lines 23–24, and then putting the parts back together on line 25. Since the third case results in an infinite loop, it is trivially `safe`.

Along with this definition, the `Oracalize` functor has a number of lemmas about the relation, which are proven from the axioms on the core and extension.

6.6 Notation for the rest of the thesis

We have now given an explanation of how to modularize an extensible semantics in Coq. To simplify the presentation in the rest of the thesis, we will do the following. First, we will assume that the core semantics is "hardwired" to the C minor of Appel and Blazy. The core semantics will have a step relation given by

$$\Psi \vdash \sigma_1 \longmapsto \sigma_2,$$

Where a state σ is a pair of world w (that is, those parts of state that do not include program syntax: locals ρ, resource map ϕ, and memory m) and control κ (the parts of the state that do include program syntax).

Appel and Blazy define several different controls, but for this presentation we only $s \cdot \kappa$ and Kstop, as defined in chapter 5.

Second, we will pretend that the new concurrent statements $\mathsf{make\_lock}$, $\mathsf{free\_lock}$, lock, unlock, and fork are just normal statements added to sequential C minor, instead of extended function calls. These statements take values instead of expressions as parameters for simplicity.

Third, we will hide most of the modularization between core and extension. To build an extension, one constructs a partial function $\mathsf{consult}$:

$$\mathsf{consult} \ : \ \mathsf{program} \times \mathsf{oracle} \times \mathsf{state} \ \rightarrow \ \mathsf{option}(\mathsf{oracle} \times \mathsf{state}),$$

where a program Ψ, an oracle Ω and a state σ are the input to the

$$\text{Core Step}\ \frac{\Psi \vdash \sigma_1 \ \longmapsto\ \sigma_2}{\Psi \vdash (\Omega, \sigma_1)\ \mapsto\!\!\!\!\longrightarrow (\Omega, \sigma_2)}$$

$$\text{Oracle Step}\ \frac{\mathsf{consult}\ (\Psi, \Omega_1, \sigma_1)\ =\ \mathsf{Some}\ (\Omega_2, \sigma_2)}{\Psi \vdash (\Omega_1, \sigma_1)\ \mapsto\!\!\!\!\longrightarrow (\Omega_2, \sigma_2)}$$

$$\text{Oracle Diverges}\ \frac{\mathsf{consult}\ (\Psi, \Omega, \sigma)\ =\ \mathsf{None}}{\Psi \vdash (\Omega, \sigma)\ \mapsto\!\!\!\!\longrightarrow (\Omega, \sigma)}$$

Note: at most one case applies at a time.

Figure 6.10: Oracular step relation

consultation, and an oracle Ω' and state σ' are the optional output.

Using this partial function, one defines the oracular step with three cases as given in figure 6.10. The first case covers the situation when the core semantics is stepping. The second case covers when the oracle is taking control for an extended statement, but eventually returning control to the core. The third case is for when the extension does not return control to the core semantics, and therefore the program enters an infinite loop. For our work, the third case is only used when a deadlock occurs during the execution of a lock statement.

Chapter 7

A Modal Substructural Logic

In chapter 4 we defined Concurrent Separation Logic, and in sections 6.2.1–6.2.2 we axiomatized `resource` and `rmap` and explained that the assertions of CSL were modeled as `predicate`s.

In section 7.1 we explain the difficulties in providing a model for CSL assertions. In section 7.2 we explain the stratification technique that allows us to provide a sound definition for resources and resource maps. Finally, in section 7.3 we explain how to use a modal logic to reason cleanly about the underlying model and define the assertions of CSL[1].

[1]Portions of this chapter have been published before as [HAZ08a], [HAZ08b], [DAH08], and [BDH+08].

7.1 Modelling difficulties

In section 6.2.1 we explained that a `predicate` is a function from a tuple of globals, locals, resource map, and memory to a proposition in the Calculus of Co-Inductive Constructions (Coq's `Prop`). Though the ability to judge globals is important for developing proofs in CSL, modeling assertions about them is not difficult, so for the rest of this chapter, we elide globals to simplify the presentation; we also elide issues related to the C minor memory model outlined in section 6.3.2. In section 6.2.1, it was also explained that neither resource maps nor memories depend on the syntax of programs and so can be used with any CompCert language, which further justifies using them whenever possible to express invariants in the soundness proof.

7.1.1 Semantic *vs.* Syntactic

There are two basic approaches to programming language research. The first, more common today, is *syntactic*, where objects are uninterpreted symbols and are defined by judgements showing how they are to be used. The second, which is the approach we have taken, is *semantic*, where objects are given formal definitions and judgements showing how they should be used are proved from those definitions as lemmas.

We initially chose to give a semantic definition for our separation logic—instead of defining it syntactically and proving soundness via metaproofs—for three reasons. First, semantic systems are more ex-

tensible than syntactic ones. It is easy to define a new assertion, and unless the underlying semantics of the language must be changed to accomodate that assertion, all of the previously proved facts still hold automatically. Therefore in section 6.2.1 we were able to give a subset of the assertions of our CSL, including "maps-to" $e \mapsto v$ and "is a lock with invariant R" $e \overset{\pi}{\rightsquigarrow} R$, without worrying that we were leaving anything out. If a user decides we have left out a useful assertion, he can define it himself.

Second, after examining the Princeton Foundational Proof-Carrying Code (FPCC) project [App01] we found evidence that semantic methods scaled better than syntactic ones in large, machine-checked, realistic systems [BDH+08].

Third, we had more experience with semantic methods in the context of a large project. Partially as a result of this experience, we find semantic proofs more natural than syntactic ones, and easier to construct.

Fourth, although we do not have any hard evidence for this opinion, we believe that semantic methods are less brittle than syntactic proofs when confronted with the kinds of engineering changes common in a large project.

7.1.2 A naïve model for assertions

The recursive nature of first-class locks makes them very difficult to model semantically since it is easy to slip into Cantor's paradox. We

$$
\begin{array}{rcl}
\mathsf{kind} & = & \mathsf{kVAL} \;+\; \mathsf{kLK} \;+\; \mathsf{kFUN} \\
\mathsf{resource} & \approx & \mathsf{NO} \;+ \\
 & & \mathsf{YES}(\mathsf{kind} \;\times\; \mathsf{pshare} \;\times\; \mathsf{list}(\mathsf{predicate})) \\
\mathsf{rmap} & \approx & \mathsf{address} \to \mathsf{resource} \\
\mathsf{world} & = & \mathsf{locals} \;\times\; \mathsf{rmap} \;\times\; \mathsf{mem} \\
\mathsf{predicate} & = & \mathsf{world} \to \mathtt{Prop}
\end{array}
$$

Figure 7.1: Naïve assertion model

want a model that is something like the one in figure 7.1, where + and $\times$ are the sum and product type constructors and we write $\approx$ to mean "we wish we could define things this way".

The idea is that a resource can either be NO, which indicates that the location cannot be used, or YES, which indicates that the resource can be used in some way. A YES resource takes a kind, which can be one of three choices: kVAL, which indicates that the location contains data; kLK, which indicates that the location is a lock; and kFUN, which indicates that the location contains a function. A YES resource also takes an pshare and list(predicate). As explained in section 6.2.2, a pshare is a nonempty share as defined in section 4.5. The "high-level" resource pseudoconstructors VAL, LK, and FUN are defined in terms of YES.

An rmap associates every location in memory with a resource. As explained in section 6.2.1, locals, rmap, and mem are bundled together into a world. A predicate then is a function from world to `Prop`.

To see where the difficulty lies, let us examine the pseudodefinition

of the YES constructor:

$$\begin{aligned}\mathsf{resource} \;\approx\; & \mathsf{NO} \;+ \\ & \mathsf{YES}(\mathsf{kind} \times \mathsf{pshare} \times \mathsf{list}(\mathsf{predicate})).\end{aligned}$$

First we will unfold the definition of predicate to get

$$\begin{aligned}\mathsf{resource} \;\approx\; & \mathsf{NO} \;+ \\ & \mathsf{YES}(\mathsf{kind} \times \mathsf{pshare} \times \mathsf{list}(\mathsf{world} \to \mathtt{Prop})),\end{aligned}$$

then the definition of world to get

$$\begin{aligned}\mathsf{resource} \;\approx\; & \mathsf{NO} + \\ & \mathsf{YES}(\mathsf{kind}\times \\ & \quad \mathsf{pshare}\times \\ & \quad \mathsf{list}((\mathsf{locals} \times \mathsf{rmap} \times \mathsf{mem}) \to \mathtt{Prop})),\end{aligned}$$

and then the pseudodefinition of rmap to get

$$\begin{aligned}\mathsf{resource} \;\approx\; & \mathsf{NO} + \\ & \mathsf{YES}(\mathsf{kind}\times \\ & \quad \mathsf{pshare}\times \\ & \quad \mathsf{list}((\mathsf{locals} \times (\mathsf{address} \to \mathsf{resource}) \times \mathsf{mem}) \to \mathtt{Prop})).\end{aligned}$$

This definition would be possible if and only if the following much simpler, related definition were possible

$$\mathsf{resource} \;\approx\; \mathsf{resource} \to \mathtt{Prop}.$$

In both cases, there is a contravariant occurrence of resource within its own definition. In other words, the cardinality of resource is strictly larger than the cardinality of resource, a contradiction even in untyped set theory. Thus these definitions are unsound.

7.1.3 From mutable references to lock invariants

To provide a sound definition we adapt techniques developed for modeling mutable references in the FPCC project [AAV03, Ahm04]. It was surprising to us that the semantics of mutable references were relevant to the semantics of first-class locks. However, reading from a mutable reference is similar to locking a lock, in the sense that both actions are a way of getting information from the outside world. Analogously, storing to a mutable reference is similar to unlocking a lock, in the sense that both actions are a way of notifying the outside world of a change of state.

The *indexed model* developed for modelling mutable references in FPCC was extremely difficult to explain and very complex to use. It was developed in higher-order logic (HOL) encoded in the Twelf theorem prover; limitations on the expressiveness of HOL required extensive, very heavyweight Gödelization techniques to define the model and prove it sound, resulting in more than 20,000 lines of proof for this part of the model alone. Also, the underlying model was exposed to the rest of the proof of the type system in various unpleasant ways.

Appel et al. substantially redesigned the indexed model [AMRV07],

to create a *modal model*. This model was defined and proved sound in Coq in approximately 1,000 lines. The enormous difference in size is due to a combination of several factors: first, substantial redesign of the definitions and better proof techniques; second, the natural difference in size between using proof scripts instead of explicitly writing out full proofs; and third, the use of dependent types to avoid the Gödelization techniques. Moreover, Appel et al. used a modal logic to hide the underlying model from the remainder of the proof.

7.2 A substructural modal model

The sketch of the modal substructural model given in figure 7.2 is divided into three parts. The first section shows how we provide a sound definition in the presence of contravariance in a way that preserves impredicativity. The second section shows how dependent types can hide the stratification in the underlying model. The third section gives some definitions that are useful when translating the dependent model into the underlying stratified model[2].

In the Coq development, some definitions are hidden from the rest of the proofs with a module type. We use $\multimap\!\bullet$ to mean that the definition is completely invisible to the proofs outside the construction of the model. We use $\multimap\!\circ$ to mean that the definition is opaque to the rest of

[2]For further details of the stratification and soundness of the construction we refer to Appel et al. [AMRV07] and Richards [Ric08]; both refer to the 1,000 line Coq development cited earlier. Appel [App08] has also illustrated some of the modifications required to support substructural reasoning.

Impredicative Stratified Model

$$
\begin{array}{rcl}
\mathsf{kind} & = & \mathsf{kVAL} \;+\; \mathsf{kLK} \;+\; \mathsf{kFUN} \\
\mathsf{predicate}_0 & \bullet\!\!- & \mathtt{unit} \\
\mathsf{resource}_n & \bullet\!\!- & \mathtt{NO}_n \;+ \\
 & & \mathsf{YES}_n(\mathsf{kind} \;\times\; \mathsf{pshare} \;\times\; \mathsf{list}(\mathsf{predicate}_n)) \\
\mathsf{rmap}_n & \bullet\!\!- & \mathsf{address} \rightarrow \mathsf{resource}_n \\
\mathsf{world}_n & \bullet\!\!- & \mathsf{locals} \;\times\; \mathsf{rmap}_n \;\times\; \mathsf{mem} \\
\mathsf{predicate}_{n+1} & \bullet\!\!- & \mathsf{predicate}_n \;\times\; (\mathsf{world}_n \;\rightarrow\; \mathtt{Prop})
\end{array}
$$

Dependent Model

$$
\begin{array}{rcl}
\mathsf{resource} & \circ\!\!- & \Sigma n : \mathbb{N}.\ \mathsf{resource}_n \\
\mathsf{rmap} & \circ\!\!- & \Sigma n : \mathbb{N}.\ \mathsf{rmap}_n \\
\mathsf{world} & = & \mathsf{locals} \times \mathsf{rmap} \times \mathsf{mem} \\
\mathsf{predicate} & = & \mathsf{world} \;\rightarrow\; \mathtt{Prop}
\end{array}
$$

Relating the Dependent and Stratified Models

$$
\begin{array}{rcl}
\mathsf{level} & : & \mathsf{rmap} \rightarrow \mathbb{N} \\
 & \bullet\!\!- & \lambda\{n, \phi_n\}.\ n \\
\mathsf{stratify} & : & \Pi n : \mathbb{N}.\ \mathsf{predicate} \;\rightarrow\; \mathsf{predicate}_n \\
 & \bullet\!\!- & \mathsf{fix}\ \mathsf{stratify}\ \lambda n.\ \lambda P.\ \mathsf{if}\ (n = 0)\ \mathsf{then}\ (\mathtt{tt} : \mathtt{unit}) \\
 & & \quad \mathsf{else}\ (\mathsf{stratify}\ (n-1)\ P, \\
 & & \qquad \lambda v : \mathsf{world}_{n-1}.\ P(v.1,\ \{n-1, v.2\},\ v.3))
\end{array}
$$

Figure 7.2: Sketch of substructural modal model

the proof; that is, the rest of the proof knows that the definition exists, but is not able to "look inside" and see the details of the definition. We use = to mean that the definition is completely public, and that the rest of the proof is able to use it directly.

There are a number of differences between the presentation here and the Coq development. The most important ones are these:

1. Here predicates judge only locals, resource maps, and memory; as explained in section 6.2.1, in the Coq development, predicates also judge globals.

2. Here we ignore issues related to byte addressability. In the Coq development, we must handle this issue. In particular, we need to ensure that the first byte of a lock cannot be separated from the following three bytes. We do this by maintaining validity predicates about valid resource maps.

3. As explained in section 4.7.3, in the presentation we do not support the relation of function preconditions to function postconditions; *i.e.*, we do not have a way of expressing that a call to an increment function returns a value exactly one larger than its parameter. In the actual Coq development, both our CSL and the underlying model support this additional expressiveness. In the model this is done by parameterizing predicate lists over an additional type and value parameter.

4. Here we use the $\rightarrow$ type constructor to construct the types of rmap_n and rmap. In the Coq development we use a quasifunctor called `Joinmap` that helps preserve certain separation properties.

5. Our presentation here is less modular than the Coq development.

6. It is nontrivial to produce definitions that will be acceptable to the Coq theorem prover from the definitions as given in the presentation, largely due to the heavy use of dependent types in the model. In appendix A we give a very compact Coq development that implements the core ideas presented in figure 7.2; this miniature development should help a reader who wishes to understand (or build) a larger Coq implementation. In the real Coq development, the entire modal substructural model has been completely defined and proved sound.

7.2.1 An impredicative stratified model

The idea is that we will stratify $\mathsf{predicate}$s, $\mathsf{resource}$s, and rmaps; we write $\mathsf{predicate}_n$ to mean a predicate at level n, $\mathsf{resource}_n$ to mean a resource at level n, and rmap_n to mean a resource map at level n.

To ensure a well-founded construction, we define $\mathsf{predicate}_0$ as `unit`. A resource at level n is only allowed to contain predicates of level n; similarly, a resource map at level n is a map from $\mathsf{address}$ to $\mathsf{resource}_n$. As discussed in section 6.2.1, we bundle locals, $\mathsf{resource}_n$, and memory together into a world_n. A predicate at level $n+1$ is a pair of $\mathsf{predicate}_n$

and a map from world_n to $\mathtt{Prop}$. Thus, a predicate at level $n+1$ contains a list of predicates for all levels between n and 0, in addition to the map from world_n to $\mathtt{Prop}$. As an example, consider $n = 3$:

$$\mathsf{predicate}_3 \;=\; (\mathtt{unit} \times (\mathsf{world}_1 \to \mathtt{Prop})) \times (\mathsf{world}_2 \to \mathtt{Prop}).$$

We define $\mathsf{predicate}_n$ as a pair because it lets us define "aging" the predicate in section 7.2.4 in a relatively simple way.

By stratifying in this manner we make the definitions well-defined. The stratified construction was one of the major technical accomplishments of the indexed model for mutable references that Ahmed et. al [AAV03, Ahm04] developed for the FPCC project.

One major advantage of this construction over previous models [AM01, AAV02] is that it supports *impredicative quantification*. That is, since the result type of $\mathsf{predicate}_n$ is $\mathtt{Prop}$ (as opposed to, *e.g.*, some kind of stratified $\mathtt{Prop}_n$), it is possible to define universal and existential quantifiers for $\mathsf{predicate}_n$ that can quantify over all of the types in Coq, even including $\mathsf{predicate}_n$. This is much stronger and more useful than *predicative quantification*, where a $\mathsf{predicate}_n$ could at most quantify over $\mathsf{predicate}_{n-1}$.

One major drawback in the model developed for FPCC is that the indexes in the model "leak out", meaning that the entire type soundness proof in the FPCC system must explicitly deal with the indexes. In our model this is not the case, and we have used $\bullet\!\!=$ to define $\mathsf{predicate}_n$,

$\mathsf{resource}_n$, world_n, and rmap_n, indicating that the rest of the proof cannot see the underlying stratification.

7.2.2 Dependent types to hide stratification

The second section in figure 7.2 contains one of the major insights in the modal model of Appel et. al [AMRV07], namely the ability to hid the underlying indexes in the model from the rest of the proof by using dependent types. Explicit reasoning about the indexes can therefore be contained to a small part of the overall soundness proof.

We use both sum (Σ) and product (Π) dependent types in our definitions. An element $s : \Sigma\alpha.\ (\beta : \alpha \rightarrow \mathsf{Type})$ is a **dependent pair**, written $s = \{a : \alpha,\ b : \beta(a)\}$, where the *type* of the second component depends on the *value* of the first component[3].

An element $f : \Pi\alpha.\ (\beta : \alpha \rightarrow \mathsf{Type})$ is a **dependent function** $f = \lambda a : \alpha.(b\ a) : \beta(a)$, where $b : \alpha \rightarrow\ \beta(a)$. Thus, the *type* of the function's result depends on the *value* of the function's parameter. One way to interpret this is that the Π type operator defines a family of related types, indexed by the value.

Using the Σ dependent type constructor, we are able to define $\mathsf{resource}$ as a dependent pair of a natural n and a stratified $\mathsf{resource}_n$ at level n. Similarily, we define rmap as a dependent pair of a natural n and a stratified rmap_n at level n. These definitions elegantly hide the underlying stratification by bundling it into a dependent pair, and are

[3]To express a normal pair we use the typical notation (a, b); the notation $\{a, b\}$ is reserved to indicate the presence of dependent types.

the definitions axiomatized in section 6.2.2. Finally, predicate is defined as was promised in section 6.2.1, as a function from a pair of rmap and mem to Prop.

We have used$\multimap$ to define resource and rmap, indicating that they are opaque to the rest of the proof. The rest of the Coq development *does not know* that the underlying model uses dependent types. When the rest of the soundness proof wishes to use resource and rmap, it must use the axioms exposed by the module, such as NO and YES as defined in section 7.2.6. Therefore the rest of the proof can be blissfully unaware of the stratification going on under the hood.

In contrast, we have used = to define world and predicate, indicating that those definitions are fully exposed to the rest of the proof.

7.2.3 Relating the models

The definitions in the third section of figure 7.2 are used only under the hood. However, as we shall see in section 7.2.6 they are critical to defining some parts of the interface that the rest of the proof does use, such as the YES pseudoconstructor.

Within the model, often it is useful to know the amount of stratification in a given rmap. The function level allows just that, taking a rmap and returning the value of its first component.

Another thing the model must be able to do is take an external view of the model (*i.e.*, a predicate) and translate it into the stratified construction (*i.e.*, a $\mathsf{predicate}_n$). A predicate in some sense is infinitely

stratified; accordingly, it is important to be able to project a predicate down into the finitely stratified model.

This is the job of the stratify operator[4]. The stratify operator takes a natural n and a predicate P, and produces a $\mathsf{predicate}_n\ P_n$, which is P stratified to level n. It does this by using the recursion operator fix in the Calculus of Constructions to recursively define the nested structure of the stratified predicate.

As explained above, a stratified predicate at level 0 is has type `unit`; the value `tt` has the `unit` type. A stratified predicate at level $n+1$ is a pair of a stratified predicate at level n and a function from a pair of rmap at level n and mem to `Prop`. The first component of this pair is defined by the recursive call to the stratify operator. The second component is more interesting, since it is the essential point of the stratification:

$$\lambda v : \mathsf{world}_{n-1}.\ P(v.1,\ \{n-1, v.2\},\ v.3)$$

By the definition of $\mathsf{predicate}_n$, this must have type $\mathsf{world}_{n-1} \to$ `Prop`, and accordingly the parameter v has type world_{n-1}. Recall that $\mathsf{world}_{n-1} = \mathsf{locals} \times \mathsf{rmap}_{n-1} \times \mathsf{mem}$. The result type must be `Prop`, which is simple enough to satisfy on its own (*e.g.*, with **true**), but the goal is to build the stratified P_n from P.

Given that we have a stratified rmap_{n-1} as the second component of v, we can build an rmap with the dependent pair $\{n-1,\ v.1\}$. By

[4]Appel et al. [AMRV07] write this operator as $\lfloor\cdot\rfloor_n$, and is define it in a more mathematical style as definition 29.

projecting out the first and third components of v as well we are able to build a world, to which we can apply the $\mathsf{predicate}$ P.

Since P is being applied to a rmap_{n-1} of level $n-1$, it will only be able to judge properties involving the rmap that are no more than $n-1$ levels of stratification deep. As an example, consider the value

$$\mathsf{stratify}\ 1\ P.$$

Simplifying the definition yields

$$(\mathtt{tt},\ \lambda v : (\mathsf{locals} \times \mathsf{rmap}_0 \times \mathsf{mem}).\ P(v.1,\ \{0,\ v.2\},\ v.3)).$$

Unfolding the definition of rmap_0 yields

$$(\mathtt{tt},\ \lambda v : (\mathsf{locals} \times (\mathsf{address} \to \mathsf{resource}_0) \times \mathsf{mem}).\ P(v.1,\ \{0,\ v.2\},\ v.3)).$$

A $\mathsf{resource}_0$ is either NO_0 or $\mathsf{YES}_0(k, \pi, \vec{P}_0)$. However, $\mathsf{predicate}_0$ is equal to $\mathtt{unit}$, so the list $\vec{P}_0$ is a list of elements of type $\mathtt{unit}$. Of course, one $\mathtt{unit}$ is much like another; this means that $\mathsf{stratify}\,1\,P$ **cannot distinguish** two YES_0 resources by examining their $\mathsf{predicate}$s.

The inability to fully distinguish YES_n resources by examining their predicates has significant implications for inversion. Suppose we have two predicates P and Q, and we know that for a given k

$$\mathsf{stratify}\ k\ P\ =\ \mathsf{stratify}\ k\ Q.$$

What can we conclude? Certainly not $P = Q$, since perhaps at some greater level of stratification $k' > k$ the two predicates will differ. In fact, in the worst case $k = 0$, and we have the horrible-looking fact that

$$\mathsf{stratify}\ 0\ (\lambda w.\,\mathbf{true})\ =\ \mathsf{stratify}\ 0\ (\lambda w.\,\mathbf{false}),$$

since, of course, $\mathtt{tt} = \mathtt{tt}$.

The best we can do is prove that

$$\mathsf{stratify}\ k\ P\ =\ \mathsf{stratify}\ k\ Q$$

if and only if the following characterization (C1) holds:

$$\text{(C1)} \qquad \forall j < k, \phi_j, \rho, m.\quad P(\rho,\ \{j,\ \phi_j\},\ m) \leftrightarrow Q(\rho,\ \{j,\ \phi_j\},\ m).$$

In other words, $\mathsf{stratify}\ k\ P = \mathsf{stratify}\ k\ Q$ if and only if P and Q are equivalent for all rmaps with stratification level **strictly** less than k.

7.2.4 Characterizing Increasing Approximation

The problem with (C1) is that it both exposes the inner workings of the stratified model by universally quantifying over ϕ_j and exposes the inner workings of the dependent model by explicitly constructing the rmap $\{j,\ \phi_j\}$. As a characterization of the behavior of the $\mathsf{stratify}$ operator it is not ideal, since the rest of the proof will not be able to use it.

