The Dissertation Committee for Ian Anthony Wehrman
certifies that this is the approved version of the following dissertation:

Weak-Memory Local Reasoning

Committee:

Warren A. Hunt, Jr., Supervisor

J Strother Moore, Supervisor

Josh Berdine

E. Allen Emerson

Donald S. Fussell

C. A. R. Hoare

Weak-Memory Local Reasoning

by

Ian Anthony Wehrman, B.S.; M.S.C.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2012

For my mother, Sally Marie Lorino, from whom I learned to learn.

Acknowledgments

This dissertation would not have been possible without the help of many people. I thank Warren Hunt, who has been a wellspring of support in favor of my doctoral degree. I am certain that I would not have not completed this project without his encouragement and generous technical, moral and financial support. I also thank Josh Berdine, who has been a true intellectual mentor to me. If I have contributed anything to our field of interest then for that I have him to thank. I am privileged to have had the opportunity to work with and learn from Tony Hoare, whose insight into the science of programming and the broader scientific process has been invaluable. I am in debt to J Moore for accepting me as an advisee at a difficult moment and for guiding me back to a successful path. I thank also Allen Emerson and Don Fussell for their participation on my dissertation committee. Finally, I warmly acknowledge the support of my family and friends. My mother has always been my unflagging champion; I do not know how I could have made it without her love through good and bad times. While in Austin, my wonderful friends—in particular Joel Brandt, Richard Chang, Benjamin Delaware, Anne Proctor and Chelsea Weathers—have broadened my horizons and brightened my life. Their compassion, camaraderie and loyalty means more to me than they likely know.

Ian Anthony Wehrman

The University of Texas at Austin

December 2012

Weak-Memory Local Reasoning

Publication No. ____________________

Ian Anthony Wehrman, Ph.D.
The University of Texas at Austin, 2012

Supervisors: Warren A. Hunt, Jr. and J Strother Moore

Program logics are formal logics designed to facilitate specification and correctness reasoning for software programs. Separation logic, a recent program logic for C-like programs, has found great success in automated verification due in large part to its embodiment of the principle of local reasoning, in which specifications and proofs are restricted to just those resources–variables, shared memory addresses, locks, etc.–used by the program during execution.

Existing program logics make the strong assumption that all threads agree on the values of shared memory at all times. But, on modern computer architectures, this assumption is unsound for certain shared-memory concurrent programs: namely, those with races. Typically races are considered to be errors, but some programs, like lock-free concurrent data structures, are necessarily racy. Verification of these difficult programs must take into account the weaker models of memory provided by the architectures on which they execute.

This dissertation project seeks to explicate a local reasoning principle for x86-like architectures. The principle is demonstrated with a new program logic for concurrent C-like programs that incorporates ideas from separation logic. The goal of the logic is to allow verification of racy programs like concurrent data structures for which no general-purpose high-level verification techniques exist.

Contents

List of Figures

Chapter 1

Introduction

Most concurrent software verification techniques rely on a surprisingly strong assumption: namely, that all processes agree on the value of shared memory at all times. This is, of course, not generally true, but it is often a *safe* assumption because of implicit guarantees provided by the memory models of modern computer architectures, which guarantee that programs without races will not observe such inconsistencies. The soundness of most concurrent software verification techniques therefore relies on race-freedom of the program under study. This is not considered a major shortcoming though because races usually indicate a program error.

There are, however, useful and interesting programs for which races do not indicate an error. For example, concurrent data structures, which optimize for speed and throughput by using locks and memory fence instructions sparingly, are often racy by design. Their correctness is demonstrated by relating the executions of the comparatively daring implementation to those of its simpler, abstract counterpart. Constructing such a relation therefore requires a technique that is tolerant of races. But that requirement comes with a serious consequence: any technique that tolerates races soundly must also admit that processes may observe the inconsistencies in the value of shared memory that result from the peculiarities of the architecture's memory model.

The verification literature offers little insight into the problem of verifying concurrent data structures and other inherently racy programs. This is because a model of a contemporary memory adds serious complication to an already difficult problem, but also because until recently formal specifications of common architectures' memory models did not exist publicly. (Or, perhaps, privately.) Fortunately, the latter problem has been alleviated with recent safety specifications for the x86, Power and ARM memory models [38, 41]. So, for these architectures, the path toward a solution to the correctness problem of concurrent data structures and other important programs now lies essentially unimpeded.

1.1 An Illustrative Example

```
                // initially: $f_0 \mapsto 0 * f_1 \mapsto 0$
$[f_0] := 1;$                        ||   $[f_1] := 1;$
if ($[f_1] == 0$) then               ||   if ($[f_0] == 0$) then
//   critical section                ||   //   critical section

      Process $P_0$                          Process $P_1$
```

Figure 1.1: Dekker's algorithm

To motivate study of a local reasoning principle for weak memory models, we consider the problem of reasoning about programs executing on such models. Some of the issues involved can be illustrated by considering the small pseudocode program in Figure 1.1.[1]

The initial condition states that the two pointer variables, f_0 and f_1 have distinct values, each of which are addresses into shared memory at which the value is 0. The program is a concurrent composition: process P_i, for $i \in \{0, 1\}$, sets its flag by storing the value 1 at address f_i, then optionally enters its critical section if

[1]The result of dereferencing a pointer variable x is indicated by $[x]$.

the result of loading the address at other flag f_{1-i} is 0.[2]

This is a simplification of Dekker's algorithm for mutual exclusion; it should not be possible for both processes to enter their critical sections simultaneously. An informal correctness argument can be made that relies on a widely assumed property of the underlying memory model, *sequential consistency*, defined by Lamport [30] to mean that,

> the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

The informal correctness argument for the program in Figure 1.1 is as follows. In any execution, either process P_0 sets flag f_0 before process P_1 sets flag f_1, or conversely, because we may assume all events are totally (sequentially) ordered. In the first case, f_0 is set before f_1, which happens before P_1 loads f_0 because the total order of events respects the program orders. So, if P_0 sets its flag first, the load of f_0 returns 1 and P_1 will not enter its critical section. A symmetric argument shows that if P_1 sets f_1 before P_0 sets f_0, then P_0 will not enter its critical section. In both cases, at most one of the two processes may enter its critical section.[3]

Sequential consistency is crucial to this argument. If the events are not totally ordered, the case split is not exhaustive. If the program orders are not included in the total order, we may not conclude that the first store precedes the other process' load, despite the fact that the first store precedes the second store (in the total order) and the second store precedes the load (in the program order). In either case, mutual exclusion may fail.

Unfortunately, common multiprocessor architectures do not generally guarantee sequential consistency, and so neither the informal argument above nor more

[2]Load and store are assumed to be atomic operations when operating on integer-valued data.
[3]This program does not, of course, preclude deadlock.

rigorous arguments based on formalizations of sequential consistency are valid. And although the memory models of various architectures all seem to be strictly weaker than sequentially consistent models, they are not individually comparable. Roughly, the Power and ARM architectures guarantee the first part of sequential consistency (a total order on memory events), but not the second (that this order includes the program orders) [41, 13]. Such memory models are called *weakly consistent*. Conversely, the x86 architecture does not guarantee a total order on memory events, but does guarantee that the observed partial order is consistent with the program orders [38]. These memory models are said to have the *total-store ordering* (TSO) property.