Increasing Approximation in Stratified Model

$$
\begin{array}{rcl}
\mathsf{age1\_predicate}_{n+1} & : & \mathsf{predicate}_{n+1} \rightarrow \mathsf{predicate}_n \\
& \bullet\!\!- & \lambda(P_n, P_{n+1}).\, P_n \\
\\
\mathsf{age1\_resource}_{n+1} & : & \mathsf{resource}_{n+1} \rightarrow \mathsf{resource}_n \\
& \bullet\!\!- & \lambda\xi_{n+1}.\, \mathsf{match}\ \xi_{n+1}\ \mathsf{with} \\
& & \quad \mid \mathsf{NO}_{n+1} \Rightarrow \mathsf{NO}_n \\
& & \quad \mid \mathsf{YES}_{n+1}(k, \pi, \vec{P}_{n+1}) \Rightarrow \\
& & \qquad \mathsf{YES}_n(k, \pi, \mathsf{map}\ \mathsf{age1\_predicate}_{n+1}\ \vec{P}_{n+1}) \\
& & \mathsf{end} \\
\\
\mathsf{age1\_rmap}_{n+1} & : & \mathsf{rmap}_{n+1} \rightarrow \mathsf{rmap}_n \\
& \bullet\!\!- & \lambda\phi_{n+1}. \\
& & \quad \lambda(a : \mathsf{address}).\, \mathsf{age1\_resource}_{n+1}(\phi_{n+1}(a))
\end{array}
$$

Increasing Approximation in Dependent Model

$$
\begin{array}{rcl}
\mathsf{age1} & : & \mathsf{rmap} \rightarrow \mathsf{option}(\mathsf{rmap}) \\
& \multimap & \lambda\{n,\, \phi_n\}.\, \mathsf{match}\ n\ \mathsf{with} \\
& & \quad \mid 0 \Rightarrow \mathsf{None} \\
& & \quad \mid m+1 \Rightarrow \mathsf{Some}\ \{m,\, \mathsf{age1\_rmap}_{m+1}\ \phi_n\} \\
& & \mathsf{end} \\
\\
\mathsf{age} & : & \mathbb{N} \rightarrow \mathsf{rmap} \rightarrow \mathsf{option}(\mathsf{rmap}) \\
& = & \mathsf{fix}\ \mathsf{age}\ \lambda n.\, \lambda\phi.\, \mathsf{match}\ (n,\ \mathsf{age1}\ \phi)\ \mathsf{with} \\
& & \quad \mid (0,\ \_) \Rightarrow \mathsf{Some}\ \phi \\
& & \quad \mid (\_,\ \mathsf{None}) \Rightarrow \mathsf{None} \\
& & \quad \mid (m+1,\ \mathsf{Some}\ \phi') \Rightarrow \mathsf{age}\ m\ \phi' \\
& & \mathsf{end}
\end{array}
$$

Figure 7.3: Increasing approximation

Since the idea behind (C1) is that P and Q are equivalent for all rmaps with stratification level strictly less than k, it seems reasonable to characterize this property in a way that does not expose the underlying model.

In figure 7.3 we build the machinery to do just that. First we define three operators, $\mathsf{age1\_predicate}_{n+1}$, $\mathsf{age1\_resource}_{n+1}$, and $\mathsf{age1\_rmap}_{n+1}$ that remove one level of stratification from $\mathsf{predicate}_{n+1}$, $\mathsf{resource}_{n+1}$, and rmap_{n+1}, respectively, by "forgetting" some stratification. A point to note is that $\mathsf{age1\_predicate}_{n+1}$, which causes the actual information loss, has a very simple definition due to the pair structure of $\mathsf{predicate}_n$. All three of these operators are undefined on stratification level 0, and, since all of the operators expose the underlying stratification, we wish to completely hide them from the rest of the proof.

Now that we have defined aging on the stratified model, we extend it to the dependent model with the age1 operator, which simply unpacks the dependent pair and then uses the $\mathsf{age1\_rmap}_{n+1}$ operator. Since we would like age1 to be defined for all rmaps, we have it return None if $j = 0$. Since age1 is actually removing structure, it is irreflexive, *i.e.*,

$$\forall \phi, \phi'.\ \mathsf{age1}\ \phi = \mathsf{Some}\ \phi' \rightarrow \phi \neq \phi'$$

The age1 operator is opaque; the rest of the proof can see that it exists, but is only allowed to use it via axioms defined in the module type. Finally, we define the age operator, which is just the nth composition

of age1. For all $n > 0$, age is irreflexive[5].

Now that we have defined age, we can use it to characterize the stratify operator by observing that

$$\mathsf{stratify}\ k\ P\ =\ \mathsf{stratify}\ k\ Q$$

is equivalent to (C1), which is in turn equivalent to

$$\begin{aligned}(\text{C2}) \qquad & \forall \phi, n, \phi', \rho, m. \\ & (\mathsf{level}\ \phi = k)\ \rightarrow\ (\mathsf{age}\ (n+1)\ \phi\ =\ \mathsf{Some}\ \phi')\ \rightarrow \\ & (P(\rho, \phi', m) \leftrightarrow Q(\rho, \phi', m)).\end{aligned}$$

This is a much better characterization of stratify because it does not expose the stratified construction. However, it is not ideal, since it requires exposing the private level function. We will see a still more elegant way of expressing this property in section 7.3.

7.2.5 Stratified separation algebras for the model

In section 4.4 we defined a stratified separation algebra by building on the ideas of Calcagno et al. [COY07]. In section 4.5 we explained that shares are such an algebra. Here we will show how to lift the separation algebra on share to resource, rmap, and world.

[5]For comparison with previous work [AMRV07], the relation $R(\phi, \phi') = \exists n.\ (\mathsf{age}\ (n+1)\ \phi\ = \mathsf{Some}\ \phi')$ is irreflexive, transitive, and Noetherian, and moves between the worlds in the underlying Kripke model.

Join relations for the stratified model

We define the join relation $\oplus_n$ of $\mathsf{resource}_n$ as follows. Two YES_n resources join

$$\mathsf{YES}_n(k, \mathsf{Some}\ \pi_1, \vec{P}_n) \oplus_n \mathsf{YES}_n(k, \mathsf{Some}\ \pi_2, \vec{P}_n) \ = \ \mathsf{YES}_n(k, \mathsf{Some}\ \pi_3, \vec{P}_n)$$

if and only if

$$\pi_1 \ \oplus \ \pi_2 \ = \ \pi_3.$$

In other words, for two YES_n resources to join, they must be the same kind, have shares that join, and have identical predicate lists. NO_n is the identity element for level n. Two $\mathsf{resource}_n$ at different levels never join.

From the join relation on $\mathsf{resource}_n$ we can build the join relation on rmap_n by lifting the relation pointwise[6]. That is, for $\phi_a\ \phi_b\ \phi_c\ :\ \mathsf{rmap}_n$, we define

$$\phi_a \ \oplus_n \ \phi_b \ = \ \phi_c$$

if and only if

$$\forall l. \quad \phi_a(l) \ \oplus_n \ \phi_b(l) \ = \ \phi_c(l).$$

As with $\mathsf{resource}_n$, two rmap_n at different levels do not join.

From the join relation on rmap_n, we define the join relation on world_n by requiring the rmap_n to join and all other members of the tuple to be

[6]In the presentation we "overload" the $\oplus_n$ and $\oplus$ symbols, using the same symbol for resource, rmap, etc.

equal:

$$(\rho, \phi_a, m) \oplus_n (\rho, \phi_b, m) = (\rho, \phi_c, m)$$

if and only if

$$\phi_a \oplus_n \phi_b = \phi_c.$$

Join relations for the dependent model

Once we have defined the join relation $\oplus_n$ on elements $\xi_a\ \xi_b\ \xi_c\ :\ \mathsf{resource}_n$, we can define the join relation $\oplus$ on $\mathsf{resource}$ by saying that

$$\{n_1, \xi_a\} \oplus \{n_2, \xi_b\} = \{n_3, \xi_c\}$$

if and only if

$$n_1 = n_2 = n_3 \ \wedge\ \xi_a \oplus_n \xi_b = \xi_c.$$

We can then define the join relation on rmap by lifting the join relation on $\mathsf{resource}$ pointwise, *i.e.*,

$$\phi_a \oplus \phi_b = \phi_c$$

if and only if

$$\forall l.\ (\phi_a\ @\ l) \oplus (\phi_b\ @\ l) = (\phi_c\ @\ l).$$

Finally, we can define the join relation on world by requiring the

rmaps to join and all other members of the tuple to be equal:

$$(\rho, \phi_a, m) \oplus (\rho, \phi_b, m) = (\rho, \phi_c, m)$$

if and only if

$$\phi_a \oplus \phi_b = \phi_c.$$

Stratified separation algebras

When the join relation is defined in this way, $\mathsf{resource}_n$, resource, rmap_n, rmap, world_n, and world become stratified separation algebras as defined in section 4.4; proved in Coq[7].

The join relations for $\mathsf{resource}_n$, rmap_n, and world_n and the proof that they form stratified separation algebras are completely hidden from the rest of the proof. However, the join relations for resource, rmap, and world and the fact that they form stratified separation algebras are exposed to the rest of the proof and are used to reason about the substructural elements in the model.

7.2.6 The public interface

The rest of the CSL soundness proof should never see the internals of the stratification. Instead, the rest of the proof should treat the definitions of resource and rmap as opaque and handle elements of type resource and rmap by using a carefully designed interface.

[7] In the actual Coq development, a special kind of function constructor, called a `Joinmap`, was used to lift the separation algebra properties from resource to rmap.

That interface consists of three parts. The first is a number of functions whose existence and type signature are exported to the rest of the proof but whose internals are hidden by the module type. The second is a series of helper definitions which are defined in terms of the opaque functions and are fully visible to the rest of the proof. We have already discussed the age1 and age operators, which were given in figure 7.3. The first of these is an opaque definition; the second is a helper definition. The third part of the interface is a long series of axioms which show how the opaque functions behave. For example, one exported axiom involving age1 is

$$\forall \phi, \phi.\ \mathsf{age1}\ \phi = \mathsf{Some}\ \phi' \ \rightarrow\ \phi \neq \phi'.$$

In other words, age1 is irreflexive.

Figure 7.4 gives some of the most important opaque functions, with the definitions of resource_at and the pseudoconstructors NO and YES. These have the types given in figure 6.3 and explained in section 6.2.2. In that section we also explained how to build up the other pseudoconstructors such as LK from YES.

The join relation $\oplus$ was discussed in section 7.2.5. To reason about the join relation we export all of the axioms of a stratified separation algebra given in section 4.4, as well as property 8 from section 4.5.

The resource_at function takes an rmap and an address and returns the resource associated with that address. As mentioned in section 6.2.2,

Opaque Interface to Modal Substructural Model

$$
\begin{array}{rcl}
\oplus & : & \mathsf{rmap} \times \mathsf{rmap} \times \mathsf{rmap} \rightarrow \mathtt{Prop} \\
 & \multimap & \text{As defined in section 7.2.5} \\
 & & \\
\mathsf{resource\_at} & : & \mathsf{rmap} \rightarrow \mathsf{address} \rightarrow \mathsf{resource} \\
 & \multimap & \lambda\{n,\ \phi_n\}.\ \lambda l.\ \{n,\ \phi_n(l)\} \\
 & & \\
\mathsf{NO} & : & \mathsf{rmap} \rightarrow \mathsf{resource} \\
 & \multimap & \lambda\phi.\ \{\mathsf{level}\ \phi,\ \mathsf{NO}_n\ (\mathsf{level}\ \phi)\} \\
 & & \\
\mathsf{YES} & : & \mathsf{rmap} \times \mathsf{kind} \times \mathsf{pshare} \times \mathsf{list}(\mathsf{predicate}) \\
 & & \quad \rightarrow \mathsf{resource} \\
 & \multimap & \lambda(\phi,\ k,\ \pi,\ \vec{P}). \\
 & & \quad \{\mathsf{level}\ \phi,\ \mathsf{YES}_n(k,\ \pi,\ \mathsf{map}\ (\mathsf{stratify}\ (\mathsf{level}\ \phi))\ \vec{P})\}
\end{array}
$$

Transparent Interface to Modal Substructural Model

$$
\begin{array}{rcl}
\perp & : & \mathsf{rmap} \times \mathsf{rmap} \rightarrow \mathtt{Prop} \\
 & = & \lambda(\phi_1, \phi_2).\ \exists\phi_3.\ \phi_1 \oplus \phi_2 = \phi_3 \\
 & & \\
\mathsf{same\_age} & : & \mathsf{rmap} \times \mathsf{rmap} \rightarrow \mathtt{Prop} \\
 & = & \lambda(\phi_1, \phi_2).\ \exists\phi_e.\ \phi_1 \perp \phi_e\ \wedge\ \phi_2 \perp \phi_e
\end{array}
$$

Figure 7.4: Interface to substructural modal model

resource_at is usually written infix using the @ symbol, a convention we will follow in this presentation as well.

The NO pseudoconstructor takes an rmap ϕ; out of this it extracts the level n and constructs the dependent pair $\{n,\ \mathsf{NO}_n\}$, which has type resource. The YES pseudoconstructor takes an rmap ϕ, a kind k, a pshare π, and a list of predicates $\vec{P}$, extracts the level n from ϕ, maps the stratify function at level n over $\vec{P}$ to get $\vec{P}_n$, and constructs the dependent pair $\{n,\ \mathsf{YES}_n(k,\ \pi,\ \vec{P}_n)\}$, which also has type resource.

We are now in a position to understand the ramifications for inverting the YES pseudoconstructor, as promised in section 6.2.2. By examining the definitions and understanding the results of inverting the stratify function, it is clear that

$$\mathsf{YES}(\phi, k, \pi, \vec{P}) = \mathsf{YES}(\phi', k', \pi', \vec{P}'),$$

if and only if level ϕ = level ϕ', $k = k'$, $\pi = \pi'$, and the predicates in $\vec{P}$ and $\vec{P}'$ are equivalent up to stratification level ϕ.

Unfortunately this characterization of YES inversion exposes the underlying model. When we define our modal substructural logic it will be possible to express this property in a very elegant way (see section 7.3.10).

We also give the helper definitions $\perp$ and same_age. $\perp$ is defined exactly as explained in section 4.3. The same_age relation, which is defined in terms of the join relation, states that two resource maps are

in the same equivalence class, as explained in section 4.4. There is a convenient connection between the public same_age relation and the private level function:

$$\mathsf{same\_age}(\phi, \phi')$$

if and only if

$$\mathsf{level}\ \phi = \mathsf{level}\ \phi'.$$

7.3 Reasoning about the model with a modal substructural logic

We have now given an explanation of the underlying stratified separation algebra, and have explained how one can use dependent types to hide the stratification from the rest of the proof. However, even reasoning about the dependently typed model can be laborious.

The second major insight of the modal model of Appel et al. [AMRV07] was that one can define a modal logic for reasoning about the model. Various operators in the modal logic will reason about different aspects of the model. The modal logic then becomes a clean interface to the opaque dependent model, which itself is a clean interface for the ugly details of the stratification.

In the Coq development we take this idea a step further [DAH08]. In general we have found that when we get to a point in the Coq development where the way forward is difficult or unclear, the solution

is to redefine the problem in terms of our modal substructural logic, frequently with the aid of a new operator. Since our logic has an underlying semantics, adding a new operator is quite easy. Reformulating a problem in the modal logic often allows the correct solution to become more apparent.

7.3.1 Assertions in the modal substructural logic

An assertion in the modal substructural logic is just a predicate. Accordingly one defines it by giving a function from world to `Prop`. We say that a world w forces a predicate P, written

$$w \models P,$$

when $P\ w$ holds. Often we wish to say that one assertion P implies another one Q, *i.e.*,

$$\forall w.\ \ w \models P \ \rightarrow\ w \models Q.$$

We will use the notation

$$P \vdash Q$$

to mean this kind of implication. When $P = \mathbf{true}$, we will simply write

$$\vdash Q.$$

Since our assertions are just functions from world to Prop, we have what is called a *shallow embedding* of our modal substructural logic in Coq. This would not be the case if we defined our own proposition language (calling it, *e.g.*, nuProp) and used it instead of Coq's Prop. The shallow embedding significantly simplifies the process of creating the machine-checked proofs. We are able to use tactics developed for Prop with our predicates; for example, we can use the destruct tactic to break apart the conjunction in our modal logic. Also, we are able to use Coq's native variable-binding mechanisms, thereby avoiding the binder issues raised in the POPLmark quagmire [ABF+05].

Almost all of our assertions are public in the sense that the remainder of the proof can see and use the definitions to reason about the underlying model. One exception is the definition of higher-order recursion operator μ_{HO}, which is opaque; the rest of the proof is given fold-unfold rules to reason about that operator as explained in section 7.3.9.

In the rest of this chapter we will give a selection of operators in our logic, explain their use, and provide their models.

7.3.2 Logical operators

In figure 7.5 we give some basic logical operators and their models. $[A]_{\mathsf{Coq}}$ indicates that the base (Coq) logic assertion A holds; A does not depend on the value of the world w. We do not have to put restrictions that w is free in A since our higher-order shallow embedding ensures

$$
\begin{aligned}
[A]_{\mathsf{Coq}} &= \lambda w.\, A \\
\mathbf{true} &= [\mathbf{true}]_{\mathsf{Coq}} \\
\mathbf{false} &= [\mathbf{false}]_{\mathsf{Coq}} \\
P \wedge Q &= \lambda w.\, P\, w \ \wedge\ Q\, w \\
P \vee Q &= \lambda w.\, P\, w \ \vee\ Q\, w \\
\forall v.P(v) &= \lambda w.\, \forall v.\, P(v)\ w \\
\exists v.P(v) &= \lambda w.\, \exists v.\, P(v)\ w \\
\mathsf{exactly}\ \phi &= \lambda(\rho, \phi', m).\, \phi = \phi'
\end{aligned}
$$

Figure 7.5: Models of logical assertions

that we avoid variable capture.

$[\cdot]_{\mathsf{Coq}}$ useful because it ensures that the entire expressive power of Coq is available in the logic. The simplest use of $[\cdot]_{\mathsf{Coq}}$ is to define the predicates **true** and **false**[8]. We will see a more powerful use of $[\cdot]_{\mathsf{Coq}}$ when we define the precisely operator in section 7.3.8.

One of the advantages of our approach is that it is easy to define new operators in terms of old ones; we have done this in the definitions of **true** and **false**. Although both are predicates—that is, functions from world to `Prop`—their definition does not contain a λ; instead they utilize the λ in $[\cdot]_{\mathsf{Coq}}$.

[8]To avoid "symbol explosion" in the presentation, we overload many symbols to operate both at the `Prop` and predicate levels. Since the two have different types in Coq, only one will be possible in any given context.

Conjunction $\wedge$ and disjunction $\vee$ are defined by lifting the operations defined on `Prop`. We also lift the universal and existential quantifiers $\forall$ and $\exists$. Since we use the quantifiers at the `Prop` level in the definitions of universal and existential quantification in our logic, we have full impredicative quantification: v can thus range over values, predicates, worlds, shares, and any other types definable in Coq.

We define the operator exactly so that we can precisely specify the resource map if we wish; exactly is not particularly useful in the CSL correctness proofs of programs, but is used in both the soundness proofs and in defining the concurrent operational semantics in chapter 8.

7.3.3 Modal operators

In figure 7.6 we give some modal operators and their models[9]. Recall that a world is a tuple of locals ρ, resource map ϕ, and memory m. The modal operators enable us to reason cleanly and orthogonally about these elements.

The simplest modal operator is close, which is used to remove the effect of the locals ρ on a predicate's behavior by universally quantifying over them. Thus, for any predicate P, close P ignores the locals ρ. If P already ignores ρ, then close $P = P$ and we say that P is *closed*.

We include three operators to reason about the resource map ϕ: approximately P, which we write as $\rhd P$; necessarily P, written $\Box P$; and

[9]Dockins et al. [DAH08] discuss the reason for calling these operators "modal"; the idea is that a modal operator is an operator that describes a relation in an underlying Kripke models.

$$
\begin{array}{rcl}
\mathsf{close}\ P & = & \lambda(\rho, \phi, m).\ \forall \rho'. \\
 & & \quad P(\rho', \phi, m) \\
 & & \\
\triangleright P & = & \lambda(\rho, \phi, m). \\
 & & \quad \forall n, \phi'.\ (\mathsf{age}\ (n+1)\ \phi = \mathsf{Some}\ \phi') \rightarrow \\
 & & \quad\quad P(\rho, \phi', m) \\
 & & \\
\Box P & = & P \wedge \triangleright P \\
 & = & \lambda(\rho, \phi, m). \\
 & & \quad \forall n, \phi'.\ (\mathsf{age}\ n\ \phi = \mathsf{Some}\ \phi') \rightarrow \\
 & & \quad\quad P(\rho, \phi', m) \\
 & & \\
\bigcirc P & = & \lambda(\rho, \phi, m). \\
 & & \quad \forall \phi'.\ \mathsf{same\_age}(\phi, \phi') \rightarrow \\
 & & \quad\quad P(\rho, \phi', m) \\
 & & \\
!P & = & \lambda(\rho, \phi, m).\ \forall m'. \\
 & & \quad P(\rho, \phi, m')
\end{array}
$$

Figure 7.6: Models of modal assertions

fashionably P, written $\bigcirc P$. In previous work [AMRV07] approximately was called later, and for this reason the Coq development refers to this operator as `later`. Each quantifies over the resource map ϕ in a different way. Unlike with the locals ρ, we cannot simply use universal quantification over all possible ϕ, since the underlying model does not allow a stratified resource map ϕ_n to say anything meaningful about stratified resource maps ϕ_{n+m+1} of higher stratification.

While a stratified resource map ϕ_n cannot say anything meaningful about resource maps of higher stratification, it can describe properties of resource maps of lesser stratification ϕ_{n-m-1}. This was the underlying

idea behind the approximation operator age_n defined in section 7.2.4. The $\triangleright P$ operator models the central idea of that section as a modal operator. If $(\rho, \phi, m) \models \triangleright P$, then P may or may not hold on (ρ, ϕ, m), but will hold on all ϕ' strictly more approximate than ϕ: $(\rho, \phi', m) \models P$.

Since all that is required for $\mathsf{close}\ P = P$ is that P not depend on local variables, it is natural to wonder which P have the property that $\triangleright P = P$. Due to the irreflexivity of $\mathsf{age}\,(n+1)$, $\triangleright P = P$ is a very strong requirement; in fact

$$\triangleright P = P$$

if and only if

$$P = \mathbf{true}.$$

This makes sense, since the most approximate predicate—that is, the one that gives the least information—is **true**. The essentially irreflexive nature of the $\triangleright$ operator was a key technical insight of the modal model of Appel et al. [AMRV07], and we will see its expressive power when we see how to invert YES in section 7.3.10.

In the modal logic, $\triangleright$ serves two purposes. The first is to give a compact characterization of approximation, which it does very well. The second is somewhat different; it is to restrict the "wildness" of terms in the underlying stratified model. There is a natural idea that if $w \models P$, and if w' is a more approximate version of w, then $w' \models P$. In informal terms, if w is "good enough" to guarantee P, then if w

becomes a little more approximate, it should be even easier for it to guarantee P. Reformulated in the modal logic, we would very much like our P to have the property that

$$P \vdash \rhd P,$$

a property we call *necessary*. Unfortunately, there are some predicates P for which this property is not true. For example, consider

$$P_{\mathsf{wild}} = \lambda(\rho, \phi, m).\ \mathsf{level}\ \phi > 5.$$

This will be true as long as ϕ has level greater than 5, but false as ϕ nears level 0.

The modal operator $\mathsf{necessarily}\ P$, written $\Box P$, forces predicates to be necessary. In figure 7.6 we give two equivalent defintions for $\Box P$. The first is that $\Box P = P \wedge \rhd P$. From this definition it is easy to show

$$\Box P \vdash P$$

and

$$\Box P \vdash \rhd P.$$

From these properties we can redefine *necessary* in terms of $\Box P$: P is necessary if and only if

$$\Box P = P.$$

The second definition of $\Box$ in figure 7.6 shows that it is very similar to a reflexive version of $\rhd$, and leads naturally to the proof that for all P,

$$\Box P = \Box \Box P.$$

Since $\Box P$ is idempotent, the property

$$\Box P \;=\; P$$

is not particularly restrictive; P is necessary as long as it does not try to expose the underlying stratified construction. In fact, almost all of the operators presented here are necessary automatically, or become necessary when applied to necessary predicates. Therefore, as long as we build our CSL proofs using the operators presented here, we can ignore the existence of unnecessary predicates. One major exception is that implication is not automatically necessary even when applied to necessary predicates; this is discussed in section 7.3.7.

The modal operator $\mathsf{fashionably}\, P$, written $\bigcirc P$, quantifies over the ϕ in the world in a different way. While $\rhd P$ and $\Box P$ were primarily concerned with reasoning about the underlying stratified construction, $\bigcirc P$ is used to force P to ignore ϕ, similar to the way that $\mathsf{close}\ P$ is used to force P to ignore ρ.

Unlike in $\mathsf{close}\ P$, where we quantify over all possible ρ, in $\bigcirc P$ we cannot quantify over all possible ϕ, since resource maps of level n are not able to say anything meaningful about resource maps of level $n+m+1$.

Accordingly, we quantify over all resource maps of the same level; this is the strongest meaningful quantification possible. Put another way, $\bigcirc P$ is only allowed to utilize one fact about the resource map ϕ, which is its level. If we want to quantify over all resource maps at this level of stratification and lower, we combine $\Box P$ and $\bigcirc P$ to get $\Box \bigcirc P$.

We use the modal operator $\mathsf{everywhere}\, P$, written $!P$, to quantify over the memory m. The definitions of $!P$ and $\mathsf{close}\ P$ are similar, since the memory m is similar to the locals ρ in the sense that neither has a particularly complicated structure that interferes with the model. If a predicate P ignores the memory, then

$$!P \ = \ P,$$

and we say that P is *ubiquitous*. Just as there are many closed predicates, there are many ubiquitous ones.

7.3.4 Substructural operators

In figure 7.7 we give three basic substructural operators and their models. The assertion **emp** holds if the resource map ϕ only contains NO resources. Recall from section 7.2.6 that NO takes a resource map ϕ as a parameter and creates a resource stratified to the same level as ϕ, which it packages up into a dependent pair. Since a $\mathsf{predicate}$ is a function from world, we can use the resource map contained in the world.

The assertion $l \Subset (k, \pi, \vec{P})$ holds when the resource map ϕ contains

$$
\begin{array}{rcl}
\mathbf{emp} & = & \lambda(\rho, \phi, m). \\
 & & \quad \forall l.\ \phi \mathbin{@} l \ = \ \mathsf{NO}\ \phi \\
 & & \\
l \circledcirc (k, \pi, \vec{P}) & = & \lambda(\rho, \phi, m). \\
 & & \quad (\forall l'.\ l \neq l' \rightarrow \phi \mathbin{@} l \ = \ \mathsf{NO}\ \phi) \wedge \\
 & & \quad \phi \mathbin{@} l \ = \ \mathsf{YES}\ \phi\ k\ \pi\ \vec{P} \\
 & & \\
P * Q & = & \lambda w. \exists w_1, w_2. \\
 & & \quad w_1 \oplus w_2 \ = \ w \ \wedge \ P\, w_1 \ \wedge \ Q\, w_2
\end{array}
$$

Figure 7.7: Models of substructural assertions

$\mathsf{YES}\ \phi\ k\ \pi\ \vec{P}$ at location l and NO elsewhere. YES then creates a list of predicates stratified to $\mathsf{level}\ \phi$; we will take advantage of this in section 7.3.10 when we express what can be concluded from

$$\mathsf{YES}\ \phi\ k\ \pi\ \vec{P} \ = \ \mathsf{YES}\ \phi'\ k'\ \pi'\ \vec{P'}.$$

The requirement that all other locations are NO is standard for separation logic; larger structures are built with the separating conjunction.

Since we have defined our join relations as a stratified separation algebra, we model the separating conjunction $P * Q$ in a way similar to Calcagno et al. [COY07]. When

$$w \ \models \ P * Q,$$

then w can be split into two subworlds w_1 and w_2 such that

$$w_1 \oplus w_2 \ = \ w$$

and

$$(w_1 \models P) \ \wedge \ (w_2 \models Q).$$

Using the separating conjunction it is possible to describe complex substructural properties of the underlying model in an elegant way.

7.3.5 Reasoning in the modal substructural logic

The motivation for defining the modal substructural logic is that we have found it much easier to reason about the model using the logic than to reason on the underlying model directly. The proofs of the soundness of CSL can become very complicated, and it is not unusual to have 50 or more premises leading to a goal. In that context, it is vital to be able to express properties concisely in order to reason about them with minimal mental overhead.

One advantage of using our logic is that we can prove a large number of lemmas describing how the logic behaves; a small subset of them are pictured in figure 7.8. Most of the lemmas take the form of equalities in the logic, which is quite convenient since we can use powerful `rewrite` tactics in Coq.

A very useful strategy we used while developing the soundness proof was to express the premises and goal of a lemma in the logic, and then use the kinds of rules given in figure 7.8 to prove the goal from the hypothesis. Sometimes it was not possible complete the proof entirely in the logic, in which case we had two choices: to define a new logic

$$
\begin{aligned}
\Box P &= \Box\Box P \\
\Box \rhd P &= \rhd \Box P \\
\Box \rhd P &= \rhd P \\
\rhd \bigcirc P &= \bigcirc \rhd P \\
! \rhd P &= \rhd ! P \\
! \bigcirc P &= \bigcirc ! P \\
P \wedge Q &= Q \wedge P \\
\rhd (P \wedge Q) &= (\rhd P) \wedge (\rhd Q) \\
\bigcirc (P \wedge Q) &= (\bigcirc P) \wedge (\bigcirc Q) \\
! (P \wedge Q) &= (! P) \wedge (! Q) \\
\forall v.\, \rhd (P(v)) &= \rhd (\forall v.\, P(v)) \\
\Box \, \mathbf{emp} &= \mathbf{emp} \\
\Box (l \circledS (k, \pi, \vec{P})) &= l \circledS (k, \pi, \vec{P}) \\
(P \wedge P') * (Q \wedge Q') &= (P * Q) \wedge (P' * Q') \\
(\rhd P) * (\rhd Q) &= \rhd ((\Box P) * (\Box Q))
\end{aligned}
$$

Figure 7.8: Rules for reasoning in the modal substructural logic

operator to express the property and then continue to reason in the logic, or to dip briefly out of the logic and reason instead directly on the underlying model. The choice made depended on the difficulty of proving the underlying fact and a guess as to the amount of use the new operator would have. Dockins et al. [DAH08] discuss this in more detail.

To give an example of reasoning in the modal logic, we prove that

$$\Box \bigcirc P \;=\; \bigcirc \Box P.$$

For comparison, the equivalent statement on the model is

$$\begin{aligned}\forall \rho, \phi, m.\quad & \big(\forall n, \phi'.\ (\mathsf{age}\ n\ \phi = \mathsf{Some}\ \phi') \rightarrow \\ & \quad (\forall \phi''.\ \mathsf{same\_age}(\phi', \phi'') \rightarrow P(\rho, \phi'', m))\big) \;\leftrightarrow \\ & \big(\forall \phi'.\ \mathsf{same\_age}(\phi, \phi') \rightarrow \\ & \quad (\forall n, \phi''.\ (\mathsf{age}\ n\ \phi' = \mathsf{Some}\ \phi'') \rightarrow P(\rho, \phi'', m))\big)\end{aligned}$$

First we unfold the definition of necessarily on the left-hand side to get

$$(\bigcirc P) \;\wedge\; (\triangleright \bigcirc P) \;=\; \bigcirc \Box P,$$

and then rewrite $\triangleright \bigcirc P$ using a rule in figure 7.8 to get

$$(\bigcirc P) \;\wedge\; (\bigcirc \triangleright P) \;=\; \bigcirc \Box P.$$

$$
\begin{array}{rcl}
l \overset{\pi}{\mapsto} v & = & l \,\mathsf{o}\!\!-\!\!\mathsf{o}\, (\mathsf{kVAL},\ \pi,\ \mathtt{nil}) \wedge \\
 & & \lambda(\rho, \phi, m).\, m(l) = v \\
l \overset{\pi}{\rightsquigarrow} R & = & l \,\mathsf{o}\!\!-\!\!\mathsf{o}\, (\mathsf{kLK},\ (\mathsf{project}\ \pi\ \mathsf{into}\ \blacktriangleleft\!\blacktriangleright),\ R :: \mathtt{nil}) \\
\mathsf{hold}\ l\, R & = & l \,\mathsf{o}\!\!-\!\!\mathsf{o}\, (\mathsf{kLK},\ \blacktriangleleft\!\blacktriangleright,\ R :: \mathtt{nil}) \\
f : \{P\}\{Q\} & = & f \,\mathsf{o}\!\!-\!\!\mathsf{o}\, (\mathsf{kFUN},\ \blacktriangleleft\!\blacktriangleright,\ P :: Q :: \mathtt{nil})
\end{array}
$$

Figure 7.9: Models of Concurrent Separation Logic assertions

Next we rewrite again using the rule for fashionable intersection to get

$$\bigcirc(P \wedge \triangleright P) \ = \ \bigcirc \Box P,$$

which by the definition of necessarily is equal to

$$\bigcirc \Box P \ = \ \bigcirc \Box P. \qquad \Box$$

Of course, this can be proved directly on the underlying model, but the details and multiple quantifiers can quickly overwhelm an understanding of what is going on, particularly in the context of a larger proof, leading to a considerably longer and more frusterating proving experience.

7.3.6 Modelling the assertions of Concurrent Separation Logic

We are now in a position to model the assertions of CSL given in chapter 4. Their definitions, given in figure 7.9, are quite straightforward since we can take advantage of the operators defined so far.

The "maps-to" assertion of separation logic $l \overset{\pi}{\mapsto} v$ means that the resource map contains a YES at location l whose kind is kVAL. The associated share is π, and the associated predicate list is `nil`. In addition, the memory contains the value v at location l.

The "is a lock" assertion of Concurrent Separation Logic $l \overset{\pi}{\rightsquigarrow} R$ means that the resource map contains a YES at location l whose kind is kLK. The share π is projected into the right half of the full share in the style of the isomorphism noted by Parkinson [Par05]. In our share models there is an isomorphism between any two nonempty shares, and $\mathsf{project}$ uses that isomorphism to squeeze π into $\blacklozenge$. For example,

$$\mathsf{project}\ \blacklozenge\ \mathsf{into}\ \blacklozenge\ =\ \blacklozenge,$$

$$\mathsf{project}\ \blacklozenge\ \mathsf{into}\ \blacklozenge\ =\ \lfloor\blacklozenge\rfloor,$$

and

$$\mathsf{project}\ \blacklozenge\ \mathsf{into}\ \blacklozenge\ =\ \lceil\blacklozenge\rceil,$$

Finally, we put the lock invariant R into the predicate list.

The $\mathsf{hold}\ l\ R$ assertion is defined quite similarly to the lock assertion,

Dangerous Implication

$$P \Rightarrow Q \;=\; \lambda\sigma.\ P\sigma \Rightarrow Q\sigma$$

$$\neg P \;=\; P \Rightarrow \mathbf{false}$$

Safe Implication

$$P \subset Q \;=\; \Box \bigcirc !(P \Rightarrow Q)$$

$$P \cong Q \;=\; (P \subset Q) \wedge (Q \subset P)$$

Figure 7.10: Logical implication in the logic

with the exception that its share is the entire left half of the full share $\blacklozenge$. Accordingly to have the full share of a lock location l, so that it is safe to destroy it, one needs to have both $l \overset{\blacklozenge}{\rightsquigarrow} R$ and $\mathsf{hold}\ l\ R$, or, more concisely, $l \overset{\blacklozenge}{\rightsquigarrow} R\ *\ \mathsf{hold}\ l\ R$.

Finally, the "is a function" assertion $f : \{P\}\{Q\}$ means that the resource map contains a YES at location f whose kind is kFUN. The share is $\blacklozenge$, and the predicate list contains the pre- and postconditions P and Q.

7.3.7 Logical Implication

Almost all of our operators are suitable for use in a CSL proof about programs. However, there are a few operators that we do recommend avoiding in that context, particularly full logical implication, which

interacts complexly with the underlying stratified model, and which was conspicuously missing from the basic logical operators defined in figure 7.5. We define four related implication operators in figure 7.10; the first two are *dangerous* in the sense that if not used carefully they will expose wild predicates in the underlying model.

Of course nothing prevents the use of the dangerous forms of implication; since the system is semantic instead of syntactic, an end-user can utilize any operator he wishes. However, full implication can be more difficult to use than expected, and so we recommend that CSL proofs avoid it. Full implication is largely used in the soundness proofs of the model itself, indicating that it can be useful in carefully controlled situations. For most situations, however, we reccomend the use of the safer, more restricted forms of implication given in the second part of figure 7.10.

The problem with unrestricted implication is that it is not always necessary in the sense given in section 7.3.3, even when it is applied to necessary predicates; in other words,

$$\neg \big(\forall P, Q.\ \ \Box(\Box P \Rightarrow \Box Q) \ = \ \Box P \Rightarrow \Box Q\big).$$

Since implication is inherently contravariant in its antecedent, it interacts badly with the stratified construction.

Of course, for certain P and Q, the unrestricted implication $P \Rightarrow Q$ is necessary. In general, as long as P avoids mentioning the stratified

construction[10], and Q is necessary, then $P \Rightarrow Q$ will be necessary. Accordingly, we sometimes use unrestricted implication in the proof when these conditions are met. The assertion $\neg P$ is defined in terms of unrestricted implication and is likewise dangerous to use.

One can apply the $\Box$ operator to an implication to make it necessary (recall that $\forall P.\ \Box\,\Box\,P = \Box\,P$), but this can be quite problematic. If the implication $P \Rightarrow Q$ is not necessary to begin with, then the meaning of $\Box(P \Rightarrow Q)$ is extremely difficult to understand; in many cases it becomes **false**, but depending on P and Q it might have other behavior.

A better approach in many cases is to use the $P \subset Q$ operator. This is an implication protected by the modal operators $\Box$, $\bigcirc$, and !. Because we use the $\Box$ operator, $P \subset Q$ is necessary. Because we use the $\bigcirc$ operator, the predicates P and Q will not be able to expose wild terms in the underlying stratification.

Translated into informal language, the unrestricted implication $P \Rightarrow Q$ means, "If P holds on world (ρ, ϕ, m), then Q holds on world (ρ, ϕ, m)." The trouble is that $P \Rightarrow Q$ does not say anything about the relationship P and Q will have as the resource map ϕ becomes more approximate.

In contrast, the subset implication means, "For this and any more approximate resource map, whenever P holds, Q will hold." Often this is exactly what is needed to express a desired property, and it plays nicely with the model.

We define the equality relation on predicates $P \cong Q$ to be the

[10] *e.g.*, any time P uses the $l \,\S\, (k, \pi, \vec{P})$, then $\vec{P} = \texttt{nil}$.

conjunction of $P \subset Q$ and $Q \subset P$; informally it means, "On this any any more approximate resource map, P and Q are identical." Another way to think about the meaning of $P \cong Q$ is:

$$(\rho, \phi, m) \models P \cong Q$$

if and only if

$$\mathsf{stratify}\ (\mathsf{level}\ \phi + 1)\ P\ =\ \mathsf{stratify}\ (\mathsf{level}\ \phi + 1)\ Q.$$

We take advantage of this insight when we invert the YES pseudoconstructor in section 7.3.10.

7.3.8 Extensionality, validity, precision, and tightness

In section 4.7.1 we presented the notions of closure to local variables, validity, precision, and tightness, and informally explained the related operators close, validly, precisely, and tightly. We are at last ready to provide their models, which we do in figure 7.11.

The extensionally P operator is similar in some ways to the $!\,P$ operator in the sense that they both quantify over memories and thereby restrict the ability of P to depend on memory. However, while the $!\,P$ operator does not allow P to use any part of memory, extensionally P does allow P to use certain parts of memory.

$$
\begin{array}{rcl}
\mathsf{extensionally}\ P & = & \lambda(\rho,\phi,m).\,\forall m', \\
& & \big(\forall l,\pi,v. \\
& & \quad (\rho,\phi,m) \models \mathbf{true} * l \overset{\pi}{\mapsto} v \ \rightarrow \\
& & \quad (\rho,\phi,m') \models \mathbf{true} * l \overset{\pi}{\mapsto} v\big) \ \rightarrow \\
& & \quad (\rho,\phi,m') \models P \\
\\
\mathsf{validly}\ P & = & \Box\,\mathsf{extensionally}\ P \\
\\
\mathsf{precisely}\ P & = & P \wedge \Box \bigcirc !\ \forall \phi,\phi'. \\
& & \big(((P \wedge \mathsf{exactly}\ \phi)\ *\ \mathbf{true}) \wedge \\
& & \ ((P \wedge \mathsf{exactly}\ \phi')\ *\ \mathbf{true})\big)\ \Rightarrow \\
& & \ [\phi = \phi']_{\mathsf{Coq}} \\
\\
\mathsf{tightly}\ P & = & \mathsf{validly\ close\ precisely}\ P
\end{array}
$$

Figure 7.11: Extensionality, validity, precision, and tightness

In particular, extensionally P allows P to use any part of memory that P has permission to use. What we want to eliminate are predicates such as

$$P \ = \ \lambda(\rho,\phi,m).\, m(3) = 2.$$

The problem is that there is no restriction on ϕ to ensure that memory location 3 can be accessed. This is in contrast to our definition of maps-to in figure 7.7, where we are very careful to place restrictions on the resource map ϕ. Predicates that are well-behaved in this sense are called *extensional*. The extensionally P operator forces predictes to be extensional; if P is already extensional then

$$\mathsf{extensionally}\ P \ = \ P;$$

if not then

$$\mathsf{extensionally}\ P\ =\ \mathbf{false}.$$

We define $\mathsf{validly}\ P$ as $\Box\,\mathsf{extensionally}\ P$, and we say a predicate P is *valid* when

$$\mathsf{validly}\ P\ =\ P.$$

In other words, a predicate is valid when it ignores memory outside its resource map and is well-behaved under the approximation operation.

In section 4.7.1 we introduced the idea of a *precise* assertion. Recall that an assertion P is precise if for any predicate Q, the separating conjunction can only divide resources in a single way. The $\mathsf{precisely}\ P$ operator expresses this idea and forces a predicate to be precise in a way that is well-behaved under the approximation operation, and if P is precise then

$$\mathsf{precisely}\ P\ =\ P.$$

A sharp-eyed reader may notice the use of the dangerous implication operator in the definition of precisely, but since it is guarded by the $\Box$ and $\bigcirc$ operators, it is safe and precisely is necessary.

A predicate is *tight* if it is valid, closed, and precise, and we define the $\mathsf{tightly}\ P$ operator as $\mathsf{validly\ close\ precisely}\ P$. For all P, $\mathsf{tightly}\ P$ is tight. If P is tight then

$$\mathsf{tightly}\ P\ =\ P.$$

Higher-order Recursion

$$
\begin{array}{rcl}
\mathsf{contractive}_{\mathsf{HO}} & : & \forall\alpha.\,((\alpha \to \mathsf{predicate}) \to (\alpha \to \mathsf{predicate})) \to \mathtt{Prop} \\
& = & \lambda f.\,\forall P, Q. \\
& & \quad (\forall a.\ \triangleright(P\,a \ \cong\ Q\,a)) \vdash (\forall a.\,(f\,P\,a) \cong (f\,Q\,a)) \\
\\
\mathbf{false}_{\mathsf{HO}} & : & \forall\alpha.\,\alpha \to \mathsf{predicate} \\
& \multimap & \lambda a.\,\mathbf{false} \\
\\
\mu_{\mathsf{HO}} & : & \forall\alpha.\,((\alpha \to \mathsf{predicate}) \to (\alpha \to \mathsf{predicate})) \to \\
& & \quad \alpha \to \mathsf{predicate} \\
& \multimap & \lambda f.\,\lambda a.\,\lambda(\rho, \phi, m). \\
& & \quad f^{(1+\mathsf{level}\ \phi)}\ \mathbf{false}_{\mathsf{HO}}\ a\ (\rho, \phi, m)
\end{array}
$$

First-order Recursion

$$
\begin{array}{rcl}
\mathsf{contractive} & : & (\mathsf{predicate} \to \mathsf{predicate}) \to \mathtt{Prop} \\
& = & \lambda f.\,\forall P, Q. \\
& & \quad \triangleright(P \cong Q) \vdash (f\,P) \cong (f\,Q) \\
& = & \lambda f.\,\mathsf{contractive}_{\mathsf{HO}}\ \mathtt{unit}\ (\lambda g.\,\lambda(t : \mathtt{unit}).\,f\ (g\ t)) \\
\\
\mu P & = & \mu_{\mathsf{HO}}\ \mathtt{unit}\ (\lambda P'.\,\lambda t : \mathtt{unit}.\,P(P'\ t))\ \mathtt{tt}
\end{array}
$$

Figure 7.12: Recursion

If P is not tight then the meaning of $\mathsf{tightly}\ P$ depends on the exact nature of P, but is likely to be equal to **false** (which is very tight).

7.3.9 Higher-order recursion

Natural invariants frequently have a recursive structure; for example, the invariant required for a linked list where each lock guards another

linked list (except for the last lock, which guards nil). We also used the recursion operator μ to define the example program's resource invariant in section 4.9.1, where it helped provide a natural way to satisfy the precondition of the unlock CSL rule. In our work we will extend the normal recursion operator μ to a higher-order version μ_{HO}, which will be used to define our Hoare tuple in section 10.1.3.

The indexed models of Appel et al. [AM01] and Ahmed et al. [AAV02, AAV03, Ahm04] and the modal model of Appel et al. [AMRV07] all supported recursion. We employ a hybrid approach: we follow Appel et al. [AMRV07] by defining contractiveness using the approximately operator, but our defintions of the recursion operator and our proofs of the fold-unfold rules are closer to those in Appel et al. [AM01]. Our defintions are given in figure 7.12.

Our semantic recursion operator is more powerful than the one defined in the indexed and modal models because it is higher order. A standard first-order recursion operator μ has the type

$$\mu \ : \ (\mathsf{predicate} \to \mathsf{predicate}) \ \to \ \mathsf{predicate}.$$

In contrast, our semantic higher-order recursion operator μ_{HO} is parameterized by an extra type parameter α:

$$\mu_{\mathsf{HO}} \ : \ \Lambda\alpha.\, ((\alpha \to \mathsf{predicate}) \to (\alpha \to \mathsf{predicate})) \to \alpha \to \mathsf{predicate}.$$

This more powerful recursion operator is used to define the Hoare tu-

ple in chapter 10. Because the higher-order recursion operator μ_{HO} is strictly more powerful than the first-order recursion operator μ, we can define μ in terms of μ_{HO}. Still, in many situations, such as defining the resource invariant in section 4.9.1, μ is powerful enough, and since it is simpler to use we include it as well.

The rest of the proof does not see the definition of μ_{HO}, which is a generalization of the first-order one found in Appel et al. [AM01]. Instead, the rest of the proof uses the following fold-unfold rule:

$$\forall f.\ (\mathsf{contractive}_{\mathsf{HO}}\ f)\ \rightarrow\ \mu_{\mathsf{HO}}\ f = f\ (\mu_{\mathsf{HO}}\ f).$$

Contractiveness is a requirement for defining recursive equations in many contexts; here we build on the elegant definitions by Appel et al. [AMRV07]. A function f is first-order contractive if for all predicates P and Q that are approximately equivalent, $f\,P$ is approximately equivalent to $f\,Q$. Higher-order contractiveness is a generalization of this idea, where we universally quantify over the additional parameter $a : \alpha$. As indicated by the second definition of $\mathsf{contractive}$ in figure 7.12, it is also possible to define first-order contractiveness in terms of higher-order contractiveness.

The proof of the higher-order fold-unfold rule roughly follows the one given in [AM01], although the induction hypotheses are more complicated because they support the higher-order construction. From the proof of the higher-order fold-unfold rule, it is trivial to prove the first-

order rule

$$\forall f.\ (\mathsf{contractive}\ f)\ \rightarrow\ \mu\ f = f\ (\mu f),$$

given the second definition of first-order contractiveness.

7.3.10 YES inversion

We have defined and explained our modal substructural logic, so we can finally characterize inverting YES. Here we show that

$$\begin{aligned}\vdash\ & \big(l \circledS (k, \pi, \vec{P})\ \cong\ l' \circledS (k', \pi', \vec{P'})\big)\ \cong \\ & \big([l = l']_{\mathsf{Coq}}\ \wedge\ [k = k']_{\mathsf{Coq}}\ \wedge\ [\pi = \pi']_{\mathsf{Coq}}\ \wedge\ \forall j.\ \triangleright (\vec{P}_j \cong \vec{P'}_j)\big).\end{aligned}$$

First, we will state a more elegant characterization for the stratify operator than could be given in section 7.2.4. In that section, we observed that

$$\mathsf{stratify}\ k\ P\ =\ \mathsf{stratify}\ k\ Q$$

is equivalent to

$$\text{(C1)}\qquad \forall j < k, \phi_j, \rho, m.\quad P(\rho,\ \{j,\ \phi_j\},\ m) \leftrightarrow Q(\rho,\ \{j,\ \phi_j\},\ m),$$

which is in turn equivalent to

$$\begin{aligned}\text{(C2)}\qquad & \forall \phi, n, \phi', \rho, m. \\ & (\mathsf{level}\ \phi = k)\ \rightarrow\ (\mathsf{age}\ (n+1)\ \phi\ =\ \mathsf{Some}\ \phi')\ \rightarrow \\ & (P(\rho, \phi', m) \leftrightarrow Q(\rho, \phi', m)).\end{aligned}$$

Now we are in a position to state a characterization related to the previous one using the operators of the logic:

$$\text{(C3)} \qquad \forall \phi, \rho, m.\ \mathsf{level}\ \phi = k \ \rightarrow\ (\rho, \phi, m) \models \triangleright\, (P \cong Q).$$

One of the advantages of giving this characterization is that it demonstrates the power of expressing ideas in the modal logic, since it is clearly more compact than the previous one. It is still not quite ideal, however, since it still requires exposing the private level operator.