There are, however, conditions under which these architectures guarantee a program's executions to be sequentially consistent—namely, in the absence of data races. These so-called "data-race free" (DRF) guarantees provide sufficient conditions under which sequential consistency can be recovered and used for a correctness argument. By guarding the memory-accessing commands in the program in Figure 1.1 with synchronization primitives like locks to eliminate races, the previous correctness argument again becomes valid. Such a program transformation might make sense for a conservative programmer concerned with correctness, but it does not constitute a helpful verification strategy because the transformation does not preserve the original program's semantics.

A less drastic semantics-altering transformation is to add fence instructions directly after the processes' store operations; this too results in an implementation of Dekker's algorithm that preserves mutual exclusion. The fences ensure that the processes do not attempt their loads until after their respective stores have completed. But proving that this modified program is correct requires an argument quite different from the one above: the loads in this program race with their opposite stores, and so DRF guarantees cannot be applied to recover sequential consistency. Hence, any correctness argument for this modified program must be cognizant of the peculiarities of the underlying memory model. Indeed, correctness arguments for an x86-like memory model are completely different for a correctness argument for an ARM-like memory model.

1.2 Project Description

This dissertation project does not, by far, solve the verification problem for racy concurrent programs executing on a weak memory models generally. Its more modest goal is instead to step toward this by developing a *program logic* for the verification, by proof, of partial correctness properties of certain programs that interact with a particular weak model of memory. Implicit in this is the goal to explicate a *local reasoning principle* for a weak memory model. Traditional correctness reasoning and specification is global: the entire system must be accounted for, which makes scaling to large programs difficult. Local reasoning dictates instead that reasoning and specification be restricted to just those resources—program variables, shared-memory addresses, locks, etc.—that are accessed or modified by the program during execution.

The particular programs studied in this dissertation are structured, C-like programs with pointers and pointer arithmetic, and concurrency constructs, including memory fences. The language was chosen based on the level of detail with which racy concurrent programs are typically described in the literature: i.e., with structured imperative control-flow constructs like loops and if-statements, but with memory fences and locking explicitly specified.

The particular weak memory model explored in this dissertation is x86-like; in particular, based on the x86-TSO memory model as defined by Owens, Sarkar and Sewell [38]. This model was chosen for a variety of reasons. First, x86 multiprocessors are now in extremely widespread use, commonly found in servers, desktops, laptops, tablets, smartphones and many other small computing appliances. Second, unlike most other modern multiprocessor architectures (e.g., ARM and Power [2, 50]), the memory model is both well understood and has, thanks to Owens et al., a clear, simple and formal specification.

In this project, local reasoning is explored in the context of an x86-like memory model in particular by developing two program logics that embodies such a principle: a logic for purely sequential programs that execute on a single processor; and a more general logic for concurrent programs that execute on multiple processors in parallel. The sequential fragment of this logic is loosely based on separation logic, a recent Hoare-style logic which has spurred a revolution in high-level program reasoning due to the simplicity with which it handles pointers using a local reasoning principle. The concurrent extension of the logic is similarly based on a concurrent extension to separation logic.

1.2.1 Components and Dependencies

The various components of the program logics described in this dissertation and their explicit dependencies are pictured in the (transitively reduced) graph of Figure 1.2. The components are represented by shapes that indicate their approximate type: semantic objects by octagons; formal languages by squares; semantic relationships by ovals; deduction systems by hexagons; and key properties by trapezoids.

The memory model is shown in a double-lined octagon, which reflects the assumption in this project that it is complete and correct, and is not further modified from its operational definition in [38], which is summarized in Section 2.3.

The machine model—in particular, the notion of machine state—depends on, but is distinct from, the memory model. For example, the memory model dictates that each processor has a private set of named registers, whereas in the machine models that we shall define, a single shared set of variable names is assumed. We will

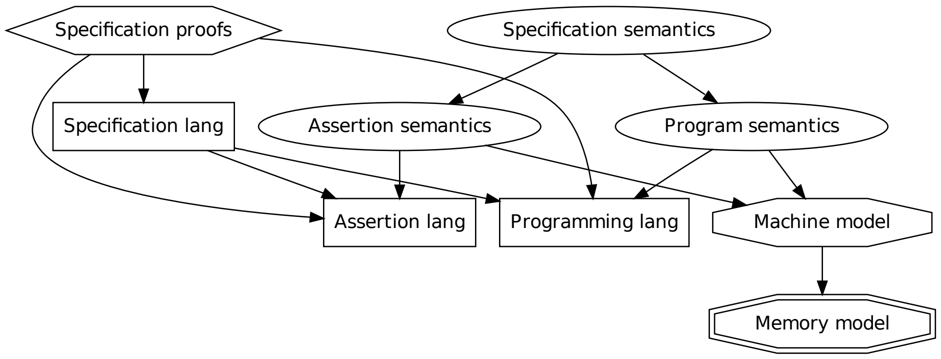

Figure 1.2: Dependencies among the components of the project

also take the liberty in the machine model to relax other restrictions on the notion of state from the memory model, such as the requirement that writes buffered by a single processor are totally ordered.

The programming language is a structured C-like language with concurrent composition. It does not explicitly depend on any other components of the project. The programming language semantics relates the programming language to the machine model, and hence depends on them both. The uniprocessor machine model and sequential programming language are described in Section 3.3; the multiprocessor machine model and concurrent programming language are described in Section 4.2.

The assertion language also does not explicitly depend on any other components of the project.[4] The assertion language semantics associates the assertion language to sets of machine states (as defined by the machine model) with a particular structure. Assertions about uniprocessor machine states are described in Section 3.5; assertions about multiprocessor machine states are described in Section 4.4.

Ideally there would also be a proof theory of assertions and a corresponding soundness theorem. We have chosen not to focus on a proof theory of assertions in this project, but will indicate some semantic implications and equivalences that

[4]Implicitly, of course, it depends significantly on the memory and machine models.

would be relevant to that end in Section 3.5.4 for the uniprocessor case, and in Section 4.4.3 in the multiprocessor case.

The specification language encompasses the programming and assertions languages, and its semantics is given in terms of the semantics of programs and assertions. The proof theory of specifications relies on the existence of a suitable proof theory of assertions for determining entailments. The soundness of the specification logic relies on the soundness of the proof theory of assertions as well as the semantics of programs and assertions. Specification of sequential programs are described in Section 3.6; specifications of concurrent programs are described in Section 4.5.

1.2.2 Contributions and Status

The significant contributions of this dissertation project are as follows:

1. An x86-like operational semantics based on state transitions for a C-like programming language with pointers and pointer arithmetic, memory fences and concurrency constructs. The model is novel, and is expressive enough to describe both processor-parallel and interleaved thread executions.

2. An assertion language for describing, naturally and concisely, x86-like system configurations, and a formal semantics in terms of the aforementioned states. Both the language and the model are novel.

3. A program specification logic—i.e., a formal language of specifications and a proof system—for describing and deducing partial correctness of the aforementioned C-like programs in terms of the aforementioned assertions, as well as a formal semantics of specifications that relates the x86-like execution of C-like programs among states described by assertions. The specification logic additionally embodies an x86-specific principal of local reasoning, which allows proofs to be constructed by describing the interaction between the program

and the small portion of system resources relevant to its behavior; and subsequently generalizing the proof to describe the interaction of the program with more complete system descriptions. This is the first known local reasoning principle for a weak memory model.

The languages, models and deduction system described above are completely and formally defined, having undergone hundreds of revisions. There are however two major components left unfinished:

1. There is no proof system for the assertion language. Omitting this component of the project was an early, intentional decision for two reasons. First, a proof system is necessary for practical application of the specification logic—and in particular automation of the specification logic—but is not critical for the study of the specification language. In practice, syntactic entailment is a practically important approximation of semantic entailment, but in principle simply having a well defined notion of semantic entailment is sufficient. Second, it is hypothesized that the traditional inference rules of first-order logic are sound for the assertion language defined here—as is the case for related theories of similar logics—and also that there is no complete set of rules—as this language encompasses, e.g., arithmetic. The task of finding a suitable proof system for the assertion can thus be seen as a purely practical issue.

2. The soundness proof for the specification logic is incomplete. Although the logic is hypothesized to be sound w.r.t. the model described later in the document, this is of course a serious drawback. Most of the relevant lemmas have been proved previously with various, preliminary versions of the model. There are no known problems with the model that are blocking a soundness proof beyond sheer size of the proof as derived from the complexity of the model. Indeed, the proofs are mostly straightforward inductions. Throughout

the document, key intermediate properties (marked as propositions) that are expected to hold are noted along with, in some cases, proof sketches.

Finally, beyond these technical omissions, it must be admitted that this dissertation gives little indication of how proofs ought to proceed in the logic; only a handful of small examples will be given later on. This is because the logic has evolved with soundness to the x86 memory memory model foremost in mind, instead of as a logical system of interest independent of its potential models. Although the examples presented do indicate that the logic is capable of highly non-trivial program reasoning, the extent of its capability is not yet well understood, and remains as perhaps the most important aspect of future work.

Chapter 2

Background

This chapter describes background material necessary to understand the technical content of this dissertation, beginning with mathematical preliminaries and frequently used notation in Section 2.1, followed by an overviews of research on program logics in Section 2.2 and memory consistency models in Section 2.3.

2.1 Mathematical Preliminaries

Terms are defined throughout this document with the following meta-notation:

$$\textit{object} \ =_{\mathit{df}} \ \textit{definition}.$$

We additionally use the following meta-notation for defining predicates:

$$\textit{predicate} \ \equiv_{\mathit{df}} \ \textit{definition}.$$

For example, for a set S and object $o \notin S$ we define

$$S^o \ =_{\mathit{df}} \ S \uplus \{o\}$$

as the extension of set S by object o. Using this notation, a domain S can be lifted to its *optional domain* by writing $S^{\perp}$ as shorthand for $S \uplus \{\perp\}$, assuming $\perp \notin S$.

2.1.1 Relations

For any set A, we write Id_A for the identity relation on A. For a binary relation R on A and $n \in \mathbb{N}$, we write $\overset{n}{R}$ for the n-fold iteration of R, defined by induction on n as follows:

$$\begin{aligned} \overset{0}{R} &=_{df} Id_A \\ \overset{n+1}{R} &=_{df} \overset{n}{R} \circ R. \end{aligned}$$

We write $\overset{+}{R}$ and $\overset{*}{R}$ for the transitive and reflexive-transitive closures of R, respectively:

$$\overset{+}{R} =_{df} \bigcup_{n \in \mathbb{N}^+} \overset{n}{R} \quad \text{and} \quad \overset{*}{R} =_{df} \bigcup_{n \in \mathbb{N}} \overset{n}{R}$$

2.1.2 Functions

For a (possibly partial) function $f : A \rightharpoonup B$ and $a \in A$ and $b \in B$, we write $f[a \leftarrow b]$ for the updated function:

$$f[a \leftarrow b] =_{df} \lambda x \,.\, \begin{cases} b & \text{if } x = a \\ f(x) & \text{otherwise.} \end{cases}$$

We write $f(a) = \perp$ if the partial function f is not defined at point a, i.e. if $a \notin \text{dom}(f)$, and $\mathsf{def}(f(a))$ otherwise. The everywhere-undefined function is indicated by $\emptyset$. The partial sum $f \uplus g$ of partial functions f and g is defined, when

$\mathrm{dom}(f) \cap \mathrm{dom}(g) = \emptyset$, as follows:

$$f \uplus g =_{df} \lambda x \,.\, \begin{cases} f(x) & \text{if } x \in \mathrm{dom}(f) \\ g(x) & \text{else if } x \in \mathrm{dom}(g) \\ \bot & \text{otherwise.} \end{cases}$$

When convenient, we also write f_a as shorthand for $f(a)$. $a \mapsto b$ is shorthand for the unique partial function f such that $f(a) = b$ and is undefined otherwise. For $A' \subseteq A$, $f|_{A'}$ is the restriction of f to domain A'.

For (possibly partial) functions $f, g : A \rightharpoonup B$, we we write $f \mathbin{\backslash\!\backslash} g$ for the result of *overriding* f with g:

$$f \mathbin{\backslash\!\backslash} g =_{df} \lambda x \,.\, \begin{cases} g(x) & \text{if } x \in \mathrm{dom}(g) \\ f(x) & \text{otherwise.} \end{cases}$$

An obvious property is that, if $g(a) = b$, for some $b \in B$, then also $(f \mathbin{\backslash\!\backslash} g)(a) = b$. Some additional properties follow in Proposition 1.

Proposition 1. *Let $f, g, h : A \rightharpoonup B$.*

1. $f \mathbin{\backslash\!\backslash} \emptyset = \emptyset \mathbin{\backslash\!\backslash} f = f$
2. $f \mathbin{\backslash\!\backslash} (g \mathbin{\backslash\!\backslash} h) = (f \mathbin{\backslash\!\backslash} g) \mathbin{\backslash\!\backslash} h$.
3. $\mathrm{dom}(f \mathbin{\backslash\!\backslash} g) = \mathrm{dom}(f) \cup \mathrm{dom}(g)$.
4. *If* $\mathrm{dom}(f) \cap \mathrm{dom}(g) = \emptyset$ *then* $f \mathbin{\backslash\!\backslash} g = f \uplus g$, *and hence* $f \mathbin{\backslash\!\backslash} g = g \mathbin{\backslash\!\backslash} f$.

2.1.3 Lists

The empty list is denoted by ε, the literal list by $[o, \ldots, o']$, list construction by $o :: l$, and list concatenation by $l +\!\!+ l'$, for objects o and lists l. We write $\mathcal{T}$ list to indicate lists of elements drawn from the set $\mathcal{T}$, and $\mathcal{E}$ for the function $(\lambda x\,.\,\varepsilon)$.

For a list $l : (A \times B)$ list, we write $\bar{l}$ for the corresponding partial lookup function:

$$\bar{l} =_{df} \lambda x\,.\begin{cases} b & \text{if } l = l' +\!\!+ [(x,b)] \\ \overline{l'}(x) & \text{if } l = l' +\!\!+ [(y,b)] \text{ with } x \neq y \\ \bot & \text{otherwise.} \end{cases}$$

For $A' \subseteq A$, $l|_{A'}$ is the sublist restriction of l to domain A'.

For convenience, we lift these function definitions pointwise to sets of lists. For example, for a set L of lists, $a :: L =_{df} \{a :: l \mid l \in L\}$.