Recall from figure 7.7 that the $\wr$ operator calls the YES pseudoconstructor, passing the ϕ in the world it is passed as its first argument. YES, in turn, then calls stratify to produce invariants stratified to level ϕ. In other words,

$$(\rho, \phi, m) \models l \wr (k, \pi, \vec{P})$$

is calling stratify on the elements of the list $\vec{P}$ to produce a list of stratified invariants of level ϕ.

The way we express inversion in the logic is to determine the necessary and sufficient conditions for concluding

$$w \models l \wr (k, \pi, \vec{P}) \ \cong\ l' \wr (k', \pi', \vec{P'}).$$

Putting together the previous observations about characterizing stratify and the understanding of the amount of stratification done by the $\wr$

operator, we are able to show that

$$w \models l \circledS (k, \pi, \vec{P}) \;\cong\; l' \circledS (k', \pi', \vec{P'}).$$

if and only if

$$l = l' \;\wedge\; k = k' \;\wedge\; \pi = \pi' \;\wedge\; \forall j.\; w \models \rhd (\vec{P}_j \cong \vec{P'}_j).$$

This can be reformulated completely in our logic as

$$w \models [l = l']_{\mathsf{Coq}} \;\wedge\; [k = k']_{\mathsf{Coq}} \;\wedge\; [\pi = \pi']_{\mathsf{Coq}} \;\wedge\; \forall j.\; \rhd (\vec{P}_j \cong \vec{P'}_j).$$

Now we have expressed both properties in terms of w. We can combine them using the $\cong$ operator as follows:

$$\begin{aligned} w \;\models\; & \big(l \circledS (k, \pi, \vec{P}) \;\cong\; l' \circledS (k', \pi', \vec{P'})\big) \;\cong \\ & \big([l = l']_{\mathsf{Coq}} \;\wedge\; [k = k']_{\mathsf{Coq}} \;\wedge\; [\pi = \pi']_{\mathsf{Coq}} \;\wedge\; \forall j.\; \rhd (\vec{P}_j \cong \vec{P'}_j)\big), \end{aligned}$$

which leads naturally to the following final characterization:

$$\begin{aligned} \vdash\; & \big(l \circledS (k, \pi, \vec{P}) \;\cong\; l' \circledS (k', \pi', \vec{P'})\big) \;\cong \\ & \big([l = l']_{\mathsf{Coq}} \;\wedge\; [k = k']_{\mathsf{Coq}} \;\wedge\; [\pi = \pi']_{\mathsf{Coq}} \;\wedge\; \forall j.\; \rhd (\vec{P}_j \cong \vec{P'}_j)\big). \end{aligned}$$

Expressed informally, $l \circledS (k, \pi, \vec{P}) = l \circledS (k, \pi, \vec{P'})$ if and only if the predicate lists $\vec{P}$ and $\vec{P'}$ are approximately equal.

7.4 Conclusions

We have now provided a sound definition for resource and rmap using a stratification technique. The stratified model is hidden behind a cleaner dependent model, which is in turn hidden behind a modal substructural logic. By reasoning about the underlying models using this logic we can express properties cleanly and reason about them more simply than if we manipulated the models directly.

This logic forms the model for the assertions of CSL, which can then utilize the logic's full expressive power. The model for the Hoare tuple itself will be deferred until chapter 10, since it depends first on the concurrent operational semantics presented in chapter 8 and oracle semantics presented in chapter 9.

Chapter 8

Concurrent Operational Semantics

In chapter 3 we introduced Concurrent C minor, and in chapter 5 gave it a formal erased concurrent operational semantics. The purpose of this chapter is to give it an q*unerased* concurrent operational semantics[1]. The concurrent operational semantics does additional bookkeeping and is therefore easy to use for metatheoretical reasoning about concurrent features such as locking a lock. The concurrent operational semantics is not easy to use for metatheoretical reasoning about the equential features of Concurrent C minor. In chapter 9, we define an oracular semantics for Concurrent C minor that will be allow for straightforward metatheoretical reasoning for the sequential features of Concurrent C minor.

[1]Normally we leave off the word unerased and just say the concurrent operational semantics.

In section 8.1 we outline the parts that make up the concurrent operational semantics. In section 8.2 we define the sequential submachine, which executes sequential code. In section 8.3 we define the state of a concurrent computation, and in section 8.4 we give a set of consistency properties for a concurrent machine state. In section 8.5 we define the concurrent step relation. Finally, in section in section 8.6, we argue why our concurrent operational semantics is a reasonable abstraction of a real machine by showing that it is a conservative approximation to the erased concurrent operational semantics defined in 5[2].

8.1 Architecture

The concurrent operational semantics has several distinct parts. The first, called the "sequential submachine", executes all of the statements that do not depend on other threads, such as `call`, store, and `loop`. We isolate all the properties of the sequential step relation on which we rely using the interface discussed in section 6.3.1. The sequential submachine is thus "resource map aware", meaning that it gets stuck if it attempts to access memory without the correct permission. In this way we support half of our modularity principle by hiding the complexities of sequential control- and data-flow from concurrent metatheoretical reasoning; as explained in chapter 9, we hide the complexities of concurrent computation from sequential metatheoretical reasoning using an oracle semantics.

[2]Portions of this chapter have been published before as [HAZ08a] and [HAZ08b].

We combine the local states of many sequential submachines to get threads and then add a schedule, lock pool, alloc pool, function pool, and memory to get a concurrent machine state. Our concurrent state has a complex set of consistency requirements that ensure that it is well-formed.

We define a concurrent step relation that transforms concurrent states into concurrent states in the context of a program Ψ. When a thread wishes to execute a sequential instruction, the concurrent step relation uses the sequential submachine. Fully-concurrent instructions such as lock are handled by the concurrent step relation directly. Our concurrent step relation has a number of unusual features, including determinism, coroutine interleaving, and nonconstructive semantics. We conclude by arguing why these features are reasonable abstractions for reasoning about real machines.

8.2 Sequential submachine

The sequential submachine is the part of the concurrent operational semantics that knows how to execute sequential instructions. The two *pseudoconcurrent* statements, make_lock and free_lock do not interact with the other threads and so are executed by the sequential submachine as well. At the fully concurrent statements—lock, unlock, and fork—the sequential submachine gets stuck.

We build it by extending the core semantics of C minor with with

$$\text{sstep-update}\ \frac{(\rho,\phi,m) \models \exists v.\ v_1 \overset{\blacktriangle}{\mapsto} v \qquad m' = [v_1 \mapsto v_2]\ m}{\Psi \vdash ((\rho,\phi,m), [v_1]{:=}v_2 \cdot \kappa) \longmapsto ((\rho,\phi,m'),\kappa)}$$

$$\text{sstep-call}\ \frac{\mathsf{age}\ 1\ \phi = \mathsf{Some}\ \phi'}{\Psi \vdash ((\rho,\phi,m), \mathsf{call}\ f\ \vec{v} \cdot \kappa) \longmapsto ((\rho,\phi',m), (\Psi(f)\ \vec{v}) \cdot \kappa)}$$

$$\text{sstep-seq}\ \frac{}{\Psi \vdash ((\rho,\phi,m), s_1; s_2 \cdot \kappa) \longmapsto ((\rho,\phi,m), s_1 \cdot s_2 \cdot \kappa)}$$

Figure 8.1: Simplified subset of sequential step relation

an extension that knows how to execute the pseudoconcurrent instructions. When we defined the erased machine in chapter 5 we provided the erased semantics for three sequential C minor statements in figure 5.1. For comparison, we give the unerased semantics for those core sequential instructions in figure 8.1. The semantics of these statements have been significantly simplified in the presentation due to the removal of nonlocal exits, stack allocated variables, and function return values.

The unerased sequential semantics are very similar to the erased sequential semantics. We have added a resource map ϕ to the world w. The sequence rule does not use it, but both the call rule and the assignment rule do. All the call rule does to ϕ is age it once; this is required for the semantic model of the Hoare tuple for reasons explained in section 10.2.2. The assignment rule uses ϕ to check that the location v_1 is writable by requiring

$$(\rho,\phi,m) \models \exists v.\ v_1 \overset{\blacktriangle}{\mapsto} v,$$

that is, that the memory location v_1 be fully owned by the resource map ϕ. It is easy to show that the unerased sequential semantics is a conservative approximation to the erased sequential semantics.

Lemma 8.1. If $\Psi \vdash (\rho, \phi, m) \longmapsto (\rho', \phi', m')$, then $\Psi \vdash (\rho, m) \overset{\mathsf{e}}{\longmapsto} (\rho', m')$.

Proof. Each case in the unerased sequential relation in figure 8.1 has a corrosponding case in the erased sequential relation in figure 5.1 with a subset of the premises. □

Recall from section 6.6 that to extend a core semantics one defines an *oracle* type and then constructs a *consult relation*

$$\mathsf{consult} \ : \ \mathsf{program} \times \mathsf{oracle} \times \mathsf{state} \ \rightarrow \ \mathsf{option}(\mathsf{oracle} \times \mathsf{state}),$$

which implements the extended statements. Recall that a state σ is a pair of a world w and a control κ. A world is a level-independent tuple of locals ρ, resource map ϕ, and memory m; it also contains a map from global names to addresses, but this has been elided from the presentation. A control contains the level-specific portions of the state, primarily program syntax.

For the sequential submachine, the oracle type will be `unit` because the make_lock and free_lock statements do not need any auxilary state. Nontrivial oracles are needed only for interpreting the meaning of the fully-concurrent instructions lock, unlock, and fork, which the sequential submachine does not attempt to implement.

$$\frac{(\rho, \phi, m) \models v \overset{\blacktriangle}{\mapsto} 0 * \mathsf{exactly}\ \phi_{\mathsf{core}} \qquad (\rho, \phi', m) \models v \overset{\blacktriangle}{\rightsquigarrow} R * \mathsf{hold}\ v\, R * \mathsf{exactly}\ \phi_{\mathsf{core}}}{\mathsf{consult}(\Psi, \mathtt{tt}, ((\rho, \phi, m), \mathsf{make\_lock}\, v\, R \cdot \kappa)) = \mathsf{Some}(\mathtt{tt}, ((\rho, \phi', m), \kappa))}$$

$$\frac{(\rho, \phi, m) \models v \overset{\blacktriangle}{\rightsquigarrow} R * \mathsf{hold}\ v\, R * \mathsf{exactly}\ \phi_{\mathsf{core}} \qquad (\rho, \phi', m) \models v \overset{\blacktriangle}{\mapsto} 0 * \mathsf{exactly}\ \phi_{\mathsf{core}}}{\mathsf{consult}\ (\Psi, \mathtt{tt}, ((\rho, \phi, m), \mathsf{free\_lock}\, v \cdot \kappa)) = \mathsf{Some}\ (\mathtt{tt}, ((\rho, \phi', m), \kappa))}$$

Figure 8.2: The consult relation of the sequential submachine

We define consult in figure 8.2. To execute $\mathsf{make\_lock}\, v\, R$, the machine ensures that the location is fully-owned data containing a zero, and updates the resource map to treat the location as a lock with invariant R. The lock is created with 100% visibility and is also created held. Since a lock at location l is "locked" if memory at l is zero, the lock is created locked, and the make_lock statement only changes the resource map. We use the exactly predicate to make sure that the world does not change in any other way.

To execute $\mathsf{free\_lock}\, v$, the machine reverses make_lock. First it verifies that the lock is entirely owned and held, and then updates the resource map to show that the location is fully-owned data containing a zero. Since the lock is held, the memory will contain a zero at v.

At any other instruction, consult is undefined. This is different from relating a state to None, which is reserved for commands that do not terminate, *e.g.*, a lock that deadlocks. The sequential submachine either

completes the execution of an extended statement or gets stuck.

Using the definitions given in section 6.6 we can build a step relation $\overset{\mathsf{ss}}{\mapsto}$ for the sequential submachine using the core step relation $\longmapsto$ defined in figure 8.1 and the consult relation defined in figure 8.2. This step relation will be a conservative approximation to the erased sequential step relation $\overset{\mathsf{e}}{\mapsto}$.

> **Lemma 8.2**. If $\Psi \vdash (\rho, \phi, m) \overset{\mathsf{ss}}{\mapsto} (\rho', \phi', m')$, then $\Psi \vdash (\rho, m) \overset{\mathsf{e}}{\mapsto} (\rho', m')$.
>
> **Proof**. There are three cases in figure 6.10. For the first case, Core Step, we know $\Psi \vdash (\rho, \phi, m) \longmapsto (\rho', \phi', m')$ and by lemma 8.1 we know $\Psi \vdash (\rho, m) \overset{\mathsf{e}}{\mapsto} (\rho', m')$. The second case, Oracle Step, means that our consult has succeeded, in which case there are two subcases: one for make_lock and one for free_lock. In both cases we do not change ρ or m and transition to control κ. This is exactly what is done for these statements in the erased sequential step given in figure 5.1. The third case, Oracle Diverges, is impossible because our consult relation never returns None. □

8.3 Machine state

A *concurrent machine state* S is a tuple

$$S = (\mho, \vec{\theta}, \mathcal{L}, \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m),$$

where $\mho$ is a *schedule*, $\vec{\theta}$ is a *thread list*, $\mathcal{L}$ is a *lock pool*, ϕ_{mp} is a *memory pool*, ϕ_{fp} is a *function pool*, and m is a memory. The memory is exactly the same as in the sequential semantics. Each of the other components will be explained in sections 8.3.1–8.3.4 below. A concurrent machine state also comes with a set of consistency requirements, as explained in section 8.4. For this presentation, any concurrent machine state should be considered consistent.

8.3.1 Schedule

A schedule $\mho$ is a **finite** list of natural numbers, which act as thread-IDs. The number at the head of the schedule tells the concurrent operational semantics which thread to execute next. When the schedule runs out, the concurrent machine safely halts computation. Our reason for quantifing over finite schedules was explained in section 5.2.

8.3.2 Threads

The purpose of a concurrent machine is to execute multiple threads of control. A *thread* θ is a tuple $(\rho, \phi, \hat{\kappa})$, where ρ are local variables, ϕ is a resource map, and $\hat{\kappa}$ is a *concurrent control*, defined in section 5.2. Since $\vec{\theta}$ denotes a list of threads, we indicate the ith thread by $\vec{\theta}_i$.

Recall that a world w contains two "local" components, local variables ρ and resource map ϕ, and one "global" component, memory m. Each thread gets its own private version of local variables ρ and re-

source map ϕ; in contrast, m is not part of a thread because all threads must use the same memory.

8.3.3 Lock pool

A lock pool $\mathcal{L}$ is a partial function from the addresses of unlocked locks to resource maps. In our setting, every resource must be owned by somebody at all times. Normally, resources are owned by the threads. However, when a lock is unlocked, a thread gives up the resources (*i.e.*, the resource map) associated with the lock. When a thread is unlocking a lock we do not wish to make the thread wait for the next thread to lock the lock. Therefore, we need a place to hold the resource map associated with the lock; this place is the lock pool $\mathcal{L}$. When the next thread grabs the lock, we transfer the resources from the lock pool to the next thread.

The look pool not simply a resource map. Instead, it is a partial function from addresses to resource maps because this structure greatly simplifies the proof by making it trivial to show that all the resource maps associated with unlocked locks are disjoint.

8.3.4 Alloc and function pools

The alloc pool ϕ_{mp} and function pool ϕ_{fp} are resource maps that own of two kinds of global resources. Each thread is given access to these global resources as it executes and relinquishes them when it yields to the next thread.

The alloc pool is the owner of the unallocated but allocatable memory, and is used to allocate activation records on the stack at function call. In the CompCert memory model there is an unlimited amount of memory available for the stack[3].

The function pool is the owner of all of the functions. In other words,

$$\forall \rho, m.\ (\rho, \phi_{\mathsf{fp}}, m) \models f_1 : \{P\}\{Q\}\ *\ \ldots\ *\ f_n : \{P\}\{Q\},$$

and all of the other resource maps in the system lack functions. The function pool allows all of the threads to have access to all of the functions. Passing the function pool around from thread to thread is not particularly beautiful, but it does ensure that the functions are available for all threads in a standard way. Developing an elegant way to guarantee global function availablity is suprisingly difficult, and thus is left for future work.

8.4 Consistent machines

A concurrent machine state carries with it a set of consistency properties that ensure that it is well-formed. In Coq we ensure the consistency

[3]In a sequential program the alloc pool is a reasonable abstraction, as the operating system will terminate any program whose stack space grows beyond the available memory in the machine, and the CompCert compiler only makes partial correctness guarantees. In a concurrent program the situation is more complicated, since additional checks must be inserted in the function call to make sure that the thread is not about to exceed its stack space. In a high-level language like Concurrent C minor, checks are inserted by the compiler when it compiles a function call into lower-level languages; however this abstraction may not allow us to adapt our concurrency system for lower compiler levels, an open problem for future work.

of concurrent machine states with a dependently-typed record.

8.4.1 Existence of total resource map

The first property is the existence of a *total resource map* ϕ_{T} that is the join of all of the resource maps in the threads $\vec{\theta}$, all the resource maps in the lock pool $\mathcal{L}$, the alloc pool ϕ_{mp}, and the function pool ϕ_{fp}:

$$\phi_{\mathsf{T}} = (\bigoplus_{\phi_i \in \vec{\theta_i}} \phi_i) \oplus (\bigoplus_{\phi_i \in \mathcal{L}} \phi_i) \oplus \phi_{\mathsf{mp}} \oplus \phi_{\mathsf{fp}}.$$

In other words, the individual resource maps combine into a cohesive whole ϕ_{T}. This guarantees, for example, that two distinct threads cannot have full ownership of a location at the same time. Thus, if a thread can write to a memory location, then no other thread can read from it. The remaining properties all use ϕ_{T}.

8.4.2 Sufficient schedule

The second property is that the schedule will outlast the approximation level of ϕ_{T}, *i.e.*,

$$\mathsf{level}\ \phi_{\mathsf{T}} < \mathsf{length}\ \mho.$$

We will be trivially safe if we reach approximation level 0.

8.4.3 Only waiting on locks

The third property is that if a thread's concurrent control is $\mathsf{Klock}\, v\, \kappa$, then v is a lock in ϕ_{T}.

We use a function from resources to resource kinds:

$$\mathsf{rkind} \;:\; \mathsf{resource} \to \mathsf{option}(\mathsf{kind}),$$

which is defined by the stratified model explained in chapter 7 as an opaque definition, as follows:

$$\begin{aligned}\mathsf{rkind} \;\triangleq\; & \lambda\{n, \xi_n\}.\ \mathsf{match}\ \xi_n\ \mathsf{with} \\ & \quad \mathsf{NO}_n \;\Rightarrow\; \mathsf{None} \\ & \quad \mathsf{YES}_n\ k\ \pi\ \vec{P} \Rightarrow\; \mathsf{Some}\ k \\ & \mathsf{end}.\end{aligned}$$

We use rkind instead of the YES pseudoconstructor because it lets us ignore the pre- and postconditions of the function and so provide a cleaner characterization. The advantage here is relatively minor, but as explained in section 8.4.6, rkind also interacts better with age in the presence of implication, and there it is vital.

Using rkind it is easy to define the property that threads can only wait on locks, as opposed to, *e.g.*, regular data:

$$\mathsf{rkind}(\phi_{\mathsf{T}}\ @\ v) = \mathsf{Some}\ \mathsf{kLK}.$$

8.4.4 Well-formed alloc pool

The fourth property is that the alloc pool ϕ_{mp} is well-formed. We enforce well-formedness by defining a predicate $\mathsf{allocpool}$, which abstracts the nature of the CompCert memory model, and express the property as

$$\forall \rho.\ (\rho, \phi_{\mathsf{T}}, m) \models \mathsf{allocpool} * \mathbf{true},$$

where we use the memory m in the concurrent machine state.

8.4.5 Well-formed function pool

The fifth property is that the function pool ϕ_{fp} contains all of the functions. First we split off the ϕ_{fp} from ϕ_{T}, as follows:

$$\phi_{\mathsf{fp}} \oplus \phi_{\mathsf{nf}} = \phi_{\mathsf{T}}.$$

The resource mape ϕ_{nf} is everything in ϕ_{T} that is not a function.

Using rkind it is straightforward to express property P1, that ϕ_{nf} does not contain any functions:

$$\text{(P1)} \quad \forall l.\ \neg\,(\mathsf{rkind}(\phi_{\mathsf{nf}} \mathbin{@} l) = \mathsf{Some\ kFUN}).$$

We can also use rkind to express property P2, that ϕ_{fp} contains only

things of kind kFUN:

$$\begin{aligned}(\text{P2}) \quad \forall l. \quad & (\mathsf{rkind}(\phi_{\mathsf{fp}} \mathbin{@} l) = \mathsf{Some\ kFUN}) \vee \\ & (\mathsf{rkind}(\phi_{\mathsf{fp}} \mathbin{@} l) = \mathsf{None}).\end{aligned}$$

The fifth property is just the conjunction of P1 and P2.

8.4.6 Well-formed lock pool

The sixth property is a series of requirements on the lock pool. This turns out to be a bit trickier to define than expected. The most critical property is that we would like something like the following, which we call $\text{P1}(\phi_{\mathsf{T}}, \mathcal{L})$:

$$\begin{aligned}\text{P1}(\phi_{\mathsf{T}}, \mathcal{L}) \quad = \quad \forall l, \pi, R. \quad & \phi_{\mathsf{T}} \mathbin{@} l = \mathsf{YES}\ \phi_{\mathsf{T}}\ \mathsf{kLK}\ \pi\ (R :: \mathtt{nil}) \ \rightarrow \\ & m(l) = \mathtt{lock\_unlocked} \ \rightarrow \\ & \forall \rho.\ (\rho, \mathcal{L}(l), m) \models R.\end{aligned}$$

In other words, for every lock in the total resource map ϕ_{T}, if that lock is unlocked (*i.e.*, if the memory at the lock location contains the value `lock_unlocked`, which is 1), then the lock invariant holds on the associated resource map in the lock pool $\mathcal{L}$.

The problem is that we need this property to be true even if the total resource map ϕ_{T} is aged to some ϕ'_{T}. Since ϕ_{T} includes all of the worlds in $\mathcal{L}$, we would also have to age those worlds with the age_lpool operation, which extends age to lock pools by aging the range pointwise.

Since $\mathcal{L}$ is contained in ϕ_T by property 1, $\mathcal{L}'$ is contained in ϕ'_T. In other words, we need the following:

$$\begin{aligned}
&\mathrm{P1}(\phi_\mathsf{T}, \mathcal{L}) \ \rightarrow \\
&\mathsf{age}\ n\ \phi_\mathsf{T} = \mathsf{Some}\ \phi'_\mathsf{T} \ \rightarrow \\
&\mathsf{age_lpool}\ n\ \mathcal{L} = \mathsf{Some}\ \mathcal{L}' \ \rightarrow \\
&\mathrm{P1}(\phi'_\mathsf{T}, \mathcal{L}').
\end{aligned}$$

Unfortunately, as P1 is stated, this is simply not true.

The first problem is the existence of wild terms in the model. If $w \models R$, it is not necessarily the case that $w' \models R$ for some more approximate w'. In chapter 7 we introduced the concept "necessary" to handle this issue. In chapter 4 we introduced the tightly operator, and said that the precondition for unlocking a lock with invariant R was tightly R. In section 7.3.8, we show that tightly is defined with the $\Box$ operator, and show that therefore tightly R is necessary. Therefore we attempt to fix the problem with property $\mathrm{P2}(\phi_\mathsf{T}, \mathcal{L})$:

$$\begin{aligned}
\mathrm{P2}(\phi_\mathsf{T}, \mathcal{L}) \ = \ \forall l, \pi, R. \ \ & \phi_\mathsf{T} \ @\ l = \mathsf{YES}\ \phi_\mathsf{T}\ \mathsf{kLK}\ \pi\ (R :: \mathtt{nil}) \ \rightarrow \\
& m(l) = \mathtt{lock_unlocked} \ \rightarrow \\
& \forall \rho.\ (\rho, \mathcal{L}(l), m) \models \mathsf{tightly}\ R.
\end{aligned}$$

Unfortunately, there is still a problem with this definition that prevents it from ageing gracefully.

Recall from section 7.3.7 that because implication is contravariant in its antecedent, it does not play nicely with approximation. Unfor-

tunately P1 and P2 use the YES pseudoconstructor in an antecedent, which in turn uses stratify, which approximates R. The solution is to use rkind in the antecedent, since it does not require R, and to quantify over the resource invariant existentially instead of universally, leading to property $\mathrm{P3}(\phi_\mathsf{T}, \mathcal{L})$:

$$\begin{aligned}\mathrm{P3}(\phi_\mathsf{T}, \mathcal{L}) \quad = \quad \forall l.\ \ & \mathsf{rkind}(\phi_\mathsf{T} \mathbin{@} l) = \mathsf{Some\ kLK} \ \rightarrow \\ & m(l) = \texttt{lock\_unlocked} \ \rightarrow \\ & \exists \pi, R.\ \mathsf{YES}\ \phi_\mathsf{T}\ \mathsf{kLK}\ \pi\ (R :: \texttt{nil}) \wedge \\ & \qquad \forall \rho.\ (\rho, \mathcal{L}(l), m) \models \mathsf{tightly}\ R.\end{aligned}$$

P3 does behave gracefully under ageing, and, informally, means almost the same thing as P1: In other words, for every lock in the total resource map ϕ_T, if that lock is unlocked (*i.e.*, if the memory at the lock location contains the value `lock_unlocked`, which is 1), then the lock invariant holds on associated resource map in the lock pool $\mathcal{L}$.