The set of *interleavings* of lists m, n, written $m \uplus n$, is defined by recursion on the structure of m and n:

$$\begin{aligned} m \uplus \varepsilon &=_{df} \{m\} \\ \varepsilon \uplus n &=_{df} \{n\} \\ a :: m' \uplus b :: n' &=_{df} a :: (m' \uplus (b :: n')) \cup b :: ((a :: m') \uplus n'). \end{aligned}$$

We now define a subset of the interleavings of lists of pairs from $A \times B$ that play an analogous role to function overriding. The result of overriding a list m with another n, written $m \backslash\!\!\backslash n$, is defined as follows:

$$l \in m \backslash\!\!\backslash n \equiv_{df} \bar{l} = \overline{m} \backslash\!\!\backslash \overline{n}.$$

As with function overriding, list overriding has the property that, if $\overline{n}(a) = b$, for some $b \in B$, and $l \in m \backslash\!\!\backslash n$, then $\bar{l}(a) = b$. Because all elements of $m \backslash\!\!\backslash n$ have the same lookup function, we may safely extend the list lookup notation as follows:

$$\overline{m \backslash\!\!\backslash n} =_{df} \lambda x\,.\,\bar{l}(x),$$

for arbitrary $l \in m \backslash\backslash n$.

As for function overriding, list overriding has the basic property that, if $a \in \mathrm{dom}(\overline{n})$, then $\overline{n}(a) = \overline{n \backslash\backslash m}(a)$. It also has the other following analogous properties, as noted by Proposition 2.

Proposition 2. *Let* $l, m, n : A \times B\ \mathsf{list}$*.*

1. $l \backslash\backslash \emptyset = \emptyset \backslash\backslash l = l$
2. $l \backslash\backslash (m \backslash\backslash n) = (l \backslash\backslash m) \backslash\backslash n$.
3. $\mathrm{dom}(\overline{m \backslash\backslash n}) = \mathrm{dom}(\overline{m}) \cup \mathrm{dom}(\overline{n})$.
4. *If* $\mathrm{dom}(\overline{m}) \cap \mathrm{dom}(\overline{n}) = \emptyset$ *then* $m \backslash\backslash n = m \uplus n$*, and hence* $m \backslash\backslash n = n \backslash\backslash m$.
5. $(m +\!\!+ n) \in (m \backslash\backslash n) \subseteq (m \uplus n)$.

2.1.4 Universes

The various universal sets are declared and in some cases defined in Figure 2.1. Note that, in the case of memory locations (i.e., addresses into memory) and processor identifiers, 0 is excluded. Also note that we shall later use the single set of identifiers $\mathbb{I}$ to represent both program variables and logical variables.

Set			Description
$\mathbb{I}$			Identifiers
$\mathbb{V}$	$=$	$\mathbb{Z}$	Values
$\mathbb{L}$	$\subseteq$	$\mathbb{N}^+$	Memory locations
$\mathbb{P}$	$\subseteq$	$\mathbb{N}^+$	Processor identifiers

Figure 2.1: Universal Sets

2.2 Program Logics

Given that the motivating problem is to reason about concurrent programs executing on a particular memory model, how might one approach the correctness of the program in Figure 1.1 and others like it such as concurrent data structures? One solution—perhaps, for now, the best—is to reason directly about the program semantics in the following way:

1. formalize the semantics of the programming language using a general purpose logic (e.g., first-order logic, higher-order logic, type theory);
2. prove that the semantics agrees with the memory model;
3. characterize the intended program property using the general logic;
4. prove using the general logic that the semantic object which represents the program at hand possesses this property.

This is a perfectly reasonable strategy and, by using a proof assistant for a selected general purpose logic (e.g., ACL2 [29], Isabelle/HOL [33], or Coq [5]), is within the realm of feasibility for many programs and some experts.[1] But, due to the generality of the logic and complexity of the semantics of programs under study, one expects such formalizations and proofs to be exceptionally complex. And although experts are certainly able to develop methodologies and abstractions to tame this complexity, the desire to reason at a higher and more intuitive level is manifest.

This is just the purpose of a *program logic*, which allows high-level formal reasoning that codifies the programmer's intuition about the behavior and correctness of the program under study. Ideally, the program logic incorporates those methodologies and abstractions that have been most useful to expert users reasoning directly about semantic objects in more general logics.

The situation is analogous to the use of temporal logics for studying reactive systems. Though it is technically possible to reason about about such systems using

[1]As an alternative to defining a high-level programming language semantics that comports with a high-level specification of a memory model, it is also possible to formally define the machine that implements the memory directly, and give the semantics of programs in terms of possible executions of this machine. This tack was taken, e.g., in work on reasoning about the behavior of multi-threaded Java programs w.r.t. a detailed model of the Java Virtual Machine [32].

a general-purpose logic, both human reasoning (e.g., Unity [12]) and automation (e.g., model checking [14]) were facilitated by specialized logics.

2.2.1 Hoare Logic

Hoare introduced the first program logic for an Algol-like language in his seminal 1969 paper [22]. A program c is specified with a pair of assertions P and Q, written in first-order logic, that describe pre- and post-execution system states, respectively. Hoare described two related logics, which differ in the style of specification: in the *total correctness* logic specifications are written $\langle P\rangle\, c\, \langle Q\rangle$ and require program termination as a necessary condition for satisfaction; in the *partial correctness* logic specifications are written $\{P\}\ c\ \{Q\}$ and allow divergent executions of the program to satisfy *any* specification. In the sequel, we focus on logics of partial correctness.

The axioms and inference rules are either *structural*, directed by the program syntax, or *logical*, directed by the logical operations of the assertion language. The CHOICE rule, for reasoning about nondeterministic choice command $c + c'$, is an example of a structural rule:

$$\frac{\{P\}\ c\ \{Q\} \quad \text{and} \quad \{P\}\ c'\ \{Q\}}{\{P\}\ c + c'\ \{Q\}} \qquad (\text{CHOICE})$$

According to this rule, in order to prove a specification $\{P\}\ c + c'\ \{Q\}$, it suffices to prove the same specification of each of the constituent commands: $\{P\}\ c\ \{Q\}$ and $\{P\}\ c'\ \{Q\}$. This is because the nondeterministic choice command may execute either c or c', but not both; and so if each constituent command satisfies the specification, then so must the composed command.

The DISJ rule, for reasoning about specifications in which the primary connective of the pre-condition is a logical disjunction, is an example of a logical rule:

$$\frac{\{P\}\ c\ \{Q\} \quad \text{and} \quad \{P'\}\ c\ \{Q\}}{\{P \vee P'\}\ c\ \{Q\}} \qquad (\textsc{disj})$$

According to this rule, to prove a specification of an arbitrary comment in which the pre-condition is a disjunction, it suffice to prove a specifications in which the pre-conditions consist of each of the disjuncts. Intuitively, regardless of whether the command is executed in a state that satisfies P or P' it shall terminate in a state that satisfies Q, or diverge.

The structural and logical rules of Hoare logic make proof construction partially mechanical. But determining whether a program meets its specifications remains generally undecidable due to the expressiveness of the assertion language. For example, the *rule of consequence* allows for the relaxation of specifications by the arbitrary strengthening of pre-conditions and weakening of post-conditions:

$$\frac{P' \Rightarrow P \quad \{P\}\, c\, \{Q\} \quad Q \Rightarrow Q'}{\{P'\}\, c\, \{Q'\}} \qquad (\textsc{cons})$$

For correct application of the rule, validity of the first-order logic implications must be proved. But first-order validity is, of course, undecidable in general, so while Hoare logic does ease some of the pain of proof construction, it is not a panacea.

2.2.2 Separation Logic

As successful as Hoare Logic has been, a significant drawback is its inability to soundly cope with pointer variables and dynamically allocated memory, thus severely complicating its application to low-level systems programs. To illustrate the problem, consider a simple program that updates the value at an address x by writing to a dereferenced pointer: $[x] := 2$. To be clear, this program does not update the value x, but instead the value in memory at the address whose value is x. If we write $\mathit{heap}(x) = 2$ to mean that the value in memory address x is equal to 2, then the following specification is clearly true:

$$\{heap(x) = 1\}\ [x] := 2\ \{heap(x) = 2\}\,.$$

We might then wish to prove a stronger specification, which describes the behavior of this program in a larger memory, with two allocated addresses x and y:

$$\{heap(x) = 1 \wedge heap(y) = 1\}\ [x] := 2\ \{heap(x) = 2 \wedge heap(y) = 1\}\,.$$

This is easily derivable in Hoare logic using the *rule of constancy*, which allows an arbitrary[2] assertion F—called a *frame*—to be conjoined uniformly onto the pre- and post-conditions of a derived specification:

$$\frac{\{P\}\ c\ \{Q\}}{\{P \wedge F\}\ c\ \{Q \wedge F\}\,.} \qquad (\textsc{const})$$

The problem, of course, is that this strengthened specification is not true. In case the pointer variables x and y are *aliased*—i.e., if $x = y$—after the pointer update it shall certainly not be the case that $heap(y) = 2$, for the update to the memory at x also implicitly updated the memory at y.

In the presence of pointer variables, the rule of constancy—which is crucial to the scalability of Hoare logic—is only sound in case the memory locations referred to by the frame are disjoint or separate from those in the *footprint* of the command; i.e., the part of the system state referenced by the program during its execution. To maintain soundness, therefore, proofs in Hoare logic about such programs require a variety of ad hoc extensions and onerous side conditions. (See, e.g., Richard Bornat's 2000 paper [6], which incorporates many sophisticated ideas into Hoare logic.) But after more than forty years of research, a major breakthrough finally

[2]Actually, there is a simple syntactic restriction on F: namely, that $\mathrm{fv}(F) \cap \mathrm{mod}(c) = \emptyset$; i.e., that the free variables of the frame are not modified by the command. This holds in the example because no variables are modified by dereferencing assignment.

came with the invention of *separation logic* [47], generally credited to John Reynolds and Peter O'Hearn. Separation Logic is a Hoare-style program logic insofar as the axioms and inference rules are similar; the crucial difference between it and Hoare logic is the choice of assertion language. Instead of the classical first-order logic assertions used by Hoare logic specifications, separation logic makes use of a different logic—a theory of the logic of *bunched implications* pioneered by O'Hearn, Pym and others [35, 45]—for describing system states with a notion of a *heap*—a finite partial function, into which pointers point, that represents part of memory—and disjointness of said states. The salient formulas that capture these notions are the *points-to assertion*, $\ell \mapsto v$, and the *separating conjunction*, $P * Q$. Models of points-to assertion $\ell \mapsto v$ are heaps with exactly one address allocated, given by ℓ, and with value v stored at that address: i.e., $h \models \ell \mapsto v$ iff $h = \{(\ell, v)\}$. Models of the separating conjunction $P * Q$ are heaps that can be partitioned by address into two subheaps, one of which models the formula P and the other Q: i.e., $h \models P * Q$ iff $h = h_P \uplus h_Q$ such that $h_P \models P$ and $h_Q \models Q$.

Besides soundness w.r.t. a sequential C-like language with pointers and dynamic memory management, separation logic is important because it embodies the principle of *local reasoning.* Unlike with Hoare logic, reasoning may be restricted to a program component's footprint, from which one may generalize to complete system states. O'Hearn, Reynolds and Yang informally describe local reasoning in the context of sequential pointer programs as follows [36]:

> To understand how a program works, it should be possible for reasoning and specification to be confined to the [memory addresses] that the program actually accesses. The value of any other [addresses] will automatically remain unchanged.

Separation logic embodies the principle of local reasoning with its *small axioms* and its *frame rule.* The small axioms describe the programming language's primitive commands, specifying only their respective footprints. For example, the small axiom[3] for the pointer assignment command (i.e., store command) requires with its pre-condition only that the relevant location be allocated with some value,[4] and the resulting post-condition describes only the result of updating this location:

$$\{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\} \qquad \text{(STORE)}$$

The local specification can then be generalized to a global specification using the *frame rule*:[5]

$$\frac{\{P\}\ c\ \{Q\}}{\{P * F\}\ c\ \{Q * F\}} \qquad \text{(FRAME)}$$

The frame rule of separation logic replaces the rule of constancy of Hoare logic. It can be used to soundly derive the desired specification for the aforementioned program as follows:

$$\dfrac{\dfrac{\dfrac{}{\{x \mapsto -\}\ [x] := 2\ \{x \mapsto 2\}}\ \text{STORE}}{\{x \mapsto 1\}\ [x] := 2\ \{x \mapsto 2\}}\ \text{CONS}}{\{x \mapsto 1 * y \mapsto 1\}\ [x] := 2\ \{x \mapsto 2 * y \mapsto 1\}}\ \text{FRAME}$$

Besides having been used to give human-readable proofs to a variety of algorithms that manipulate complex pointer-based data structures (e.g., the Schorr-Waite graph-marking algorithm [60]), useful fragments of separation logic have been automated as part of program verifiers and static analyses, which have been successfully applied to programs with tens of thousands of lines of source code [3, 61, 4].

[3]Actually, an axiom schema parametrized by the expressions e and e'.

[4]$e \mapsto -$ is shorthand for $\exists v \,.\, e \mapsto v$.

[5]As in the rule of constancy, the frame rule also requires that $\mathrm{fv}(F) \cap \mathrm{mod}(c) = \emptyset$.

2.2.3 Concurrent Program Logics

Research into logics for concurrent programs has progressed independently from research into logics for increasingly expressive sequential programs. Major efforts are summarized below.

The Owicki-Gries Logic Early attempts at handling concurrency within a program logic culminated in an extension of Hoare logic by Owicki and Gries [39], which adds a rule for parallel composition of two program components with the cumbersome side condition that every assertion in one component's specification be invariant under the operation of each atomic command executed by the other component. While elegant in its simplicity, the side condition restricts the logic's usefulness. First, every application of the parallel composition rule requires a number of invariant proofs quadratic in the size of the components, making scalability difficult. Second, the side condition creates dependencies on the *proofs* of the component specifications, not just on the specifications themselves. This effectively rules out independent proof construction for the individual components and yields a highly non-compositional logic.

Rely/Guarantee Some relief from these problems was provided by Jones in his rely/guarantee logic [28], in which Hoare-style specifications are augmented with two additional assertions: the *rely* condition, which bounds the interference from the environment that a component can tolerate while still meeting its pre- and post-specifications; and the *guarantee* condition, which bounds the interference the program itself may inflict upon the environment. Application of the rely/guarantee parallel composition rule requires proofs that each process' rely condition subsumes the others' guarantee conditions.

While a considerable improvement over the Owicki-Gries logic—subsumption proofs linear in the number of components versus quadratic in their size, and dependence only among components' specifications instead of their proofs—rely/guarantee still has shortcomings. The logic cannot be considered truly compositional because each component may be specified with a variety of interference conditions, and it is not clear which are appropriate until attempting the parallel composition. Furthermore, it can be laborious to specify sufficiently strong guarantee conditions. For example, the guarantee condition for a component with three variables (x, y, z) that updates only one $(x := x + 1)$ must describe not just the relevant variable change $(x' = x + 1)$, but also that all others remain the same $(\ldots \wedge y' = y \wedge z' = z)$—the latter condition being difficult because it suggests a sort of quantification over variable names not possible in first-order logic, instead requiring an explicit numeration of state variables.