In the actual Coq development, we use a weaker variant of P3, which we call P3′, and in which we only guarantee $\triangleright\, \mathsf{tightly}\ R$ instead of $\mathsf{tightly}\ R$:

$$\begin{aligned}\mathrm{P3}'(\phi_\mathsf{T}, \mathcal{L}) \quad = \quad \forall l.\ \ & \mathsf{rkind}(\phi_\mathsf{T} \mathbin{@} l) = \mathsf{Some\ kLK} \ \rightarrow \\ & m(l) = \texttt{lock\_unlocked} \ \rightarrow \\ & \exists \pi, R.\ \mathsf{YES}\ \phi_\mathsf{T}\ \mathsf{kLK}\ \pi\ (R :: \texttt{nil}) \wedge \\ & \qquad \forall \rho.\ (\rho, \mathcal{L}(l), m) \models \triangleright\, \mathsf{tightly}\ R.\end{aligned}$$

Since for all P, $\triangleright\, P$ is strictly weaker than $\Box\, P$, P3′ is a weaker guar-

antee than P3, meaning that it is easier to introduce but harder to use. In future work we plan to move the Coq development from P3′ to P3.

In addition to P3′, we require that for each lock location l, memory at l is either `lock_unlocked` (1) or `lock_locked` (0); we also require that if a lock is locked, then l not be in the domain of the partial function $\mathcal{L}$.

8.5 Concurrent step relation

The full concurrent small-step relation is given in figures 8.3 and 8.4. Figure 8.3 contains the portion of the step relation that is responsible for executing sequential statements.

The key sequential rule is cstep-seq, which uses the core semantics of the sequential submachine to perform a sequential step (recall from section 6.6 how we construct the $\mapsto$ relation from $\mathsf{consult}$). The head of the schedule is i, meaning that the next thread to execute is $\vec{\theta}_i$, which is equal to $(\rho, \phi, \mathsf{Krun}\,\kappa)$.

First the alloc and function pools are joined to ϕ to make ϕ_{s}, and then we build a world $(\rho, \phi_{\mathsf{s}}, m)$ and step in the sequential submachine with that world and the control κ to a subsequent world $(\rho', \phi'_{\mathsf{s}}, m')$ and control κ'. Since the sequential submachine's oracle has type `unit` we use the value `tt` as a parameter to the step relation.

After taking a step in the submachine, we split out new alloc and function pools from ϕ'_{s}. The submachine is allowed to age the resource

$$\text{cstep-seq}\ \frac{\begin{array}{c}\vec{\theta}_i = (\rho, \phi, \mathsf{Krun}\,\kappa) \\ \phi_{\mathsf{s}} = \phi \oplus \phi_{\mathsf{mp}} \oplus \phi_{\mathsf{fp}} \\ \Psi \vdash (\mathtt{tt}, ((\rho, \phi_{\mathsf{s}}, m), \kappa)) \overset{\mathsf{ss}}{\longmapsto} (\mathtt{tt}, ((\rho', \phi'_{\mathsf{s}}, m'), \kappa')) \\ \phi'_{\mathsf{s}} = \phi' \oplus \phi'_{\mathsf{mp}} \oplus \phi'_{\mathsf{fp}} \\ (\rho', \phi'_{\mathsf{mp}}, m') \models \mathsf{allocpool} \\ \mathsf{age}\ n\ \phi_{\mathsf{fp}} = \mathsf{Some}\ \phi'_{\mathsf{fp}} \\ \mathsf{age\_lpool}\ n\ \mathcal{L} = \mathsf{Some}\ \mathcal{L}' \\ \mathsf{age\_list}\ n\ \vec{\theta} = \mathsf{Some}\ \vec{\theta}' \\ \vec{\theta}'' = [i \mapsto (\rho', \phi', \mathsf{Krun}\,\kappa')]\ \vec{\theta}'\end{array}}{\Psi \vdash (i :: \mho, \vec{\theta}, \mathcal{L}, \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m) \implies (i :: \mho, \vec{\theta}'', \mathcal{L}', \phi'_{\mathsf{mp}}, \phi'_{\mathsf{fp}}, m')}$$

$$\text{cswitch}\ \frac{\begin{array}{c}\mathsf{age\_lpool}\ 1\ \mathcal{L} = \mathsf{Some}\ \mathcal{L}' \\ \mathsf{age\_list}\ 1\ \vec{\theta} = \mathsf{Some}\ \vec{\theta}' \\ \mathsf{age}\ 1\ \phi_{\mathsf{mp}} = \mathsf{Some}\ \phi'_{\mathsf{mp}} \\ \mathsf{age}\ 1\ \phi_{\mathsf{fp}} = \mathsf{Some}\ \phi'_{\mathsf{fp}}\end{array}}{\mathsf{CSwitch}\ (i :: \mho, \vec{\theta}, \mathcal{L}, \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m) = (\mho, \vec{\theta}', \mathcal{L}', \phi'_{\mathsf{mp}}, \phi'_{\mathsf{fp}}, m)}$$

$$\text{cstep-texit}\ \frac{\begin{array}{c}\vec{\theta}_i = (\rho, \phi, \mathsf{Krun}\,(\mathsf{Kstop})) \\ \mathsf{CSwitch}\ (i :: \mho, \vec{\theta}, \mathcal{L}, \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m) = S\end{array}}{\Psi \vdash (i :: \mho, \vec{\theta}, \mathcal{L}, \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m) \implies S}$$

Figure 8.3: Sequential steps in the concurrent step relation

map to make it more approximate if it wishes, and so we allow the function pool, lock pool, and other threads to age as well so that all resource maps will have the same level and therefore will be able to join together into a total resource map.

One key point is that the schedule does not change during cstep-seq. Thus the next instruction to be executed will be from the same thread—that is, the concurrent step relation does not context switch when executing normal sequential instructions. In section 5.3 we dis-

$$\text{cstep-prelock}\ \frac{\begin{array}{c}\vec{\theta}_i = (\rho, \phi, \mathsf{Krun}\,\mathsf{lock}\,v \cdot \kappa) \\ (\rho, \phi, m) \models \mathbf{true} * v \overset{\pi}{\rightsquigarrow} R \\ \vec{\theta'} = [i \mapsto (\rho, \phi, \mathsf{Klock}\,v\,\kappa)]\ \vec{\theta} \\ \mathsf{CSwitch}\ (i :: \mathfrak{V}, \vec{\theta'}, \mathcal{L}, \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m) = S\end{array}}{\Psi \vdash (i :: \mathfrak{V}, \vec{\theta}, \mathcal{L}, \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m) \implies S}$$

$$\text{cstep-nolock}\ \frac{\begin{array}{c}\vec{\theta}_i = (\rho, \phi, \mathsf{Klock}\,v\,\kappa) \qquad m(v) = 0 \\ \mathsf{CSwitch}\ (i :: \mathfrak{V}, \vec{\theta}, \mathcal{L}, \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m) = S\end{array}}{\Psi \vdash (i :: \mathfrak{V}, \vec{\theta}, \mathcal{L}, \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m) \implies S}$$

$$\text{cstep-lock}\ \frac{\begin{array}{c}\vec{\theta}_i = (\rho, \phi, \mathsf{Klock}\,v\,\kappa) \\ m(v) = 1 \qquad m' = [v \mapsto 0]\ m \qquad \mathcal{L} = v : \phi_{\mathsf{lock}}, \mathcal{L}' \\ \vec{\theta'} = [i \mapsto (\rho, \phi \oplus \phi_{\mathsf{lock}}, \mathsf{Krun}\,\kappa)]\ \vec{\theta}\end{array}}{\Psi \vdash (i :: \mathfrak{V}, \vec{\theta}, \mathcal{L}, \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m) \Longrightarrow (i :: \mathfrak{V}, \vec{\theta'}, \mathcal{L}', \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m')}$$

$$\text{cstep-unlock}\ \frac{\begin{array}{c}\vec{\theta}_i = (\rho, \phi, \mathsf{Krun}\,\mathsf{unlock}\,v \cdot \kappa) \\ m(v) = 0 \qquad m' = [v \mapsto 1]\ m \qquad \phi = \phi' \oplus \phi_{\mathsf{lock}} \\ (\rho, \phi, m) \models (\mathbf{true} * \mathsf{hold}\ v\,P) \\ (\rho, \phi_{\mathsf{lock}}, m) \models \mathsf{tightly}\ P \\ \mathcal{L}' = v : w_{\mathsf{lock}}, \mathcal{L} \qquad \vec{\theta'} = [i \mapsto (\rho, \phi', \mathsf{Krun}\,\kappa)]\ \vec{\theta} \\ \mathsf{CSwitch}\ (i :: \mathfrak{V}, \vec{\theta'}, \mathcal{L}', \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m') = S\end{array}}{\Psi \vdash (i :: \mathfrak{V}, \vec{\theta}, \mathcal{L}, \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m) \implies S}$$

$$\text{cstep-fork}\ \frac{\begin{array}{c}\vec{\theta}_i = (\rho, \phi, \mathsf{Krun}\,\mathsf{fork}\,v\,\vec{v} \cdot \kappa) \\ (\rho, \phi_{\mathsf{fp}}, m) \models \mathbf{true} * v : \{\mathsf{validly\ precisely}\ P\}\{Q\} \\ \phi = \phi_{\mathsf{parent}} \oplus \phi_{\mathsf{child}} \\ (\mathsf{marshal}(\vec{v}), \phi_{\mathsf{child}}, m) \models \mathsf{validly\ precisely}\ P \\ \vec{\theta'} = [i \mapsto (\rho, \phi_{\mathsf{parent}}, \mathsf{Krun}\,\kappa)]\ \vec{\theta} \\ \vec{\theta''} = \vec{\theta'} + ((\rho_0, w_{\mathsf{child}}, \mathsf{Krun}\,(\mathsf{call}\,v\,\vec{v} \cdot \mathsf{Kstop})) :: \mathtt{nil}) \\ \mathsf{CSwitch}\ (i :: \mathfrak{V}, \vec{\theta''}, \mathcal{L}, \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m) = S\end{array}}{\Psi \vdash (i :: \mathfrak{V}, \vec{\theta}, \mathcal{L}, \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m) \implies S}$$

Figure 8.4: Fully concurrent steps in the concurrent step relation

cussed why this is reasonable.

In contrast, almost all of the other cases of the concurrent step relation do context switch. The context switch relation, $\mathsf{CSwitch}\ (S) = S'$, handles all of the details of performing a context switch by removing the head of the schedule (thus allowing the next thread to execute) and aging all of the worlds.

Thread exit (cstep-texit) does not remove the thread from the list, but scheduling a terminated thread simply results in a context switch. As explained in chapter 4, we do not reason about the resources that threads free (or fail to free) on termination, so we do not place any restrictions on ϕ.

Figure 8.4 contains the portion of the step relation that is responsible for executing the fully-concurrent statements.

Three rules describe lock acquisition. Rule cstep-prelock checks to make sure that the location that the thread is attempting to lock is a lock in the thread's resource map and then changes the thread's status from runnable (Krun) to waiting on a lock (Klock). Finally, cstep-prelock context switches to give other threads a chance to grab the lock first.

Rule cstep-nolock is the "block" case of a blocking lock—we try to grab the lock, but it is not yet available. In this case we simply context switch.

Rule cstep-lock is the rule that actually acquires the lock. It flips the memory location of the lock from a 1 to a 0 and then joins the resource map associated with the lock ϕ_{lock} in the lock pool $\mathcal{L}$ with

the resource map of the thread. By the restrictions on the lock pool contained in the consistant machine state, this resource map satisfies the lock's resource invariant[4]. Rule cstep-lock does not context switch after grabbing the lock. If we wished to context switch here it would not be difficult to modify our proofs; however it is unnecessary (since we quantify over all schedulers) and so we do not do it. See section 5.3 for a further discussion of interleaving.

The cstep-unlock rule reverses the lock rule. It flips the memory location associated with the lock from a 0 to a 1 and puts the resource map associated with the lock ϕ_{lock} into the lock pool $\mathcal{L}$. It is able to uniquely identify this resource map due to the premise

$$(\rho, \phi_{\mathsf{lock}}, m) \models \mathsf{tightly}\ P,$$

since tightly P is tight, and therefore precise. Since P is an arbitrary predicate in (classical) logic, this premise makes our semantics nonconstructive, *i.e.*, noncomputable. See section 8.6 for further discussion of why this abstraction is reasonable.

In the Coq development we use a different premise,

$$(\rho, \phi_{\mathsf{lock}}, m) \models \triangleright\, \mathsf{tightly}\ P,$$

[4]The invariant is satisfied on all more approximate resource maps due to the way YES inversion works. To establish that it holds at the current level of approximation is impossible in the general case, but in the case that actually occurs in the proof of the lock rule of CSL, a complex induction is able to prove that it holds immediately. Perhaps a modification of this rule would somehow lead to a simpler proof of the CSL lock rule.

which is related to the $\triangleright$ discussed in 8.4.6. When we remove the $\triangleright$ there we can simultaneously remove the $\triangleright$ in the cstep-unlock rule.

Finally, cstep-fork creates a new thread. Because we spawn function-calls as opposed to arbitrary commands that might have free C-minor variables, we don't need complex side-conditions on free variables and can use the empty environment ρ_0 in the new thread. There is a non-constructive test that the function's precondition is satisfied:

$$(\mathsf{marshal}(\vec{v}), \phi_{\mathsf{child}}, m) \models \mathsf{validly\ precisely}\ P.$$

In this test the parameters of the function call are marshaled into a local environment ρ so that the predicate will be able to judge them. Like the unlock rule, this test is nonconstructive. The sequential semantics does not need such a test at every function call; we need it here to know what world is transferred by the (precise) predicate P that is the function's precondition. Preconditions of nonspawnable (ordinary) functions need not be precise.

8.6 Reasonableness of the step relation

Our semantics has two features that are unusual in a concurrent semantics. The first unusual feature, exhibited in the cstep-seq and cstep-lock rules, is that the semantics does not interleave at every instruction. We discussed this feature in section 5.3.

The second unusual feature of our semantics is the nonconstructive

tests in the cstep-unlock and cstep-fork rules. If we are executing a program for which we have a proof in CSL, then we can prove that the test will always succeed. However, the presence of these nonconstructive tests is unsatisfying. More generally, the presence of resource maps in the semantics is worrying: we need to verify that they are not allowing unrealistic behavior. We can defend both the nonconstructive tests and the presence of resource maps in the semantics by relating our semantics to the erased semantics defined in chapter 5.

The erased machine defined in chapter 5 is similar to the unerased machine defined in this chapter; the difference is that in the erased machine all of the resource maps have been removed. The unerased machine is actually a conservative approximation to the erased one, as demonstrated by the erasure theorem:

> **Theorem (Erasure)**. If $\Psi \vdash (\mathcal{U}, \vec{\theta}, \mathcal{L}, \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m) \implies (\mathcal{U}', \vec{\theta}', \mathcal{L}', \phi'_{\mathsf{mp}}, \phi'_{\mathsf{fp}}, m')$, and $\vec{\theta}^{\mathsf{e}}$ and $\vec{\theta}'^{\mathsf{e}}$ are erased versions of the thread lists $\vec{\theta}$ and $\vec{\theta}'$, respectively, then $\Psi \vdash (\mathcal{U}, \vec{\theta}^{\mathsf{e}}, m) \overset{\mathsf{e}}{\implies} (\mathcal{U}', \vec{\theta}'^{\mathsf{e}}, m')$.
>
> **Proof**. By case analysis on the $\implies$ relation. We observe that each case in that relation has a corrosponding case in the $\implies_{\mathsf{e}}$ relation with a subset of the premises. In each case the schedule, locals (inside a thread), concurrent controls (inside a thread), thread list (up to erasure), and memory are identical. For the cstep-seq case we use lemma 8.2. □

This is a useful sanity check: the *real* machine takes no decisions

based on erasable information; the erased semantics simply gets stuck less often.

When to erase. One could imagine proving safety of a concurrent program with respect to the unerased semantics, then erasing, and last compiling. However, this would be a mistake since the compiler may do concurrency-unsafe optimizations. Instead, we should preserve the resource maps in the semantics of each intermediate representation as we compile from Concurrent C minor to machine language; this gives the compiler a specification of concurrency-safe optimizations. After we have reached machine language, we can use the resource maps to prove that our interleaving model is sound for machines with weak memory models. We erase last.

8.7 Conclusions

We have now given a formal concurrent operational semantics for Concurrent C minor. This semantics isolates the behavior of the sequential step relation from the concurrent reasoning, making it easy to use to reason about concurrent features such as locking a lock. However, it is not a major goal to use our operational semantics to reason about the sequential features of Concurrent C minor. To do so we define an oracular semantics for Concurrent C minor in chapter 9.

Chapter 9

Oracle Semantics

In chapter 3 we introduced Concurrent C minor, and in chapter 8 we gave it a formal concurrent operational semantics. That semantics allows for natural reasoning about the fully concurrent operations of Concurrent C minor, such as lock and unlock. However, it does not allow for natural reasoning about the sequential language features such as control flow. In this chapter we define an *oracular semantics* for Concurrent C minor that is ideal for reasoning about sequential language features in the context of a concurrent program[1].

9.1 Why an oracular semantics

A compiler, or a Hoare tuple, considers a single thread at a time. The compiler (and its correctness proof) wants to compile code uniformly even around the concurrent operations. Similarly, in a CSL proof, the

[1]Portions of this chapter have been published before as [HAZ08a] and [HAZ08b].

statement	sequential semantics	oracular semantics
	σ_0	(Ω_0, σ_0)
$[e_1]{:=}e_2$		
	σ_1	(Ω_0, σ_1)
if e then s_1 else s_2		
	σ_2	(Ω_0, σ_2)
lock v		
	stuck	(Ω_1, σ_3)
$[e_1]{:=}e_2$		
		(Ω_1, σ_4)

Figure 9.1: The oracle allows for reasoning after a concurrent instruction

commands c_1 and c_2 in

$$\{P\}\ c_1\ ;\ c_2\ \{Q\}$$

may contain concurrent operations, but, because of C minor's nonlocal exits, a soundness proof for the sequence rule of separation logic is complicated even without the headaches of concurrency. We want a semantics of single-threaded computation in a concurrent context.

The sequential submachine of section 8.2 is single-threaded but incomplete because it gets stuck at the fully-concurrent operations lock, unlock, and fork. What we want is a deterministic sequential operational semantics[2] that knows how to handle concurrent communications, which it will do by consulting an oracle.

[2]The correctness proofs of the CompCert compiler and the sequential separation logic proofs use determinism to simplify their task.

Figure 9.1 demonstrates the value of this type of semantics. In the left column is a sequence of statements in Concurrent C minor. The first two statements are sequential statements, the third is a fully-concurrent lock statement, and the final statement is sequential. In the middle column is a series of states σ (*i.e.*, memory, locals, etc.) from a sequential semantics such as the sequential submachine; the series starts with some initial state σ_0. In the right column is a series of oracle Ω and state σ pairs for a new oracular semantics; we start with the same state σ_0 as in the regular sequential semantics, as well as some intitial oracle Ω_0.

As each statement is executed, the states in the middle and right columns are updated by the action of the statement. After executing the first statement, which is a store to memory, in the middle column we have a new state σ_1. In the right column, we get exactly the same state σ_1. Since a store instruction is not a concurrent instruction, we do not use the oracle and so it is unchanged. Assuming that the substatements s_1 and s_2 do not contain concurrent instructions, we reach the subsequent state σ_2.

Now we reach a concurrent statement, lock. Here, the standard sequential semantics is stuck; the sequential submachine is not able to determine the result of the lock instruction, since it depends on the other threads. However, on the right-hand side, the semantics is able to take advantage of the oracle Ω_0. The oracle represents all of the other threads in the concurrent machine. The right-hand semantics consults the oracle, which predicts what the state σ_3 will be after the concurrent

machine executes the lock instruction, and also produces a new oracle Ω_1, which will be usable for the next concurrent instruction. After the concurrent instruction, the right-hand side is able to continue to process sequential statements as before.

To build the desired semantics we will build an *oracular machine* using our extension system explained in chapter 6. Our extension will handle all of the concurrent instructions of Concurrent C minor.

9.2 Concurrent oracle

Like any extension, we must define the type of an oracle and build a partial function $\mathsf{consult}$. When we constructed the sequential submachine in section 8.2, we used the `unit` oracle. Here, we define a more meaningful oracle as follows:

$$\Omega \ : \ \mathsf{oracle} \ := \ (\mho, \vec{\theta}, \mathcal{L}).$$

An oracle now contains a schedule $\mho$, a thread list $\vec{\theta}$, and a lock pool $\mathcal{L}$.

We generalize a *sequential continuation* (Ω, w, κ) to a *concurrent continuation* $(\Omega, w, \hat{\kappa})$, whose concurrent control $\hat{\kappa}$ may be ready $(\mathsf{Krun}\,\kappa)$ or blocked on a lock $(\mathsf{Klock}\,v\,\kappa)$. An oracle allows us to build a concurrent machine state S from a thread number i and a concurrent continuation. Alternatively, given a concurrent machine state S and thread-ID i, we can construct a concurrent continuation (oracle Ω, world w, and

$$\text{projection}\ \frac{\begin{array}{c}\phi = \phi_{\mathsf{t}} \oplus \phi_{\mathsf{mp}} \oplus \phi_{\mathsf{fp}} \\ (\rho, \phi_{\mathsf{mp}}, m) \models \mathsf{allocpool} \\ \forall l.\ (\mathsf{rkind}(\phi_{\mathsf{fp}}@l) = \mathsf{None}) \vee (\mathsf{rkind}(\phi_{\mathsf{fp}}@l) = \mathsf{Some\ kFUN}) \\ \forall l.\ \mathsf{rkind}(\phi_{\mathsf{t}}@l) \neq \mathsf{Some\ kFUN} \\ \Omega = (\mho, \vec{\theta}, \mathcal{L}) \\ \mathsf{age\_list}\ n\ \vec{\theta} = \mathsf{Some}\ \vec{\theta}' \\ \mathsf{age\_lpool}\ n\ \mathcal{L} = \mathsf{Some}\ \mathcal{L}' \\ \vec{\theta}' = [\theta_1, \ldots, \theta_{i-1}, \theta_{i+1}, \ldots, \theta_n] \\ \vec{\theta}'' = [\theta_1, \ldots, \theta_{i-1}, (\rho, \phi_{\mathsf{t}}, \hat{\kappa}), \theta_{i+1}, \ldots, \theta_n]\end{array}}{(\Omega, (\rho, \phi, m), \hat{\kappa}) \overset{i}{\propto} (\mho, \vec{\theta}'', \mathcal{L}', \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m)}$$

Figure 9.2: Oracular projection

concurrent control $\hat{\kappa}$). The precise relationship is given by the relation $(\Omega, w, \hat{\kappa}) \overset{i}{\propto} S$, pronounced "$(\Omega, w, \hat{\kappa})$ is the ith projection of S".

The projection relation is given in figure 9.2. To build the concurrent machine state S from the concurrent continuation $(\Omega, (\rho, \phi, m), \hat{\kappa})$ and thread number i, we first split the alloc and function pools from the resource map ϕ, leaving the remaining resource map ϕ_{t}. The predicate allocpool is precise, and the function pool contains all of the functions and nothing more, so splitting them off is deterministic. Next we can age the thread list $\vec{\theta}$ and lock pool $\mathcal{L}$ in case they are less approximate than the resource maps ϕ, ϕ_{mp}, and ϕ_{fp}. Finally we insert the thread $(\rho, \phi_{\mathsf{t}}, \hat{\kappa})$ into the thread list at position i.

$$\text{Ready}\ \frac{\vec{\theta}_i = (\rho, w, \mathsf{Krun}\,\kappa)}{\text{Ready}\ i\ (i :: \mho, \vec{\theta}, \mathcal{L}, \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m)}$$

$$\text{SO-done}\ \frac{\text{Ready}\ i\ S}{\Psi \vdash \mathsf{StepOthers}\ i\ S\ S}$$

$$\text{SO-step}\ \frac{\neg(\text{Ready}\ i\ S) \quad \Psi \vdash S \Longmapsto S' \quad \Psi \vdash \mathsf{StepOthers}\ i\ S'\ S''}{\Psi \vdash \mathsf{StepOthers}\ i\ S\ S''}$$

Figure 9.3: Running the other threads

9.3 Concurrent consult relation

We need to build a consult relation to execute the concurrent statements of Concurrent C minor. The consult function constructs a concurrent machine S from the oracle and then runs the other theads in that machine until control returns.