RG-Sep A partial solution to this problem of specification has recently appeared via separation logic. A new concurrent program logic from Vafeiadis and Parkinson, dubbed RG-Sep [55] ("a marriage of rely/guarantee and separation logic"), mates a generalization of separation logic's assertion language and frame rules with the rely/guarantee logic. Besides inheriting the local reasoning features of separation logic, RG-Sep eases the pain of specifying environmental interference by semantically partitioning the system state into private and shared parts. A new class of boxed assertions $\boxed{P}$ is used to describe shared state, and the logical operations are adjusted so that, e.g., a separated conjunction of boxed assertions $\boxed{P} * \boxed{Q}$ allows their footprints to overlap.[6] The proof rules are then modified so that, e.g., parallel composition requires only that each component be tolerant of environmental interference to shared state, not private state. This could be used in the previous example to obviate the explicit enumeration needed to describe invariance of the irrelevant state variables.

Embodied within RG-Sep (along with its successors [17]) are some of the most

[6]An interesting consequence is that $\boxed{P} * \boxed{Q}$ is logically equivalent to $\boxed{P \wedge Q}$.

advanced ideas about high-level reasoning techniques for fine-grained concurrent shared-memory programs, including local reasoning. Vafeiadis' 2008 dissertation [53] includes correctness proofs for a variety of complex, racy concurrent data structures using the logic. RG-Sep is by no means simple, but does yield relatively concise, readable proofs about difficult algorithms. Indeed, the only significant criticism leveled here is that, as with all the other concurrent logics discussed, RG-Sep is not sound w.r.t. weak memory models for racy programs.

Concurrent Separation Logic A simplified variant of the original Owicki-Gries logic does away with the complicated interference side condition, instead restricting its scope to well locked[7] and race-free programs. For this smaller (but still large and useful) class of programs, it is possible to use the Owicki-Gries logic to perform *invariant-based* reasoning, in which threads' interaction with shared state may rely only on a specified invariant, and must always preserve that same invariant.

As with Jones' original rely/guarantee logic, this simplified Owicki-Gries logic suffers from the difficulty of constantly having to specify, at each step of the proof, not just what has changed in the program state, but also what has not changed. Once again, the solution is to incorporate the concurrent reasoning of the simplified Owicki-Gries logic in a separation logic-style logic. This logic, called Concurrent Separation Logic (CSL), originally developed by Peter O'Hearn and Stephen Brookes [34, 9], does just that. Hoare-style specifications are elaborated with an invariant I:

$$I \vdash \{P\}\ c\ \{Q\},$$

which indicates informally that, from a state which can be partitioned into a shared part, which satisfies I, and a private part, which satisfies P, that 1) the program c

[7]The exact definition of "well locked" is technical and rather complicated, but purely syntactic, and hence preferable to the logical interference side conditions imposed by the full Owicki-Gries logic.

does not ever encounter a memory error; 2) that throughout execution some portion of the state always satisfies I; and 3) if the program terminates, then it does in a state which can be partitioned finally into a shared part that satisfies I and a private part that satisfies Q. The PAR rule for reasoning about the parallel composition of program components c and c' is, by comparison with the other approaches, extremely simple:

$$\frac{I \vdash \{P\}\ c\ \{Q\} \quad \text{and} \quad I \vdash \{P'\}\ c'\ \{Q'\}}{I \vdash \{P * P'\}\ c \,\|\, c'\ \{Q * Q'\}} \tag{PAR}$$

This rule asserts that if c and c' can be proved to maintain the same invariant along with their own private specifications, then the parallel combination $c \,\|\, c'$ must also maintain that invariant and, assuming the private portions of state accessed by the two components are disjoint, maintains their private specifications as well.

2.3 Memory Consistency Models

A *memory consistency model* (or just *memory model*) is a specification of a concrete implementation of a shared memory system. Memory consistency models bound the possible results of reading an address in shared memory according to the history of writes and reads that have already taken place or are in progress. For the sake of reasoning about the behavior of concurrent software programs interacting with a shared memory, memory consistency models are a crucial abstraction: not only are concrete implementations of shared memory in modern computer systems incredibly complex, but their details may additionally be considered proprietary and secret.[8]

Many memory consistency models have been defined and studied [52, 21, 1]. Below, we discuss (informally) two of particular interest: sequential consistency,

[8]As a consequence of this complexity and secrecy, it can be challenge to determine whether or not a memory implementation is soundly described by a particular memory consistency model. This interesting problem is beyond the scope of this project.

the most commonly used abstraction of memory for well-behaved programs; and the x86-TSO memory model, which describes the interaction between memory and arbitrary software programs on x86-like multiprocessor computers.

2.3.1 Sequential Consistency

Sequential consistency was originally defined by Leslie Lamport in 1979 [30]. For a sequentially consistent program, he wrote:

> the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

As an example, consider again the program in Figure 1.1, in which thread 0 writes to f_0 and then reads from f_1; and thread 1 writes to f_1 and reads from f_0. If we write (p, o, ℓ, v) to indicate that thread p performs operation o (where $o \in \{r, w\}$ is either a read or a write) on location ℓ with value v (indicate the value written or value read), then:

$$(0, w, f_0, 1), (1, w, f_1, 1), (1, r, f_0, 1), (0, r, f_1, 1)$$

is a sequentially consistent execution, because the operations of each thread occur, within the execution, in the same order as the operations appear in their respective programs (with the writes preceding the reads), and the results of the loads clearly are consistent with the total order of operations given.

On the other hand, the following two executions are not sequentially consistent:

$$(0, r, f_1, 0), (1, w, f_1, 1), (1, r, f_0, 0), (0, w, f_0, 1)$$
$$(0, w, f_0, 1), (0, r, f_1, 0), (1, w, f_1, 1), (1, r, f_0, 0),$$

In the first execution, it is not the case that the operations of thread 0 take place in the order specified by the program: the read precedes the write. In the second execution, the order of operations is consistent with the order specified by the program, but the result is not the same as if the operations were executed in that order: the final read of f_0 by thread 1 should not result in value 0, because it directly succeeds the write to f_1.

Operationally, this constraint on the possible execution traces of a program can be realized by modeling memory as a simple location-value map, in which reads and writes to memory happen atomically. That is, memory is represented by a (possibly partial) function $\mathbb{L} \rightharpoonup \mathbb{V}$; and the semantics of the program is defined such that no more than one thread may read or update the memory at once. This, of course, is the most widespread model of memory used in the semantics of imperative, shared-memory programs, and consequently the vast majority of program reasoning and verification techniques tacitly assume that programs are sequentially consistent; i.e., that all its executions can be described by its interactions with this simple model of memory.

2.3.2 The x86-TSO Memory Model

After a great deal of investigation, experimentation and discussion with manufacturers, Owens, Sarkar and Sewell published a specification [38] of the x86 memory model. This model has become widely accepted as accurate [9] and forms the basis for a great deal of related research—e.g., the Comp-Cert project-in-progress [31], which attempts to construct a realistic (i.e., which performs realistic optimizations) and formally verified compiler for C and C++.

The model identified by Owens et al. is essentially a *total-store order* (TSO) model which, as the name implies, indicates that there is a single total order, agreed upon by all processors, that organizes store events. Their x86-TSO model is described as a collection of legal traces of memory events, defined both axiomatically and operationally. The latter, informally, is described in terms of *write buffers*: per-processor FIFO queues of "writes" (i.e., location-value pairs). The informal semantics of store events entails buffering a new write in the processor's write buffer; the semantics of load events entails returning the value of the most recent buffered write to the intended location in the processor's write buffer or, if no such buffered write exists, of the shared memory.

Formally, the operational model is given with a labeled transition relation between machine states. These states are four-tuples (R, m, B, l) in which:

- $R : \mathbb{P} \rightarrow \mathbb{I} \rightharpoonup \mathbb{V}$ represents a register file for each processor;
- $m : \mathbb{L} \rightharpoonup \mathbb{V}$ represents a shared memory;
- $B : \mathbb{P} \rightarrow (\mathbb{L} \times \mathbb{V})\ \mathsf{list}$ represents a write buffer for each processor;
- $l : \mathbb{P}^{\perp}$ represents a global lock.

Transitions (labeled by memory events) between states indicate the possibility and effect of those memory events. For example, in any state (R, m, B, l), processor p may write a value v into its register i; i.e., it may update the state such

[9] An earlier paper [51], suggested that the x86 memory model more closely resembled *causal consistency*—a model in which there is a single agreed-upon order not just for the store events, but for all causally related memory events—but this was later contradicted by counterexamples that lead to the current proposed TSO model.

that $R_p(i) = v$. Similarly, if $R_p(i) = v$ then p may read value v from its register i. A summary of the other events processor p may perform is as follows:

- it may load from its write buffer the most recent value of a location—or, if a write to that location is not found in its write buffer, from memory—if the lock is either available (*i.e.*, the lock value is $\perp$) or is held by p, but not if some other processor $q \neq p$ holds the lock;
- it may store a value to a memory location by adding a new write to the head of its write buffer regardless of the status of the lock;
- it may flush (or, synonymously in this document, commit) the least recent write in its buffer to memory if it holds the lock or the lock is available;
- it may fence, flushing all writes buffered on p to memory, resulting in an empty write buffer;
- it may acquire the lock (i.e., change the lock value in the current state to p) if the lock is available;
- it may release the lock (i.e., change the lock value in the current state to $\perp$) if it holds the lock.

The x86-TSO memory model includes all the sequentially consistent executions, as well as others. For example, the sequentially consistent execution from the previous section:

$$(0, w, f_0, 1), (1, w, f_1, 1), (1, r, f_0, 1), (0, r, f_1, 1),$$

is shown valid (informally) under the x86-TSO model as follows. Assume in the sequel that no processor holds the lock. The first thread may perform its store operation by enqueuing a write $(f_0, 1)$ in its buffer, and then immediately flushing that write to memory; and then similarly for the second thread, which enqueues a write $(f_1, 1)$ and then immediately flushes it to memory. If the second thread then reads location f_0 it finds value 1 (because it has no other writes to f_0 in its buffer); and similarly if the first thread then reads location f_1 it finds value 1 for the same reason.

Next, consider again the two non-sequentially consistent executions of Figure 1.1 from the previous section:

$$(0, r, f_1, 0), (1, w, f_1, 1), (1, r, f_0, 0), (0, w, f_0, 1)$$
$$(0, w, f_0, 1), (0, r, f_1, 0), (1, w, f_1, 1), (1, r, f_0, 0),$$

The first non-sequentially consistent execution is invalid w.r.t. the x86-TSO model as well, because the operations of a single thread again do not take place in the order specified by the program. The second non-sequentially consistent execution, however, is valid under the x86-TSO model: the first process enqueues a write $(f_0, 1)$ in its buffer, and then reads 0 from location f_1 in memory (because its buffer contains no writes to f_0); then the second process enqueues a write $(f_1, 1)$ in its buffer, and then reads 0 from location f_0 in memory (because its buffer has no writes to f_1, only the other buffer).

When combined with the earlier claim that the x86-TSO memory model also includes all sequentially consistent executions, this example shows that x86-TSO memory model is strictly weaker than sequential consistency. Consequently, x86-TSO is considered to be a *weak memory model.*

Note that, despite the completely informal examples above, this specification of the x86-TSO memory model only *bounds* the sort of memory events that can occur in program executions; it does not give meaning to the programs of any particular language, like x86 assembly or C programs. It can, however, be used to give semantics to programs in those languages. Owens et al. use their model as a guide in assigning semantics to a significant subset of the x86 assembly language, for example. The semantics of a simple C-like programming language used in this dissertation project, described in Sections 3.3 and 4.2, is also guided by the x86-TSO memory model. And although it should be possible to prove that this semantics respects the bounds of the model, that is not the focus of the project. But even without such a correspondence proof, it should be clear that the semantics of the programming language to be described later is manifestly weak.

2.3.3 Data-Race-Freedom Guarantees

Although the memory models of most modern computer architectures—including, as we have seen, x86-like computer architectures—are relatively weak, allowing additional executions compared to a sequentially consistent memory model, the tacit assumption that programs are sequentially consistent made by most verification techniques is actually, in many cases, perfectly reasonable. This is because modern computer architectures, including x86, guarantee sequentially consistent execution for a large class of programs: namely, the data-race free programs.[10] Because of these so-called data-race-freedom (DRF) guarantees, verification techniques which assume sequential consistency are perfectly sound in case the program under consideration is race-free. And because races are generally considered to be errors, a reasonable verification workflow is to first check that a program is race-free before proceeding with the verification of more elaborate properties; or, alternatively, to leverage techniques which succeed only for race-free programs. But not all correct programs are race-free; for these programs, verification techniques may need to account for the full complexities of the underlying memory model.

[10]An x86-specific notion of data-race, as well as theorems about data-race-freedom guarantees for the x86, are described in [37].

Chapter 3

A Sequential Program Logic

In this chapter we describe a program logic for a sequential, single-processor, weak-memory programming model. The semantics of programs and assertions will be given in terms of system states with a single write buffer, and the logic is tailored to reason about the behavior of sequential programs w.r.t. these states. The logic developed in this chapter is not especially useful; the behavior of sequential program execution on this model is essentially the same as on the typical, strong-memory model (in which memory is modeled as a single array of addresses), and existing logics for reasoning about this behavior are certainly simpler than the one developed here. But the sequential program logic is a pedagogical stepping stone toward the concurrent program logic, for which the behavior of programs may well be significantly different from that of a strong-memory model. The development of the sequential logic is vastly simpler than the weak-memory, multi-processor logic, and many of the more difficult issues in that task can be introduced and explained more easily in a single-processor setting.

A detailed description and soundness proof of an earlier single-processor, weak-memory program logic was given previously [56].

3.1 An Example Sequential Proof

Consider the following simple sequential program c, which loads the value at an address x into variable t, writes the value $t+1$ into address y, and then flushes that write back to memory with a fence instruction:

$$c =_{df} t := [x] \,;\, [y] := t + 1 \,;\, \mathsf{fence}\,.$$

This sequential program will fail with a memory error unless the value of x, from which the program loads, is a properly allocated memory address, as well as the value y, to which the program writes. Any provable specification of this program must require this of x and y initially. If the value of x and y are indeed allocated addresses, and the value in memory at address x is initially some value z, then upon termination the write buffer will be empty and the value in memory at location y will be $z+1$.