We run the other threads with the StepOthers relation, defined in figure 9.3. The StepOthers relation takes a thread-ID i and two concurrent machine states S and S'. We say that thread i is *ready* when i is at the head of the schedule in S. When we step other threads we have two choices. The first case, SO-done, says that if thread i is ready, then we have finished stepping other threads and control has returned to thread i. The second case, SO-step, says that if thread i is not ready, then we take a step in the concurrent machine and then test again.

Now that we have defined how we run the other threads, we are ready to define the consult relation in figure 9.4. There are three cases

$$\Omega\text{-invalid}\ \frac{\Omega = (i :: \mho, \vec{\theta}, \mathcal{L}) \quad \nexists S.\ (\Omega, w, \mathsf{Krun}\,\kappa) \overset{i}{\propto} S}{\mathsf{consult}(\Psi, \Omega, (w, \kappa)) = \mathsf{None}}$$

$$\Omega\text{-diverges}\ \frac{\begin{array}{c}\Omega = (i :: \mho, \vec{\theta}, \mathcal{L}) \\ (\Omega, w, \mathsf{Krun}\,\kappa) \overset{i}{\propto} S \\ \Psi \vdash S \Longrightarrow S' \\ \nexists S''.\ \Psi \vdash \mathsf{StepOthers}\ i\ S'\ S''\end{array}}{\mathsf{consult}(\Psi, \Omega, (w, \kappa)) = \mathsf{None}}$$

$$\Omega\text{-steps}\ \frac{\begin{array}{c}\Omega = (i :: \mho, \vec{\theta}, \mathcal{L}) \\ (\Omega, w, \mathsf{Krun}\,\kappa) \overset{i}{\propto} S \\ \Psi \vdash S \Longrightarrow S' \\ \Psi \vdash \mathsf{StepOthers}\ i\ S'\ S'' \\ (\Omega', w', \mathsf{Krun}\,\kappa) \overset{i}{\propto} S''\end{array}}{\mathsf{consult}(\Psi, \Omega, (w, \kappa)) = \mathsf{Some}\ (\Omega', (w', \kappa'))}$$

Figure 9.4: The oracular consult relation

when the oracular machine wants to consult.

In the first case, Ω-invalid, there is no concurrent machine state S compatible with the concurrent continuation. This happens if the oracle Ω is somehow incompatible with the world w. There are a variety of ways that this could be the case; one simple example is that one of the threads inside the oracle thinks that it has full ownership of a location that is also owned by the world w. Of course, in the concurrent machine this is forbidden by the consistency requirements.

It our proofs it is convienent to quantify universally over all oracles, instead of of simply quantifying over valid oracles, which would require an additional premise in the definitions and proofs. By looping safely if

we are handed an invalid oracle by the universal quantification, we can gracefully handle invalid oracles without requiring such a premise. By returning None from the consult function we indicate that the semantics should loop endlessly, thereby trivially becoming safe.

In the remaining two cases, we are able to construct a concurrent machine S, and take at least one concurrent step. This step uses the cstep-seq case in the concurrent step relation to execute a make_lock or free_lock, uses cstep-prelock to execute the first part of a lock, uses cstep-unlock to execute an unlock, or uses cstep-fork to execute a fork.

After taking this step, the machine decides (classically) whether control will return to the current thread by branching on the StepOthers judgement. Rule Ω-diverges handles the case when control does not return, for example if the schedule is unfair, if another thread executes an illegal instruction, or if the current thread is deadlocked. In these cases the oracle machine loops endlessly, thereby becoming trivially safe. The idea that we are safe if another thread blows up may seem strange: the point is that we are proving the safety of *this* thread. If another thread causes the concurrent machine to get stuck, it is not this thread's fault.

The final case, rule Ω-steps, is when control returns after running the other threads. In this case we project out the ith thread from the new concurrent machine state S'' and proceed with the new oracle, world, and control that came from running the concurrent machine.

Classical reasoning is unavoidable in this system: first, the concur-

rent machine itself requires classical reasoning to find a world satisfying an unlock assertion because we allow users to use a classical logic, and second, determining whether control will return to a given thread reduces the halting problem. The nonconstructivity of our operational semantics is not a bug: we are not building an interpreter, we are building a specification for correctness proofs of compilers and program logics.

9.4 The oracular step

As explained in section 6.6, we define the oraclular step relation using the consult relation. As explained in that section, the oracular step uses the underlying step relation (which is the step relation for sequential C minor) when executing core statements.

We use the oracular step to keep "unimportant" details of the concurrent machine from interfering with proofs about the sequential language. The key features of the oracular step are the following:

1. It is deterministic, which simplifies sequential metatheory
2. When it encounters a synchonization operation, it is able to make progress with the oracle, whereas a regular step relation gets stuck
3. It composes with itself, whereas the regular step relation does not (because memory will change between steps due to other threads)
4. In the cases where control would never return, such as deadlock, we will be safe.

With these properties we can prove properties of the sequential features of Concurrent C minor in a natural way.

9.5 Conclusion

In chapter 8 we gave Concurrent C minor a formal concurrent operational semantics. That semantics was easy to use for reasoning about the concurrent features of the language, but not natural to use for reasoning about sequential features. In this chapter we have defined an oracular semantics for Concurrent C minor to support natural reasoning about sequential features.

We have not yet formally stated the precise relationship between the oracular semantics and the concurrent semantics. The oracular semantics is built from the concurrent semantics, but we wish to reason somewhat differently. The oracular semantics is a "thread-local" version of the semantics, so it is natural to think that if we could prove properties of each thread using the oracular semantics, then we could combine those proofs into a proof of the entire concurrent machine. However, we are not yet in a position to formally state the connection, which will be the subject of section 10.3.

Chapter 10

A Modal Hoare Judgment and Oracular Soundness

In chapter 3 we introduced Concurrent C minor. In chapter 4 we gave Concurrent C minor an axiomatic semantics, Concurrent Separation Logic. In chapter 8 we gave Concurrent C minor an operational semantics. At last we are ready to connect these two semantics by proving the soundness of the CSL axioms with respect to the operational semantics.

Our CSL axiomatic semantics is designed to reason about one thread at a time, and we noted in chapter 8 that the concurrent operational semantics was not particularly easy to use for reasoning about such sequential computation. In chapter 9 we developed an oracle semantics that presents a single-threaded view of the concurrent machine and is suitable for reasoning about one thread at a time.

We connect our axiomatic semantics to our operational semantics in

three stages. First, in section 10.1 we develop a definition for the Hoare tuple based on the oracle semantics. Second, in section 10.2 we connect our axiomatic semantics to our oracle semantics by proving the rules of CSL as lemmas from that definition. Third, in section 10.3 we connect our oracle semantics to our concurrent semantics, thereby proving the soundness of our approach[1].

10.1 A modal Hoare judgment

Here we develop the definition for the Hoare tuple. First, in section 10.1.1, we explain a simple continuation-passing definition in the style of Appel and Blazy [AB07]. Then in section 10.1.2 we explain why this style of definition will not allow us to embed predicates into program syntax as we wish to do for the make_lock statement of Concurrent C minor. Finally, we show how to redefine the Hoare tuple using our modal logic of chapter 7 and explain how the new definition allows us to embed predicates into program syntax.

10.1.1 A continuation-passing Hoare judgment

Our semantic model for Hoare tuples is rooted in the idea of safety, defined in figure 10.1. We start with the notion of *immediately safe*. A state σ is immediately safe if it is not stuck; in other words, if it is safely halted or can take a step. A state can be safely halted in two

[1] Portions of this chapter have been published before as [HAZ08a] and [HAZ08b].

$$\text{isafe\_stop}\frac{}{\Psi \vdash \mathsf{immediately\_safe}\ (\Omega, (w, \mathsf{Kstop}))}$$

$$\text{isafe\_level}\frac{\mathsf{level}\ \phi = 0}{\Psi \vdash \mathsf{immediately\_safe}\ (\Omega, ((\rho, \phi, m), \kappa))}$$

$$\text{isafe\_step}\frac{\Psi \vdash (\Omega, \sigma)\ \mapsto\!\!\!\!\longrightarrow\ (\Omega', \sigma')}{\Psi \vdash \mathsf{immediately\_safe}\ (\Omega, \sigma)}$$

$$\text{stepstar\_0}\frac{}{\Psi \vdash (\Omega, \sigma)\ \mapsto\!\!\!\!\longrightarrow^*\ (\Omega, \sigma)}$$

$$\text{stepstar\_S}\frac{\Psi \vdash (\Omega, \sigma)\ \mapsto\!\!\!\!\longrightarrow\ (\Omega', \sigma') \quad \Psi \vdash (\Omega', \sigma')\ \mapsto\!\!\!\!\longrightarrow^*\ (\Omega'', \sigma'')}{\Psi \vdash (\Omega, \sigma)\ \mapsto\!\!\!\!\longrightarrow^*\ (\Omega'', \sigma'')}$$

$$\Psi \vdash \mathsf{safe}\ \sigma \quad = \quad \forall\, \Omega, \Omega', \sigma'.\ \ \Psi \vdash (\Omega, \sigma)\ \mapsto\!\!\!\!\longrightarrow^*\ (\Omega', \sigma') \quad \rightarrow \quad \Psi \vdash \mathsf{immediately\_safe}\ (\Omega', \sigma')$$

Figure 10.1: Oracular safety

ways. The first, case isafe_stop, is the standard way of saying that the machine has been safely halted when the code has reached the end of the computation. The second, case isafe_level, is less standard. After we reach approximation level 0, we will no longer be able to age the resource maps; moreover, all of the stratified predicates are now `unit` and therefore no longer guarantee anything.

This does not mean that the program is now unsafe; it means that the program's observer has stopped caring about the computation. We prove our programs sound starting with arbitrarily large amounts of stratification. Therefore, our programs are proved sound with regard to any finite amount of time. If we wish to know that our program is safe

for 100 steps of computation, we stratify predicates to level 100 (we age the world at most once per step of computation). If we wish to know that our program is safe for 100 billion steps of computation, we stratify predicates to level 100 billion. The third way of being immediately safe, case isafe_step, is the standard way of observing that if we can take a step then we are not stuck.

We define the $\mapsto^*$ relation in the usual way as the composition of the $\mapsto$ relation with cases stepstar_0 and stepstar_S. We then define safe in the usual way: for any state σ' reachable from σ, σ' is immediately safe. Notice that we quantify universally over all initial oracles Ω; in section 9.3 we explained that quantifying over all oracles instead of simply valid ones is sound due to the Ω-invalid case of the oracular $\mathsf{consult}$ relation: if we get an invalid oracle we are safe if we try to consult it because we loop forever. The unrestricted universal quantification simplifies the proofs that do not use the oracle since it means that they do not have to carry around validity premises about the oracle, which would weaken the isolation between the sequential and concurrent reasoning.

Now that we have defined safety, we are ready to start to define the semantics of our Hoare tuple. In figure 10.2 we give a naïve definition of the Hoare tuple in a continuation-passing style developed by Appel and Blazy. This is a simpler definition than the one they use, which supports the frame rule in a more semantic style, and has additional parameters to handle nonlocal exits from blocks and functions. Their rule was given

$$\Psi \vdash P \,\square\, \kappa \;=\; \forall\, w.\;\; w \models P \;\rightarrow\; \Psi \vdash \mathsf{safe}(w, \kappa)$$

$$\Gamma \vdash \{P\}\ c\ \{Q\} \;\approx\; \forall\, \kappa.\;\; \Psi \vdash \Gamma * Q \,\square\, \kappa \;\rightarrow\; \Psi \vdash \Gamma * P \,\square\, c \cdot \kappa$$

Note: it is not possible to prove a rule for function call because Ψ is free in the definition of $\Gamma \vdash \{P\}\ c\ \{Q\}$.

Figure 10.2: Naïve continuation-passing style definition of Hoare tuple

in full in figure 2.5 in chapter 2. In the Coq development we include all of the features they describe to handle the nonlocal exits and semantic frame rule, but here we eliminate many of the more advanced features, which are for sequential control flow, to concentrate on the key ideas.

We define the notion that a predicate P *guards* a control κ in the context of a program Ψ, written $\Psi \vdash P \,\square\, \kappa$. The idea is that P permits only states on which it is safe to run κ. Thus, for all worlds w such that $w \models P$, the state (w, κ) is safe.

Next, we define the Hoare tuple $\Gamma \vdash \{P\}\ c\ \{Q\}$. For any sequence of instructions κ, if $\Gamma * Q$ guards κ then $\Gamma * P$ must guard the statement c followed by κ. We add the parameter Γ, which is the assertion that describes the functions, to both the pre- and postconditions as a convenience for the user, so that he does not have to write $\{\Gamma * P\}\ c\ \{\Gamma * Q\}$.

This is a continuation-passing style definition, and it may not be obvious at first glance that it is equivalent to the informal notion of "if we start with a state that satisfies P, and we run s, and s terminates,

then we end with a state that satisfies Q". It is equivalent because we are quantifying over *all* instruction sequences κ that are guarded by Q. For any property of Q that is checkable with our operational semantics, we can develop a small tester sequence that tests the property and gets stuck if it does not hold. If we are still concerned, it is easy to add a statement that asserts that the predicate Q holds and if it does not then gets stuck.

This definition is one of partial correctness; in other words it does not assume that the statement s will terminate. However, since it asserts that $s \cdot \kappa$ is safe if P holds, then it does imply that the statement s does not get stuck.

Using this definition, we can prove all of the sequential rules of separation logic except for the rule for `call`. In fact, as mentioned above, Appel and Blazy use a more complex version of this definition to prove all of the rules of sequential separation logic with respect to C minor in Coq. Unfortunately, in the current context, we cannot use the definition of the Hoare tuple given in figure 10.2 to prove the soundness of the function call CSL rule:

$$\Gamma * (f : \{P\}\{Q\}) \vdash \{P\} \ \mathsf{call}\, f \ \ \{Q\}.$$

The problem is that the program Ψ, used in function call to match a function name with a function body, is free in the definition of the Hoare rule. Obviously, a sound definition should not have any free variable;

unfortunately, fixing the problem is not easy. One solution is to make Ψ a parameter of the Hoare tuple, an approach taken by Schirmer [Sch06]. However, this solution requires the CSL use to drag around his entire program through his correctness proofs, which is unfortunate, and makes function pointers more difficult to use. Moreover, the user already is passing around a description of the functions in Γ—why should he also pass around the program Ψ?

We cannot simply quantify universally over the program Ψ; this makes the call rule easy to use—but impossible to prove, so it will not do. Another bad idea is to quantify existentially over the program Ψ; this makes the call rule easy to prove—but impossible to use.

We are fast running out of options. Appel and Blazy put the program Ψ into the world w and let predicates judge program syntax. However, in our context this solution is not good enough since we do not wish to put program syntax into worlds.

10.1.2 Embedding predicates into syntax

We cannot put program syntax into the world due to the $\mathsf{make\_lock}\, l\, R$ statement. This statement takes a semantic predicate R—*i.e.*, a function from world to `Prop`—that becomes the resource invariant of the new lock l. Thus, a predicate is embedded into program syntax.

Since program syntax is directly exposed to the user, we wish to provide a simple inductive definition for program syntax and avoid complex stratification techniques. If we wish to avoid stratification techniques,

then the definition for the type of "predicate" must come before the definition of program syntax. Since a predicate is a function from world to `Prop`, a world cannot contain program syntax, which is why a world is simply a tuple of locals ρ, resource map ϕ, and memory m.

One very important predicate is $f : \{P\}\{Q\}$, *i.e.*, "f is a function with precondition P and postcondition Q". This predicate is used in the precondition of both the call and fork CSL rules; the real difficulty is in proving the call rule. What is the relationship between the assertion $f : \{P\}\{Q\}$ and the code that makes up f—that is, $\Psi(f)$? In our naïve definition of the Hoare tuple, there was no relation at all, since Ψ was a free variable.

As we discussed, we do not wish to provide Ψ as a parameter to the Hoare tuple. Instead, what we want to do is quantify over all "good" programs Ψ – that is programs compatible with Γ. The idea is to add an additional premise after a universal quantification; this way we hope to have a definition that is both possible to use and possible to prove.

What we are looking for is some relation $\Gamma \vdash \Psi$, called the *believe relation*, which says that the program Ψ is compatible with Γ. Compatible with Γ means that if there is an assertion $f : \{P\}\{Q\}$ in Γ, then we can believe that assertion about the function body $\Psi(f)$.

Given such a relation, we can then attempt to define the Hoare tuple

as follows:

$$\Gamma \vdash \{P\}\ c\ \{Q\} \approx\ \forall \Psi.\ \ \Gamma \vdash \Psi\ \rightarrow \\ \forall \kappa.\ \Psi \vdash \Gamma * Q \mathbin{\square} \kappa\ \rightarrow \\ \Psi \vdash \Gamma * P \mathbin{\square} c \cdot \kappa$$

Recall that we use $\approx$ to mean "we wish we could define things this way". This definition says that for any program Ψ compatible with Γ, if Q guards κ, then P must guard $c \cdot \kappa$.

What remains is to define the relation $\Gamma \vdash \Psi$. The obvious choice is

$$\Gamma \vdash \Psi \approx\ \forall f, P, Q.\quad (\Gamma \vdash \mathbf{true} * f : \{P\}\{Q\})\ \rightarrow \\ \Psi \vdash \{P\}\ \Psi(f)\ \{Q\}$$

In other words, for any function assertion that is part of Γ, the Hoare tuple guarantees that the body of that function has precondition P and postcondition Q.

Unfortunately, there is a problem with these definitions: the definition of the Hoare tuple depends on the believe relation, while the definition of the believe relation depends on the definition of the Hoare tuple. Worst of all, this dependence is contravariant, meaning that no solution exists, even in untyped set theory.

10.1.3 A modal judgment

Fortunately, we have our modal logic from chapter 7, which was developed to handle just this kind of thorny contravariant recursive situation.

Although we originally developed it to model lock invariants, we can use it to define the Hoare tuple itself since the logic has powerful features such as impredicative quantification and higher-order recursion.

The Hoare tuple we explain here is much simpler than the one in the Coq development. The major simplifications include:

1. **No globals in the world type**. As in chapter 7, we removed from the world type the map from global names to addresses.

2. **No relation between function pre- and postconditions**. As in chapter 7, we have removed the extra parameter that relates function pre- and postconditions to each other.

3. **No marshaling of arguments**. As explained in chapter 8, we have a method of marshaling arguments at function call so that our function preconditions can judge them. We omit this feature here for simplicity.

4. **No support for nonlocal exits**. As explained in chapter 2, C minor supports various kinds of nonlocal exits; to reason about these we add parameters R and B to the Hoare tuple. We omit them here since these features are directly related neither to concurrency nor to the core of our new definition of the Hoare tuple.

5. **Miscellaneous**. Other changes to improve the presentation.

We define our Hoare tuple using our modal substructural logic in figure 10.3. Recall that a predicate in the logic is a function from world

$$
\begin{array}{rcl}
\mathsf{safe}_\Psi\ \kappa & : & \mathsf{predicate} \\
& = & \lambda w.\ \Psi \vdash \mathsf{safe}(w, \kappa) \\
\\
P \Box_\Psi \kappa & : & \mathsf{predicate} \\
& = & P \subset \mathsf{safe}_\Psi\ \kappa \\
\\
\mathsf{Hargs} & : & \mathsf{type} \\
& = & \mathsf{predicate} \times \mathsf{predicate} \times \mathsf{statement} \times \mathsf{predicate} \\
\\
\mathsf{believe} & : & (\mathsf{Hargs} \to \mathsf{predicate}) \to \mathsf{predicate} \to \mathsf{program} \to \\
& & \quad \mathsf{predicate} \\
& = & \lambda H.\ \lambda \Gamma.\ \lambda \Psi. \\
& & \quad \forall f, P, Q. \\
& & \qquad (\Gamma\ \subset\ \mathbf{true} * f : \{P\}\{Q\})\ \subset \\
& & \qquad \rhd H(\Gamma, P, \Psi(f), Q) \\
\\
\mathsf{Htuple} & : & \mathsf{Hargs} \to \mathsf{predicate} \\
& = & \mu_{\mathsf{HO}}\ \mathsf{Hargs}\ (H : \mathsf{Hargs} \to \mathsf{predicate}). \\
& & \lambda(\Gamma, P, c, Q). \\
& & \quad \forall \Psi.\ \mathsf{believe}\ H\ \Gamma\ \Psi\ \Rightarrow \\
& & \qquad \forall \kappa.\ (\Gamma * Q) \Box_\Psi \kappa\ \Rightarrow \\
& & \qquad\quad (\Gamma * P) \Box_\Psi (c \cdot \kappa) \\
\\
\Gamma \vdash \Psi & : & \mathsf{predicate} \\
& = & \mathsf{believe}\ \mathsf{Htuple}\ \Gamma\ \Psi \\
\\
\Gamma \vdash \{P\}\ c\ \{Q\} & : & \texttt{Prop} \\
& = & \vdash\ \mathsf{Htuple}(\Gamma, P, c, Q)
\end{array}
$$

Figure 10.3: A modal Hoare tuple

(which is a tuple of locals ρ, resource map ϕ, and memory m) to `Prop`.

We start with $\mathsf{safe}_\Psi\,\kappa$, or predicate-level safe, which lifts the safe notion into the modal substructural logic by "hardcoding" the program Ψ and control κ into the predicate. If

$$w \models \mathsf{safe}_\Psi\ \kappa,$$

then the control κ is safe with world w in the context of program Ψ.

Just as in the case of the simpler definitions, we define the notion of a predicate P guarding a control κ in the context of a program Ψ with $P \,\square_\Psi\, \kappa$, or predicate-level guard. Recall from section 7.3.7 that the predicate $P \subset Q$ is a safe form of logical implication in the modal logic, informally equivalent to "On any world of equal or greater approximation than the current one, P implies Q". Therefore

$$P \,\square_\Psi\, \kappa$$

means that for any world w' with level less than or equal to the level of w, if $w' \models P$, then $\Psi \vdash \mathsf{safe}\ (w', \kappa)$.

Our modal Hoare judgment takes a single argument of type Hargs, which is a tuple of predicate (Γ), predicate (P), statement (c), and predicate (Q). We must uncurry the arguments so that we will be able to use the higher-order recursion operator μ_{HO}, which is how we will "tie the knot" when defining the believe and Htuple predicates.

The believe predicate informally says that for any function specifi-

cation $f : \{P\}\{Q\}$ in Γ, the body of the function f, that is, $\Psi(f)$, has precondition P and postcondition Q.

The first argument of believe is a function H from Hargs to predicate—that is, a function that has the same type as the Hoare tuple. The higher-order recursion operator μ_{HO} will "tie the knot" and make sure that H will be the Hoare tuple itself. The second argument to believe is the predicate Γ, and the third argument is the program Ψ; the purpose of believe is to relate Γ to Ψ using H.

First, believe quantifies over all functions f, preconditions P, and postconditions Q. Then for any function specification $f : \{P\}\{Q\}$ that is implied by Γ—that is, if[2]

$$\Gamma \subset \mathbf{true} * f : \{P\}\{Q\},$$

we require that $H(\Gamma, P, \Psi(f), Q)$ hold on all strictly more approximate worlds, which we enforce with the $\triangleright$ modal operator:

$$\triangleright H(\Gamma, P, \Psi(f), Q).$$

[2]In the Coq development, we use a slightly different form of the implication

$$\Gamma \subset \mathbf{true} * f : \{P\}\{Q\},$$

which is

$$\bigcirc! \, (\Gamma \Rightarrow \mathbf{true} * f : \{P\}\{Q\}).$$

This implication is very unusual for us in that we almost never use the "unsafe" form of implication $\Rightarrow$; moreover looking at it now we suspect that unsafe implication is not needed in the proofs. One of many projects for cleaning up the Coq proofs in the future will be to replace $\Rightarrow$ with $\subset$ in the Coq definition of believe.

We require the $\triangleright$ operator so that $\mathsf{believe}$ is contractive in H.

There is a subtlety in this definition involving the quantification over P and Q and the relationship of them to Γ. We are able to quantify over predicates P and Q because we support full impredicative quantifiation. Next, recall from section 7.3.10 that due to the way YES inverts, that for a given function f there are multiple P and Q that will be indistinguishable; all that we know is that they are equivalent at strictly greater approximation. This is another reason for applying the $\triangleright$ operator before we pass P and Q to H.

The Htuple predicate implements our Hoare tuple. We start with the higher-order recursion operator μ_{HO}, which was explained in section 7.3.9 and which has type

$$\mu_{\mathsf{HO}} \ : \ \forall\alpha.\,((\alpha \to \mathsf{predicate}) \to (\alpha \to \mathsf{predicate})) \to \alpha \to \mathsf{predicate}.$$

We instantiate the type parameter α with the type Hargs; then μ_{HO} binds the variable H for recursive self reference. Now we provide a function of type $\mathsf{Hargs} \to \mathsf{predicate}$, which we do with a λ as usual, pattern-matching the arguments (Γ, P, c, Q) of the Hoare tuple.

Now we universally quantify over all programs Ψ, and require

$$\mathsf{believe}\ H\ \Gamma\ \Psi,$$

which guarantees that Γ describes the functions contained in Ψ. The rest of the definition is very close to the one given in figure 10.2: for all

controls κ, if Q guards κ, then P guards $c \cdot \kappa$—the difference is that Ψ is no longer a free variable in the definition.

Once our Hoare tuple is defined we can define a notation $\Gamma \vdash \Psi$ for $\mathsf{believe\ Htuple}\ \Gamma\ \Psi$, which is equal to the $\mathsf{believe}$ inside the definition of Htuple by fold-unfold.

Finally, we define our "user-level" Hoare tuple $\Gamma \vdash \{P\}\ c\ \{Q\}$ as

$$\vdash \mathsf{Htuple}(\Gamma, P, c, Q),$$

using the notation defined in section 7.3.1, where $\vdash P$ is shorthand for $\forall w.\ w \models P$. A Hoare rule is sound only if it is true for all worlds.

10.2 Hoare judgments in CSL

We are now ready at last to prove the axioms of CSL sound with respect to the Concurrent C minor oracular semantics. Later, in section 10.3, we will connect the oracular semantics to the concurrent operational semantics and thereby achieve an end-to-end result.

The Hoare rules divide into three categories. The first category, which is by far the most numerous, covers all of the sequential Hoare rules except for call. The second category contains the call rule and the rules for building up the predicate Γ. The third category coveres the concurrent rules.

10.2.1 Sequential rules

In section 2.5.3 we presented the sequential Hoare rules of Appel and Blazy in figure 2.6. Appel and Blazy proved those rules sound in Coq with respect to sequential C minor; their proof was a sizable engineering development, and complex enough without worrying about complexities arising from concurrent computation.

Appel was able to adapt those Coq proofs to our new definitions without altering their essential structure; very little changed, providing strong evidence that our oracular semantics was able to hide the complexities of concurrency from the metatheory proofs about sequential language features, even in the extremely picky context of a machine-checked proof.

10.2.2 Function call

The proof of the call rule, on the other hand, did have to change since it interacts with the new $\Gamma \vdash \Psi$ predicate. In fact, the semantics of function call had to change so that the resource map was aged during the call; remember from figure 8.1 that rule sstep-call does just that. Here we prove a simplified CSL call rule where we remove the function arguments, but in the Coq development we prove the rule for the full C minor call statement. We will also assume that P and Q are valid in the sense given in section 7.3.8.

Theorem 10.1. $\Gamma * (f : \{P\}\{Q\}) \vdash \{P\}\ \mathsf{call}\, f\ \ \{Q\}$

Proof. We unfold the definition of the Hoare tuple and then introduce the premises $w \models (\Gamma * f : \{P\}\{Q\}) \vdash \Psi$ and $w \models (\Gamma * (f : \{P\}\{Q\}) * Q)\ \square_\Psi\ \kappa$; we then wish to prove $w \models (\Gamma * (f : \{P\}\{Q\}) * P) \square_\Psi \mathsf{call}\ f \cdot \kappa$. To prove this goal, we may assume a w' such that $\mathsf{level}\ w' \leq \mathsf{level}\ w$ and that $w' \models \Gamma * f : \{P\}\{Q\} * P$, and must prove $\Psi \vdash \mathsf{safe}(w', \mathsf{call}\ f\ \cdot \kappa)$. We can assume that $\mathsf{level}\ w' > 0$ since otherwise we would be immediately safe. Therefore, there exists w'' such that $\mathsf{age}\ 1\ w' = \mathsf{Some}\ w''$. By the sstep-call rule from figure 8.1, we can then take a step from $(w', \mathsf{call}\ f \cdot \kappa)$ to $(w'', \Psi(f) \cdot \kappa)$. By the definition of safety, it suffices to prove this state safe.

We know $\vdash (\Gamma * f : \{P\}\{Q\}) \subset (\mathbf{true} * f : \{P\}\{Q\})$. Therefore, by the definition of $w \models (\Gamma * f : \{P\}\{Q\}) \vdash \Psi$, and since we know that $\mathsf{level}\ w' \leq \mathsf{level}\ w$, we know that $w' \models \triangleright\, \mathsf{Htuple}(\Gamma * f : \{P\}\{Q\}, P, \Psi(f), Q)$. Since w'' is strictly more approximate than w' and since Htuple is fashionable, we know that $w'' \models \mathsf{Htuple}(\Gamma * f : \{P\}\{Q\}, P, \Psi(f), Q)$.

$\Gamma \vdash \Psi$ is fashionable, meaning that it holds on all worlds of the same level. Therefore, from $w \models (\Gamma * f : \{P\}\{Q\}) \vdash \Psi$, we know $w''' \models (\Gamma * f : \{P\}\{Q\}) \vdash \Psi$ for some w''' of the same level as w and such that there exists an n such that $\mathsf{age}\ n\ w''' = \mathsf{Some}\ w''$. $\Gamma \vdash \Psi$ is necessary, meaning that it holds on all worlds that are approximations of the current world, so from $w''' \models (\Gamma * f : \{P\}\{Q\}) \vdash \Psi$ we know that

$w'' \models (\Gamma * f : \{P\}\{Q\}) \vdash \Psi$. Since $P \,\Box_\Psi\, \kappa$ is both necessary and fashionable, from $w \models (\Gamma * (f : \{P\}\{Q\}) * Q) \,\Box_\Psi\, \kappa$ we know $w'' \models (\Gamma * (f : \{P\}\{Q\}) * Q) \,\Box_\Psi\, \kappa$. From these two facts and $w'' \models \mathsf{Htuple}(\Gamma * f : \{P\}\{Q\}, P, \Psi(f), Q)$, we know $w'' \models (\Gamma * f : \{P\}\{Q\}) * P \,\Box_\Psi\, (\Psi(f) \cdot \kappa)$. Therefore it suffices to show that $w'' \models \Gamma * f : \{P\}\{Q\}) * P$. Since Γ, $f : \{P\}\{Q\}$, and P are valid, this follows directly from $w' \models \Gamma * f : \{P\}\{Q\})P$.

Proved in Coq. □

Like *all* the proofs about sequential features, this proof did not mention anything about concurrency. The ability to prove complicated sequential results that are blind to the fact that they are running in a concurrent context is a major strength of our approach.

Combining proofs about functions

The major task for the CSL user is to prove $\Gamma \vdash \Psi$—that is, to prove pre- and postconditions for all of the functions in Ψ. For proving individual function bodies, he will use the rules of CSL. Here we show how to combine individual function bodies into a proof of $\Gamma \vdash \Psi$, which is a proof about the program as a whole.

$$\text{func-nil}\frac{}{\vdash (\Gamma \vdash \Psi : \mathbf{emp})}$$

$$\text{func-cons}\frac{\begin{array}{c}\vdash (\Gamma \vdash \Psi : \Gamma') \\ \vdash (\Gamma \subset (\mathbf{true} * \Gamma' * f : \{P\}\{Q\})) \\ \Gamma \vdash \{P\}\ \mathsf{call}\ f\ \{Q\}\end{array}}{\vdash (\Gamma \vdash \Psi : (\Gamma' * f : \{P\}\{Q\}))}$$

$$\text{func-believe}\frac{\vdash (\Gamma \vdash \Psi : \Gamma)}{\vdash (\Gamma \vdash \Psi)}$$

Figure 10.4: Building $\mathsf{believe}$

For this purpose we define a variant of the $\mathsf{believe}$ predicate, $\mathsf{believe}'$:

$$\begin{array}{rcl}\mathsf{believe}' & : & \mathsf{predicate} \to \mathsf{program} \to \mathsf{predicate} \to \mathsf{predicate} \\ & = & \lambda\Gamma.\ \lambda\Psi.\ \Gamma' \\ & & \quad \forall f, P, Q. \\ & & \qquad (\Gamma' \ \subset\ \mathbf{true} * f : \{P\}\{Q\})\ \subset \\ & & \qquad \rhd \mathsf{Htuple}(\Gamma, P, \Psi(f), Q)\end{array}$$

We use the notation $\Gamma \vdash \Psi : \Gamma'$ as shorthand for $\mathsf{believe}'\ \Gamma\ \Psi\ \Gamma'$. $\Gamma \vdash \Psi : \Gamma'$ means that all the functions in Γ' are proved correct with respect to their function bodies in Ψ. Since a function body may call all functions (not just those proved correct so far), they may use any of the specifications contained in Γ. This allows Γ' to be built up one function at a time.

To help the user prove $\Gamma \vdash \Psi$, we provide the rules in figure 10.4. The idea is that we will start with Γ' as **emp**, with rule func-nil. Then

we will add the functions in Γ to Γ' one at a time, with rule func-cons. The user will prove each function body using the rules of CSL. When every function in Γ has been added to Γ', then the two variants of $\mathsf{believe}$ are equivalent as expressed in rule func-believe.

> **Theorem 10.2**. The rules func-nil, func-cons, and func-believe are sound with respect to the semantic definitions of $\Gamma \vdash \Psi$ and $\Gamma \vdash \Psi : \Gamma'$.
>
> **Proof**. Rule func-nil is vacuously true since **emp** does not contain any functions. Rule func-believe is true immediately from the definitions. For rule func-cons, $\mathsf{believe}'$ is quantifying over all of the functions in Γ' plus the new function f. For all of the functions in Γ', we use the premise $\Gamma \vdash \Psi : \Gamma'$. For the new function f, we need to prove $\vdash \rhd \mathsf{Htuple}(\Gamma, P, \Psi(f), Q)$, which follows immediately since for any P, $(\vdash P) \to (\vdash \rhd P)$.
>
> Proved in Coq. □

The CSL user thus proves the bodies of all his functions with respect to Γ using the rules of CSL, and then uses the rules in figure 10.4 to prove $\Gamma \vdash \Psi$.

10.2.3 Concurrent rules

In section 4.8 we presented the concurrent rules of Concurrent Separation Logic in figure 4.1. We are now ready to prove them sound with

respect to the oracular semantics.

Theorem 10.3. The rules of Concurrent Separation Logic for the concurrent statements of Concurrent C minor as given in figure 4.1 are sound with respect to the semantics of the Hoare tuple defined in figure 10.3.

Proof. All of the concurrent statements of Concurrent C minor are handled by the consult partial function. Therefore, our job is to show that the preconditions of the rules guarantee that consult does not get stuck and that afterwards the postconditions hold. Since in the definition of safe we universally quantify over all oracles, our proofs must hold for any oracle.

The oracular consult partial function was defined in section 9.3 with three cases. In the first case, Ω-invalid, we handle invalid oracles, where there is no concurrent machine state compatible with the oracle. If the oracle is invalid, we loop endlessly. Since looping endlessly is safe regardless of the precondition, and since a postcondition is possible given an infinite loop, we can easily prove the Hoare rules if we have been given an invalid oracle by the universal quantification.

In both the second case, Ω-diverges, and the third case, Ω-steps, there is a concurrent machine state S consistent with the oracle. In both cases, we must take a concurrent step

from S to some subsequent state S'; this step can get stuck, and it is the first task of the CSL rule preconditions to ensure that it does not. The key difficulty in showing that you can take a step is showing that the state S' meets the consistency requirements given in section 8.4. This will be one of our major tasks in the case analysis below.

Assuming that this first task is done and therefore the preconditions are good enough to guarantee that the concurrent step relation is able to step from S to S', the consult function cannot get stuck. Therefore, the remaining task in the soundness proof is to prove the postcondition of the CSL rule. The oracle classically decides whether control will ever return to the thread by branching on the StepOthers relation.

In the first case, Ω-diverges, control will never return and the machine enters an infinite loop. As before, this makes it possible to prove any postcondition.

In the final case, Ω-steps, control returns after running the other threads. In this case it is necessary to use the precondition of the CSL rules to prove their postcondition by doing induction on the StepOthers relation.

With these preliminaries out of the way, we will briefly consider each rule in turn.

1. $\Gamma \vdash \{e \overset{\blacktriangle}{\mapsto} 0\}\ \mathsf{make\_lock}\, e\, R\ \{e \overset{\blacktriangle}{\rightsquigarrow} R\ *\ \mathsf{hold}\ e\, R\}$

   An examination of the make_lock case of the consult function defined in figure 8.2 will make clear that the consult will not get stuck and will guarantee the post-condtion. The concurrent machine will run cstep-seq to execute this instruction. Since the new lock is created locked, it is simple to show that the machine is still consistent since we only require complex properties to hold for unlocked locks.

   Proved in Coq. □

2. $\Gamma \vdash \{e \overset{\blacktriangle}{\rightsquigarrow} R\ *\ \mathsf{hold}\ e\, R\}\ \mathsf{free\_lock}\, e\ \{e \overset{\blacktriangle}{\mapsto} 0\}$

   An examination of the free_lock case of the consult function defined in figure 8.2 will make clear that the consult will not get stuck and will guarantee the post-condtion. The concurrent machine will run cstep-seq to execute this instruction. Since we have removed a lock from the total resouce map ϕ_{T}, it is easy to show consistency of the new state because we can quantify over one fewer lock.

   Proved in Coq. □

3. $\Gamma \vdash \{e \overset{\pi}{\rightsquigarrow} R\}\ \mathsf{lock}\, e\ \{e \overset{\pi}{\rightsquigarrow} R\ *\ \mathsf{tightly}\ R\}$

   The lock rule is by far the hardest rule to prove. It is relatively easy to show that the concurrent machine will run cstep-prelock to execute the instruction and

that the next state will be consistent. However, showing that the postcondition holds is quite challanging. Essentially the problem is that after we grab the lock with cstep-lock, we do not context switch. By using the consistency requirements on lock pools it is simple to show a postcondition of $\triangleright \mathsf{tightly}\ R$, but this is not good enough to prove our CSL rule, since we need $\mathsf{tightly}\ R$, which is stronger. Accordingly, we need to make a more complex induction, where we argue that since the time when the lock was unlocked, we have already context switched at least once, and therefore we have moved from $\triangleright \mathsf{tightly}\ R$ to $\mathsf{tightly}\ R$.

Proved in Coq. □

4. $R = (\mathsf{hold}\, e\, R * S) \rightarrow \Gamma \vdash \{\mathsf{tightly}\ R\}\ \mathsf{unlock}\, e\ \{\mathbf{emp}\}$

   An examination of the preconditions of the CSL rule will demonstrate that the concurrent machine will run cstep-unlock to execute this instruction. Proving the postcondition of **emp** is not difficult. The difficulty is in showing that the new machine state is consistent, and in particular that the rules for the lock pool are respected; they are since we know that R holds and that R is tight.

   Almost entirely proved in Coq. Proved in detail in appendix B.1. □

5. $\Gamma \vdash \{f : \{P\}\{Q\} * \mathsf{validly\ precisely}\ P\}\ \mathsf{fork}\ f\ \{f : \{P\}\{Q\}\}$

   An examination of the preconditions of the CSL rule will demonstrate that the concurrent machine will run cstep-fork to execute this instruction. It is not difficult to show that the new state is consistent or to prove the postcondition.

   The fork rule is relatively simple to prove because our definition of safety is quite uncaring about the behavior of other threads (if they go wrong, we are immediately safe due to case Ω-diverges). Therefore, in the soundness proof of the CSL fork rule, we are only concerned about the parent thread. When we connect the oracle semantics to the concurrent semantics in section 10.3, we will have to prove that the has been properly started in the preservation theorem.

   Proved in Coq. □

Now that we have proved all of the rules of CSL sound with respect to the oracular semantics of Concurrent C minor, half of our soundness task is done. The other half, covered in the remainder of this chapter, is to connect the oracular semantics to the concurrent semantics, which we do with progress and preservation theorems.

$$\text{c\_isafe\_level}\frac{\mathsf{level}\ \phi_{\mathsf{mp}} = 0}{\Psi \vdash \mathsf{c\_isafe}\ (\mho, \vec{\theta}, \mathcal{L}, \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m)}$$

$$\text{c\_isafe\_sched}\frac{\nexists \theta.\ \vec{\theta_i} = \theta}{\Psi \vdash \mathsf{c\_isafe}\ (i :: \mho, \vec{\theta}, \mathcal{L}, \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m)}$$

$$\text{c\_isafe\_step}\frac{\Psi \vdash S \implies S'}{\Psi \vdash \mathsf{c\_isafe}\ S}$$

$$\text{cstepstar\_0}\frac{}{\Psi \vdash S \implies^* S}$$

$$\text{cstepstar\_S}\frac{\Psi \vdash S \implies S' \quad \Psi \vdash S' \implies^* S''}{\Psi \vdash S \implies^* S''}$$

$$\Psi \vdash \mathsf{csafe}\ S \ = \ \forall S'.\ \ \Psi \vdash S \implies^* S' \ \rightarrow \ \Psi \vdash \mathsf{c\_isafe}\ S'$$

Figure 10.5: Concurrent safety

10.3 Soundness of the Oracular Approach

Now we connect the oracular semantics to the concurrent semantics. First we define the notion of *concurrent safety* in figure 10.5. We say that a concurrent machine state S is *concurrently immediately safe* (just referred to as immediately safe when the context is clear) in three situations.

First, case c_isafe_level, the concurrent machine is immediately safe if the resource maps inside it have approximation level 0. Recall that since all resource maps in S must join together due to the consistency

requirements on S, all those resource maps must have the same approximation level, and therefore it is sufficient to test the level of the alloc pool. Just as in section 10.1.1, we only provide guarantees about correct behavior for an arbitrarily large but finite amount of time.

Second, case c_isafe_sched, the concurrent machine is immediately safe if the schedule is trying to pick a nonexistent thread. Since we quantify over all schedules we do not have to worry that some of the schedules we quantify over wish to pick nonexistent threads; we also quantify over all schedules that pick only existing threads.

Third, case c_isafe_step, the concurrent machine is immediately safe if it can take a step. That is, "immediately safe" means "not stuck".

We define the $\Longmapsto^*$ relation as the composition of the concurrent step relation in the usual way with cases cstepstar_0 and cstepstar_S. We define csafe in the usual way: for any state S' reachable from S, S' is immediately safe.

Our major goal will be to prove concurrent safety from oracular safety. Due to the nature of the concurrent step relation, concurrent safety is actually quite a strong property. Recall from section 8.5 that the cstep-unlock rule requires that a lock's resource invariant be true before unlocking. This means that concurrent safety implies that all resource invariants are obeyed. Since outside programs can only communicate with a program via locks, this implies that all outside communication is done correctly.

We will prove concurrent safety with a progress theorem and preser-

$$\text{safe-as\_never}\ \frac{(\Omega, w, \hat{\kappa}) \overset{i}{\propto} S \qquad \nexists S'.(\Psi \vdash \mathsf{StepOthers}\ i\ S\ S')}{\Psi \vdash \mathsf{safe\text{-}as}\ i\ (\Omega, w, \hat{\kappa})}$$

$$\text{safe-as\_eventually}\ \frac{\begin{array}{c}(\Omega, w, \hat{\kappa}) \overset{i}{\propto} S \\ \Psi \vdash \mathsf{StepOthers}\ i\ S\ S' \\ (\Omega', w', \mathsf{Krun}\,\kappa) \overset{i}{\propto} S' \\ w' \models \mathsf{safe}_{\Psi}\ \kappa\end{array}}{\Psi \vdash \mathsf{safe\text{-}as}\ i\ (\Omega, \sigma, \hat{\kappa})}$$

$$\begin{array}{rcl}\Psi \vdash \mathsf{all\text{-}threads\text{-}safe}(S) & = & \forall i, \Omega, \sigma, \hat{\kappa}. \\ & & \quad (\Omega, \sigma, \hat{\kappa}) \overset{i}{\propto} S \ \rightarrow \\ & & \quad \Psi \vdash \mathsf{safe\text{-}as}\ i\ (\Omega, \sigma, \hat{\kappa})\end{array}$$

Figure 10.6: Progress invariant

vation theorem. As usual, the progress theorem will define an invariant sufficient to guarantee that the concurrent machine is not stuck and the preservation theorem will guarantee that the invariant is preserved as the machine computes.

10.3.1 Progress

In figure 10.6 we define the property safe-as i. Informally, a concurrent continuation $(\Omega, \sigma, \hat{\kappa})$ is safe-as i if, supposing it is the ith thread of the concurrent machine consistent with oracle Ω, then if this thread is ever ready to run then it will be oracularly safe. There are two cases: in the first, case safe-as_never, the thread will never again be selected to run sequential code again; in the second, case safe-as_eventually, the thread

will eventually be selected to run again, and when it is selected, it will be safe. Recall that to be safe we must quantify over all oracles, so we do not need to worry about the actual oracle Ω'.

We define $\Psi \vdash \mathsf{all\text{-}threads\text{-}safe}(S)$ to mean that each projection of S will be oracularly safe the next time it is ready and selected. Together with the consistency requirements on the concurrent machine state as explained in section 8.4, this is enough to prove progress.

Theorem 10.4 (Progress). If $\Psi \vdash \mathsf{all\text{-}threads\text{-}safe}(S)$, then S is immediately safe.

Proof. Part of taking a step in the concurrent machine is proving that the next concurrent machine has the consistency property. More than half of the difficulty in the theorem is proving consistency of the next concurrent machine state. These properties would normally be proved in the preservation theorem, but due to the use of dependent types to guarantee consistency part of the preservation theorem is proved in the progress theorem.

By case analysis on the thread whose thread-id is at the head of the schedule. The initial case is on the concurrent control of the thread.

1. The concurrent control is $\mathsf{Klock}\, v\, \kappa$, so the thread is waiting on lock v. If $m(v) = 0$, *i.e.* the lock is locked, we will execute step cstep-nolock and keep waiting.

Since nothing has changed it is easy to show that the new state is consistent. If $m(v) = 1$, *i.e.*, the lock is unlocked, then we will execute cstep-lock to grab the lock. The consistency requirements are easy to satisfy since the quantification over the lock pool must deal with one fewer unlocked lock.

2. The concurrent control is $\mathsf{Krun}\ \kappa$, so the thread is runnable. We have two cases. Since thread i is at the head of the schedule, we know $\Psi \vdash \mathsf{StepOthers}\ i\ S\ S$. By all-threads-safe we know safe-as i, where i is the thread at the head of the schedule; there are two possibilities, but we know that it cannot be safe-as_never, since thread i is ready to run. Therefore it must be safe-as_eventually, and thread i must be oracularly safe. If it is safe because the thread has reached Kstop then we execute cstep-texit. Otherwise, the thread must be able to take an oracular step. There are three possibilities:

   a) If the oracle step is in case Core Step, then κ must be a sequential instruction, and we know that the machine steps sequentially. In this case we execute cstep-seq. We facts about the sequential step relation to prove that the next state is consistent.

   b) If the oracle step is in case Oracle Step, then we know that a consult succeeded on the oracle step.

$$\mathsf{prkind}\ l\ k \quad = \quad \lambda(\rho, \phi, m).\ \ \mathsf{rkind}(\phi\ @\ l) = \mathsf{Some}\ k$$

$$\begin{array}{l}\Psi \vdash \mathsf{all\text{-}funs\text{-}spawnable}(S) \ = \\ (\rho_0, \phi_{\mathsf{T}}, m) \ \models \ \exists \Gamma.\ \ (\Gamma \ \vdash \Psi) \wedge \\ \qquad\qquad\qquad\qquad \big(\forall f.\ \mathsf{prkind}\ f\ \mathsf{kFUN} \ \rightarrow \\ \qquad\qquad\qquad\qquad\quad \exists P, Q.\ f : \{P\}\{Q\} \ \wedge \\ \qquad\qquad\qquad\qquad\qquad\quad \Gamma \subset \mathbf{true}\ f : \{P\}\{Q\}\big)\end{array}$$

Figure 10.7: Preservation invariant

There are three possibilities:

i. We can prove that we are not in case Ω-invalid.

ii. We can prove that we are not in case Ω-diverges.

iii. In case Ω-steps, we have as a premise that the concurrent step relation takes a step (recall that proving that it takes a step is in fact the major task of the CSL rules).

c) We can prove that the oracle step cannot be in case Oracle Diverges.

Proved in Coq. □

10.3.2 Preservation

The preservation theorem is more complex due to the existence of forks, since we need to know that the child will be safe if its precondition is satisfied. To guarantee that the child will be safe, we require an additional invariant, all-funs-spawnable, given in figure 10.7.

First we define the predicate version of the rkind function, prkind, which maps resources to resource kinds. Recall from section 8.4.6 that rkind is useful as an antecedent with unrestricted (dangerous) implication, and we will use prkind here for this purpose. If

$$(\rho, \phi, m) \models \mathsf{prkind}\ l\ k$$

then ϕ @ l has resource kind k.

Then we define $\Psi \vdash \mathsf{all\text{-}funs\text{-}spawnable}(S)$. Recall that for any concurrent machine state S there exists a total resource map ϕ_T that is the join of all of the resource maps in S. all-funs-spawnable states that there exists a predicate Γ that is compatible with the program Ψ and that for any location f with kind kFUN, there exists a precondition P and postcondition Q such that

$$(\rho_0, \phi_\mathsf{T}, m) \models f : \{P\}\{Q\}$$

and that Γ implies, *i.e.* all the functions in ϕ_T are contained in Γ. Together this means that all of the functions in Ψ are conservatively approximated by their pre- and postconditions in Γ. We use the prkind construction and quantify existentially instead of universally over the precondition P and postcondition Q here for the same reason as in section 8.4.6. The predicate does not use the locals, so we are able to apply it to an empty locals ρ_0; it also does not use the memory m, but since we have m easily available in S we just use it.

The predicate Γ is constructed by the user using the rules of CSL as explained in section 10.2.2.

Lemma 10.1. If $\Psi \vdash \mathsf{all\text{-}funs\text{-}spawnable}(S)$, and $\Psi \vdash S \Longmapsto S'$, then $\Psi \vdash \mathsf{all\text{-}funs\text{-}spawnable}(S')$.

Proof. Proved by case analysis on the step relation.

In all cases except for cstep-seq, the total resource map ϕ_{T} does not change except perhaps becoming more approximate (resources are transfered from one part of the machine to another, but this does not change the total resource map). In step cstep-seq, the total resource map does not get larger or add new functions. The use of rkind ensures that our use of "dangerous" implication is sound even as the total resource map becomes more approximate in much the same way as in section 8.4.6.

Proved in Coq[3]. □

Lemma 10.2. If $\Psi \vdash \mathsf{all\text{-}threads\text{-}safe}(S)$, $\Psi \vdash \mathsf{all\text{-}funs\text{-}spawnable}(S)$, and $\Psi \vdash S \Longmapsto S'$, then $\Psi \vdash \mathsf{all\text{-}threads\text{-}safe}(S')$.

Proof. There are three basic cases. The first is that we need to show that the thread that was just selected and run is

[3]As of the writing of this thesis, the Coq proof of the preservation theorem is currently broken due to maintenence and cleaning elsewhere in the system, particularlly related to the function pool ϕ_{fp}; however everything here claimed to be proved in Coq has been proved in Coq at one time.

still safe. The second is that we need need to show that one of the threads that was not selected is still safe. The third only occurs with the fork rule, and it is when we need to show that the new child thread is safe.

1. The thread that was just selected and run is still safe. This is quite simple: if one is safe and takes a step one is still safe.

   Proved in Coq. □

2. A thread that was not selected to run is still safe. There are two possibilities: either the thread will never run, in which case it is safe by case safe-as_never, or it must have been in case safe-as_eventually, in which case we can prove that it is safe by induction on the StepOthers judgement.

   Proved in Coq. □

3. A new child that was just spawned is safe. In this case we use the fact that all functions are spawnable to show that since the child's precondition was satisfied it is safe.

   Proved in detail in appendix B.2. □

This completes the proof of lemma 10.2. □

Theorem 10.5 (Preservation). If $\Psi \vdash \mathsf{all\text{-}threads\text{-}safe}(S)$, $\Psi \vdash \mathsf{all\text{-}funs\text{-}spawnable}(S)$, and $\Psi \vdash S \Longmapsto S'$, then $\Psi \vdash \mathsf{all\text{-}threads\text{-}safe}(S')$ and $\Psi \vdash \mathsf{all\text{-}funs\text{-}spawnable}(S')$.

Proof. By lemmas 10.1 and 10.2.

Proved in Coq. □

10.3.3 Safety

Now that we have both the progress and preservation theorems, we are ready to prove concurrent safety from oracular safety.

Theorem 10.6 (Safety). If $\Psi \vdash \mathsf{all\text{-}threads\text{-}safe}(S)$ and $\Psi \vdash \mathsf{all\text{-}funs\text{-}spawnable}(S)$, then $\Psi \vdash \mathsf{csafe}\ S$.

Proof. By induction on the $\Longmapsto^*$ relation and the progress and preservation theorems.

Proved in Coq. □

We are nearly done. Recall that a program in Concurrent C minor begins with a call to a special `main` function.

Corollary 10.1. For any schedule υ and arbitrary stratification level for the initial resource maps, if the initial thread call `main`() is oracularly safe and all functions are spawnable, then the concurrent machine is safe.

Proof. Follows directly from the safety theorem. In the initial state S there is only one thread, so it is easy to prove $\Psi \vdash \mathsf{all\text{-}threads\text{-}safe}(S)$.

By the definition of the Hoare tuple we can prove the initial call to the `main` function safe on any world that satisfies its precondition. Therefore we have the following theorem, expressed as a deductive rule:

$$\frac{\begin{array}{c}\Gamma \vdash \Psi \qquad\qquad (\rho_0, \phi_{\mathsf{fp}}, m) \models \Gamma \\ (\rho_0, \phi_{\mathsf{mp}}, m) \models \mathsf{allocpool} \qquad \forall l.\ \mathcal{L}_0(l) = \mathsf{None} \\ \Gamma \subset \mathbf{true} * \texttt{main} : \{P\}\{Q\} \\ \theta = (\rho, \phi, \mathsf{Krun}\,(\texttt{call main}) \cdot \mathsf{Kstop}) \\ (\rho, \phi, m) \models \mathsf{validly}\ P\end{array}}{\Psi \vdash \mathsf{csafe}\ (\mathcal{U}, \theta :: \texttt{nil}, \mathcal{L}_0, \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m)}$$

Theorem 10.7. That is, if

1. there exists a Γ that gives function pre- and postconditions for the functions in program Ψ
2. the function pool ϕ_{fp} forces Γ
3. we have an alloc pool ϕ_{mp}
4. we start with an empty initial lock pool $\mathcal{L}_0$
5. Γ includes a precondition P for `main`
6. we have exactly one thread in the machine, which is a call to `main` and has an inital set of locals ρ and resource map ϕ

7. the locals ρ, resource map ϕ, and memory m force `main`'s precondition P

then the unerased concurrent machine state $(\mathcal{U}, \theta :: \texttt{nil}, \mathcal{L}_0, \phi_{\mathsf{mp}}, \phi_{\mathsf{fp}}, m)$ is safe.

Proof. Proved from corollary 10.1. □

How hard will this theorem be for the CSL user to use? Premise 1 is developed by the user using the rules of CSL as explained in section 10.2.2. Premise 2 is simple: a function pool of arbitrary stratification can be built from Γ by the user. Premise 3 is simple: an alloc pool of arbitrary stratification can be built. Premise 4 is simple: the empty lock pool is easy to construct. Premise 5 is simple as long as the user does not forget the `main` function. Premise 6 is simple; the single thread is easy to construct. The difficulty of premise 7 depends entirely on how difficult the user wishes to make the precondition of main to satisfy; we point out that **emp** is very easy to satisfy.

Therefore, we have shown that if a user proves his program sound with CSL, then assuming that his `main` function's precondition is satisfiable his program will be safe when run on the concurrent machine. Our only remaing task is to connect to the erased concurrent operational semantics from chapter 5.

Corollary 10.2. If a user proves his program sound with CSL, then assuming his `main` function's precondition is sat-

isfiable then his program will be safe when run on the erased concurrent machine.

Proof. Directly from the previous theorem and the erasure theorem of chapter 8. □

Note: No parallel decomposition lemma. Unlike in previous work [Bro07], we do not require a parallel decomposition lemma. We can avoid such a lemma because

1. We put resource maps into the concurrent operational semantics.

2. In the progress lemma we prove that the next state obeys the consistency requirements, which guarantees that the threads have disjoint resource maps.

3. In the store operation in the sequential semantics we get stuck if we try to write to memory we do not own.

4. We prove the safety of the parent in the CSL fork rule.

5. We prove the safety of the child in the preservation lemma.

In previous semantics the parallel decomposition lemma was a source of significant difficulty and our ability to avoid having to reason about all of these issues simultaneously is a demonstration of the benefits of our modularity.

10.4 Conclusion

In chapter 3 we introduced Concurrent C minor. In chapter 4 we gave Concurrent C minor an axiomatic semantics, Concurrent Separation Logic, and in chapter 8 we gave Concurrent C minor an operational semantics. We have now proved our logic sound with respect to our operational semantics.

Chapter 11

Conclusions and Future Work

We have proved the soundness of a sophisticated Concurrent Separation Logic with first-class locks and threads with respect to the concurrent operational semantics of Concurrent C minor. Our soundness proof is largely implemented in Coq in a highly modular way, so that the actions of other threads do not complicate the parts of the soundness proof about sequential features and the complexities of sequential control flow do not complicate the parts of the proof about the concurrent features. Moreover, we have a strategy for applying our techniques to the CompCert certified compiler, so that in the future we will be able to have an end-to-end result: proved properties of concurrent source programs will be true of the code that executes on the machine.

In chapter 3 we presented the Concurrent C minor language and

gave an example program in section 3.2. Then in chapter 4 we developed a new concurrent separation logic and demonstrated its power by using it to verify an example program. In chapter 5 we gave Concurrent C minor a formal erased concurrent operational semantics.

In chapter 6 we presented engineering techniques that allow for significant modularization, particularly in the context of a compiler with multiple intermediate languages.

In chapter 7 we developed a new modal substructural logic and showed how to reason about complex parts of the underlying model by reasoning in the logic.

In chapter 8 we developed an unerased concurrent operational semantics that was easier to reason about than the erased concurrent operational semantics from chapter 5. We proved an erasure theorem that showed that the unerased semantics was a conservative approximation to the erased semantics.

The concurrent operational semantics of chapter 8 was not easy to use for reasoning about the sequential features of the Concurrent C minor language, so in chapter 9 we developed a thread-local oracle semantics for it.

Finally, in chapter 10 we developed a new modal Hoare tuple using the oracle semantics and showed how to use it to prove the rules of CSL sound. We also showed how to combine results from the oracular semantics into a result on the concurrent semantics with a progress and preservation lemma, thereby proving the soundness of Concurrent

Separation Logic with respect to the concurrent operational semantics of Concurrent C minor.

11.1 State of the Coq Development

As we have indicated in the text, the Coq development is incomplete. The tasks that remain are:

1. Integrate the cleaner definitions for joinable types and shares that were presented in chapter 4.

2. Add the function pool to the concurrent machine state. We are attempting to develop a better solution to the problem of global and immutable data in separation logic, and since the function pool is a bit of a hack we have held off implementing it in Coq until we can study the other alternatives in more detail.

3. Finish the modularization work presented in chapter 6. We have been modularizing the proof incrementally.

4. Do clean-ups as noted in the text.

5. Finish the proof of the CSL unlock rule.

6. Finish the preservation proof.

We developed a slogan when working on the proof, which was "As expected, it took longer than expected." We also agree with the obser-

vation of Leroy, who noted that "Building [proof] scripts is surprisingly addictive, in a videogame kind of way" [Ler06].

11.2 Future work

One of the major benefits of the project has been that it has a number of promising tasks for future work. These include:

1. Modifying the CompCert compiler to handle concurrency. Our plan is to create an oracle semantics for each intermediate language of the compiler by applying our extension and then update the correctness proofs to show that the compiler preserves the behavior of each oracular semantics.

2. Developing techniques to reason about lock-free algorithms of various kinds. In practice, many of the concurrent algorithms used do not use locks, instead relying on various lock-free techniques such as atomic loads and stores. One goal is to see how we can apply our oracle semantics towards reasoning about these kinds of algorithms.

3. Reasoning in the presence of weak memory models. Modern processors execute memory accesses out of program order in a way that can change the semantics of concurrent programs. We wish to reason about this behavior so that we can prove that our semantics is sound on real processors.

4. Exposing implication to end-user by improving dependent/stratified model. Because users of Hoare logics are accustomed to having full implication, it would be useful to be able to improve our assertion models to expose full implication to the CSL user.

5. Developing techniques to better reason about globals and shared immutable data in separation logic. We had a suprisingly difficult time creating global values. We added the function pool to solve this problem, but we want to design a more elegant technique.

11.3 Concluding thoughts

We have developed a soundness proof for a powerful new Concurrent Separation Logic with first-class locks and threads. Along the way, we developed a modal substructural logic; designed a new style of concurrent operational semantics that did bookkeeping to guarantee behavior; designed a new oracle semantics that presented a thread-local view of a concurrent machine and that allowed for the re-use of metaproofs of sequential language features; and provided a new modal definition of the Hoare tuple. Our Coq proofs have been almost completed, giving a high degree of assurance, and are designed to support the modification of the CompCert compiler, thereby supporting the goal of providing an end-to-end guarantee about the behavior of concurrent programs.

Appendix A

A Miniature Model in Coq

A.1 Headers

```
(* Aquinas Hobor *)

Require Import List.

Parameter address : Type.
Parameter value : Type.
Parameter share : Type.
Parameter kind : Type.

Definition memory : Type := address -> value.
```

A.2 Stratified Model

```
Section Stratified_Model.

Inductive res_n' (inv : Type) : Type :=
 | NO'
 | YES': kind -> share -> (list inv) -> res_n' inv.

Definition rmap_n' (inv : Type) : Type :=
 address -> res_n' inv.

Fixpoint inv_n (n: nat) : Type :=
 match n with
  | O => unit
  | S n => ((inv_n n) * (rmap_n' (inv_n n) * memory -> Prop))%type
 end.

Definition res_n (n: nat) : Type :=
 res_n' (inv_n n).

Definition rmap_n (n: nat) : Type :=
 rmap_n' (inv_n n).

End Stratified_Model.
```

A.3 Dependent Model

```
Section Dependent_Model.


Inductive resource' : Type :=
 res_Level : forall (n : nat),  res_n n -> resource'.
Definition resource := resource'.


Inductive rmap' : Type :=
 rm_Level : forall (n : nat), rmap_n n -> rmap'.
Definition rmap := rmap'.


Definition predicate : Type := (rmap * memory) -> Prop.


End Dependent_Model.
```

A.4 Private definitions relating public to private

```
Section Private_Defs.

Definition level (rm: rmap) : nat :=
 match rm with rm_Level n _ => n end.

Fixpoint stratify (n : nat) (P : predicate) : inv_n n :=
 match n as n return inv_n n with
  | O => tt
  | S n => (stratify n P,
            fun v =>
             match v with (phi_n, mem) =>
              P (rm_Level n phi_n, mem)
             end)
 end.

End Private_Defs.
```

A.5 Public interface to acquire resource

```
Section Public_Interface.

Definition resource_at (rm: rmap) (addr: address) : resource :=
 match rm with rm_Level n rm_n => res_Level n (rm_n addr) end.

Definition NO (rm: rmap) : resource :=
 res_Level (level rm) (NO' (inv_n (level rm))).

Definition YES (rm: rmap)
               (k: kind)
               (sh: share)
               (P : list predicate) : resource :=
 res_Level (level rm) (YES' (inv_n (level rm))
                            k
                            sh
                            (map (stratify (level rm)) P)).

End Public_Interface.
```

Appendix B

Proofs

B.1 The unlock rule

Lemma. The unlock rule of CSL is sound.

Proof. As discussed in section 10.2.3, we first need to show that the precondition of the CSL lock rule guarantees that the concurrent step relation will not be stuck in cases Ω-diverges and Ω-steps. The precondition of the lock rule is tightly R, and in addition we have the fact that $R = (\mathsf{hold}\ e\, R\ *\ S)$. We need to show that these guarantee that we can take a step with the cstep-unlock rule of the concurrent machine. We are guaranteed that we are at the head of the scheduler by the StepOthers relation, which handles the first premise. Since we know tightly R, and $R = (\mathsf{hold}\ e\, R * S)$, we know tightly $(\mathsf{hold}\ e\, R * S)$. Since hold is tight, we know hold $e\, R$. By the consistency requirements on the concurrent machine state, if this permission is in a thread's resource

map, the lock must be locked, so $m(e)$ must be equal to 0, which handles the second premise. The third premise, involving the new memory m' is true by construction. The forth premise, which says that we can split off a world ϕ_{lock} from ϕ, is always true, although in most cases ϕ_{lock} is not uniquely determined. The sixth premise, which is that we have $\mathsf{hold}\ e\, R$, has already been demonstrated. The seventh premise, which is that $\mathsf{tightly}\ R$ holds on the resource map ϕ_{lock}, is guaranteed since it held on the resource map ϕ. Moreover, since all tight predicates are precise, this guarantees that ϕ_{lock} is unique. The eigth premise, which is that we can add the newly unlocked lock to the lock pool, is true by constrction and because we can prove that it was not in the lock pool before, since due to the consistency requirements lock pools can only hold unlocked locks and prior to this point the lock was locked. The nineth premise, the construction of new thread list, is true by construction. The tenth premise, that we can context switch, is true because as long as we have remaining stratification levels we can context switch, and if we had run out of stratification levels then we would have been immediately safe. Now that we have proved all of the premises to the cstep-unlock rule, we must prove that the new concurrent machine state S' is consistent. From section 8.4 we must prove six properties. First, we must show that there exists a new total resource map. The lock rule does not add or remove from the total resource map, instead simply moving parts of the resource map from a thread to the lock pool. It does age the total resource map once, but since we had at least one level

of stratification this is possible to do. Second, we must show that the scheduler is still longer than the remaining stratification; since context switch removes exactly one item from the scheduler and ages the worlds exactly once this is simple. Third, we must prove that if any threads are waiting on a memory location then that location is a lock. Since we have not modified any of the other threads, and are not waiting on a lock ourselves, and have not freed any locks, this is simple. Fourth, we must have a well-formed alloc pool; since we have not changed it this is trivial. Fifth, we must have a well-formed function pool; since we have not changed it this is trivial. Sixth and last, we must show that we still have a well formed lock pool. As discussed, the property P3 behaves correctly as the world ages; since we have not touched any of the other locks in the lock pool, and the consistency requirements on the concurrent machine state ensures that the newly unlocked lock's memory locations do not overlap with any previously unlocked lock's memory locations, all of the other locks in the lock pool are still sound. For the lock we are unlocking, we need to show that there exists a lock invariant for it—but this is easy, since we have that as a premise—and that the invariant holds tightly on the lock's resource map; however this is just premise seven from the cstep-unlock rule. Therefore, we have a consistant machine state and so are able to take a step with the cstep-unlock rule. Now we have to prove the postcondition of the rule. Fortunately this is quite easy. If the concurrent machine never returns control, case Ω-diverges, we can prove any postcondition. If the concurrent machine

does return control, case Ω-steps, we need only satisfy the postcondition of **emp**, and since we have given away our precondition by putting it into the lock pool, it is easy to satisfy.

B.2 Forking a child

Lemma. The child resulting from the fork is safe-as i.

Proof. By all-funs-spawnable, we know that there exists a Γ which believe can connect to the program Ψ. By the precondition to the cstep-fork rule, we know $f : \{\mathsf{validly\ precisely}\ P\}\{Q\}$ for some precondition P and postcondition Q. Since this implies that $\mathsf{rkind}(\phi_{\mathsf{T}}\ @\ f) = \mathsf{Some\ kFUN}$, by the second half of all-funs-spawnable we know that there exists some (possibly different) precondition P' and postcontcondition Q' such that $f : \{P'\}\{Q'\}$ and $\Gamma \subset \mathbf{true} * f : \{P'\}\{Q'\}$. By YES inversion we know that $\triangleright\, \mathsf{validly\ precisely}\ P = \triangleright\, P'$. By the definition of believe, we know $\triangleright\, \mathsf{Htuple}(\Gamma, P', \Psi(f), Q')$. By the precondition to the cstep-fork rule, we know validly precisely P. Since any valid predicate is necessary, we know $\Box\, \mathsf{validly\ precisely}\ P$, and since $\Box\, P \vdash \triangleright\, P$, we know $\triangleright\, \mathsf{validly\ precisely}\ P$. By the YES inversion above this is equal to $\triangleright\, P'$. After we age the world as part of the context switch in the cstep-fork rule, we will have P' and $\mathsf{Htuple}(\Gamma, P', \Psi(f), Q')$. By the definition of Htuple and since we have believe Htuple $\Gamma\ \Psi$, by fold-unfold we know

$$\begin{aligned} \forall \kappa.\ \ & \mathsf{pguard}\ \Psi\ (\Gamma * Q')\ \kappa\ \Rightarrow \\ & \mathsf{pguard}\ \Psi\ (\Gamma * P')\ (\Psi(f) \cdot \kappa). \end{aligned}$$

Recall that the control Kstop was immediately safe. Therefore, for all P, pguard Ψ P Kstop. Thus, we have

$$\mathsf{pguard}\ \Psi\ (\Gamma * P')\ (\Psi(f) \cdot \mathsf{Kstop}).$$

By the cstep-fork rule, the child will be started with initial control[1] call $f \cdot$ Kstop and an initial world satisfying $\triangleright$ validly precisely P, *i.e.*, P'. The key action of the call statement is to step from this control to the control $\Psi(f) \cdot$ Kstop. Since the function pool ϕ_{fp} is passed into each function as it runs, and since the function pool contains all of the functions in Γ, at the time when the thread is running we will know $\Gamma * P'$. By the definition of pguard, we will thus know psafe Ψ $\Psi(f) \cdot$ Kstop, which means that the child is safe-as i.

[1] We remove the function arguments for simplicity.

Bibliography

[AAV02] Amal Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *17th Annual IEEE Symp. on Logic in Computer Science*, pages 75–86, June 2002.

[AAV03] Amal Ahmed, Andrew W. Appel, and Roberto Virga. An indexed model of impredicative polymorphism and mutable references. http://www.cs.princeton.edu/~appel/papers/impred.pdf, January 2003.

[AB07] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step C minor. In *20th International Conference on Theorem Proving in Higher-Order Logics (TPHOLs 2007)*, 2007.

[ABF+05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the

masses: The poplmark challenge. In *18th International Conference on Theorem Proving in Higher-Order Logics (TPHOLs 2005)*, 2005.

[Ahm04] Amal J. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, Princeton, NJ, November 2004. Tech Report TR-713-04.

[AM01] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, September 2001.

[AMRV07] Andrew W. Appel, Paul-Andre Melliès, Christopher D. Richards, and Jerôme Vouillon. A very modal model of a modern, major, general type system. In *Proc. 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 109–122, January 2007.

[App01] Andrew W. Appel. Foundational proof-carrying code. In *Symp. on Logic in Computer Science (LICS '01)*, pages 247–258. IEEE, 2001.

[App06] Andrew W. Appel. Tactics for separation logic. unpublished manuscript, 2006.

[App08] Andrew W. Appel. A core resource calculus for higher-order separation logics. unpublished manuscript, 2008.

[BCOP05] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL '05: 32nd ACM Symp. on Principles of Programming Languages*, pages 259–270, 2005.

[BDH+08] C. J. Bell, Robert Dockins, Aquinas Hobor, Andrew W. Appel, and David Walker. Comparing semantic and syntactic methods in mechanized proof frameworks. In *Proceedings of the 2nd International Workshop on Proof-Carrying Code (PCC 2008)*, 2008.

[Boy03] J. Boyland. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium*, pages 55–72, 2003.

[Bro07] Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1):227–270, May 2007.

[COY07] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS 2007 IEEE Symposium on Logic in Computer Science*, 2007.

[DAH08] Robert Dockins, Andrew W. Appel, and Aquinas Hobor. Multimodal separation logic for reasoning about opera-

tional semantics. In *Proceedings of the 24th Conference on the Mathematical Foundations of Programming Semantics (MFPS 2008)*, 2008.

[Dij68] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. New York, NY, 1968.

[Doc08] Robert Dockins. Share models, 2008. Private conversation & Coq implementation.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967.

[GBC+07] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *Proceedings 5th Asian Symposium on Programming Languages and Systems (APLAS'07)*, 2007.

[GBCS07] Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. Thread-modular shape analysis. In *PLDI '07: 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In

17th Annual International Symposium on Computer Architecture (ISCA 1990), pages 15–26, May 1990.

[Got08] Alexey Gotsman. private conversation, 2008.

[Han93] P. Brinch Hansen. Monitors and concurrent pascal: A personal history. In *2nd ACM Conference on the History of Programming Languages*, pages 1–35, 1993.

[HAZ08a] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Proceedings of the 17th European Symposium on Programming (ESOP 2008)*, pages 353–367, 2008.

[HAZ08b] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic (extended version). Technical Report TR-825-08, Princeton University, June 2008.

[Hoa69] C A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):578–580, October 1969.

[Hoa74] C A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 17(8):666–677, August 1978.

[IO01] Samin Ishtiaq and Peter O'Hearn. BI as an assertion language for mutable data structures. In *POPL 2001: The 28th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 14–26. ACM Press, January 2001.

[Jon83] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[LB08] Xavier Leroy and Sandrine Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54, 2006.

[LPP05] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *IEEE Conference on Software Engineering and Formal Methods*, 2005.

[Man08] William Mansky. Automating separation logic for concurrent c minor. Senior Thesis, Princeton University, 2008.

[Moo89] J. Strother Moore. A mechanically verified language implementation. In *Journal of Automated Reasoning, 5(4)*, pages 461–492, 1989.

[Nec97] George C. Necula. Proof-carrying code. In *Procedings of the 24th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '97)*, pages 106–119, January 1997.

[OG76] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. In *Acta Informatica, 6*, pages 319–340, 1976.

[O'H] Peter O'Hearn. Further reading on local reasoning and separation logic. www.dcs.qmul.ac.uk/∼ohearn/furtherreading.html. Web page fetched June 30th, 2008.

[O'H07] Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1):271–307, May 2007.

[Par05] Matthew J. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.

[Rey02] John Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002: IEEE Symposium on Logic in Computer Science*, pages 55–74, July 2002.

[Ric08] Christopher D. Richards. *The Approximation Modality in Models of Higher-Order Types*. PhD thesis, Princeton University, 2008. Unpublished Draft.

[Sch06] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[VP07] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR 2007*, pages 256–271, 2007.