In the program logic we develop throughout this chapter, we express this informal specification of c with the following formal specification:

$$\vdash \{x \mapsto z * y \mapsto -\} \; c \; \{x \mapsto z * y \mapsto z+1\}\,. \tag{3.1}$$

We write $e \mapsto f$ to mean that the value in memory at the address given by the integer-valued expression e is equal to the value given by expression f, or $e \mapsto -$ if the value in memory at e is irrelevant. We also write the *separating conjunction* $e \mapsto f * e' \mapsto f'$ to mean that e and e' are distinct, allocated memory addresses, with values in memory given by f and f' respectively. Hence, the pre-condition on the left asserts that x and y are allocated memory locations, the former with value z, and the post-condition on the right asserts that, upon executing the command c, the value at x remains z and the value at y has been set to $z+1$.

Note that the specification in Equation 3.1 above would not be true if the traditional *additive conjunction* $\wedge$ were used instead of the separating conjunction. That is, the following specification:

$$\vdash \{x \mapsto z \wedge y \mapsto -\}\ c\ \{x \mapsto z \wedge y \mapsto z+1\}\,.$$

is not true because it allows for x and y to be *aliased*—i.e., to denote the same memory address. For in that case the store to y would change the value in memory at y as well as at x, leaving $x \mapsto z+1$ instead of $x \mapsto z$ as specified. Also note that the specification in Equation 3.1 would not be true without the trailing fence command because the buffered write will not necessarily have committed to memory at the moment the commands have completed their execution.

A *proof sketch* of the program specification in Equation 3.1 is as follows, the details of which will be explained shortly:

```
{$x \mapsto z * y \mapsto -$}
    $t := [x]$
{$x \mapsto z * (y \mapsto - \wedge t = z)$}
    $[y] := t + 1$
{$x \mapsto z * (y \mapsto - \lhd y \rightsquigarrow z + 1)$}
    fence
{$x \mapsto z * y \mapsto z + 1$}
```

As the name indicates, a proof sketch provides a skeleton of a complete proof of a program specification. For a program $c_1\,;c_2$, a proof sketch may take the form:

```
{$P$}
    $c_1$
{$R$}
    $c_2$
{$Q$}
```

which indicates that a complete proof will include proofs of the sub-specifications $\vdash \{P\}\ c_1\ \{R\}$ and $\vdash \{R\}\ c_2\ \{Q\}$ that will be combined, in a final step, using a rule of inference (SEQ) for composing specifications of sequentially combined commands:

$$\dfrac{\dfrac{\vdots}{\vdash \{P\}\ c_1\ \{R\}} \quad \dfrac{\vdots}{\vdash \{R\}\ c_2\ \{Q\}}}{\vdash \{P\}\ c_1\,;c_2\ \{Q\}}\ \textsc{seq}$$

where the vertical dots represent the remaining proofs to be supplied.

Returning to the proof sketch of the program specification in Equation 3.1, we see that to complete the proof we must derive three sub-specifications, one for each primitive command:

1. $\vdash \{x \mapsto z * y \mapsto -\}\ t := [x]\ \{x \mapsto z * (y \mapsto - \wedge t = z)\}$
2. $\vdash \{x \mapsto z * (y \mapsto - \wedge t = z)\}\ [y] := t + 1\ \{x \mapsto z * (y \mapsto - \lhd y \rightsquigarrow z + 1)\}$
3. $\vdash \{x \mapsto z * (y \mapsto - \lhd y \rightsquigarrow z + 1)\}\ \mathsf{fence}\ \{x \mapsto z * y \mapsto z + 1\}$

The first specification asserts that the result of loading the value z in memory at address x into variable t results in a state that is otherwise unchanged except that $t = z$. The second specification asserts that the result of storing the value $t + 1$ to the address y results in the addition of a new buffered write to the state, which may later be committed to memory, overwriting the current, unspecified value. The third specification asserts that the result of explicitly flushing this write to memory overwrites the existing value in memory at y with the value $z + 1$.

We shall now derive these three specifications. For the first specification, we start by using an instance of the LOAD axiom scheme to show that the result of evaluating the load command $t := [x]$ from pre-condition $x \mapsto z$ yields the post-condition $x \mapsto z \wedge t = z$.

$$\dfrac{}{\vdash \{x \mapsto z\}\ t := [x]\ \{x \mapsto z \wedge t = z\}}\ \textsc{load}$$

We then use the *spatial frame rule* FRAME-SP to extend the pre- and post-condition with a description of the value of an additional, distinct memory address y:

$$\dfrac{\dfrac{}{\vdash \{x \mapsto z\}\ t := [x]\ \{x \mapsto z \wedge t = z\}}\ \text{LOAD}}{\vdash \{y \mapsto - * x \mapsto z\}\ t := [x]\ \{y \mapsto - * (x \mapsto z \wedge t = z)\}}\ \text{FRAME-SP}$$

Finally, we observe that the derived pre- and post-conditions are not syntactically equal to those of the desired specification, but are logically equivalent:

$$y \mapsto - * x \mapsto z \equiv x \mapsto z * y \mapsto -$$

$$y \mapsto - * (x \mapsto z \wedge t = z) \equiv x \mapsto z * (y \mapsto - \wedge t = z)$$

More generally, the derived pre-condition is logically implied by the desired pre-condition, and the derived post-condition logically implies the desired post-condition. Hence, we may strengthen the pre-condition and weaken the post-condition accordingly with the rule of consequence CONS:

$$\dfrac{\dfrac{\dfrac{}{\vdash \{x \mapsto z\}\ t := [x]\ \{x \mapsto z \wedge t = z\}}\ \text{LOAD}}{\vdash \{y \mapsto - * x \mapsto z\}\ t := [x]\ \{y \mapsto - * (x \mapsto z \wedge t = z)\}}\ \text{FRAME-SP}}{\vdash \{x \mapsto z * y \mapsto -\}\ t := [x]\ \{x \mapsto z * (y \mapsto - \wedge t = z)\}}\ \text{CONS}$$

This completes the derivation of the first specification.

For the second specification, we begin with an instance of the axiom scheme for the store command in which the pre-condition describes just the value of y in memory:

$$\dfrac{}{\vdash \{y \mapsto -\}\ [y] := t + 1\ \{y \mapsto - \lhd y \rightsquigarrow t + 1\}}\ \text{STORE}$$

The post-condition describes, with the *leads-to* assertion $y \rightsquigarrow t + 1$, the addition of a new, buffered write to address y with value $t + 1$. The *temporal separating conjunction* $y \mapsto - \lhd y \rightsquigarrow t + 1$ indicates that the writes described on the left side precede those on the right side.

Next, we again apply the spatial frame rule FRAME-SP to extend the specification with a description of the value of memory at x and the value of variable t, namely $(x \mapsto z \wedge t = z)$, which we abbreviate for space reasons as F_2 below:

$$\dfrac{\dfrac{}{\vdash \{y \mapsto -\}\ [y] := t+1\ \{y \mapsto - \lhd y \rightsquigarrow t+1\}}\ \text{STORE}}{\vdash \{F_2 * y \mapsto -\}\ [y] := t+1\ \{F_2 * (y \mapsto - \lhd y \rightsquigarrow t+1)\}}\ \text{FRAME-SP}$$

The derived pre-condition is logically equivalent to—and, hence, is implied by—the desired pre-condition, and the derived post-condition logically implies the desired post-condition

$$(x \mapsto z \wedge t = z) * y \mapsto - \equiv x \mapsto z * (y \mapsto - \wedge t = z)$$
$$(x \mapsto z \wedge t = z) * (y \mapsto - \lhd y \rightsquigarrow t+1) \models x \mapsto z * (y \mapsto - \lhd y \rightsquigarrow z+1)$$

Hence, we finish the derivation of the second specification with an application of the rule of consequence CONS:

$$\dfrac{\dfrac{\dfrac{}{\vdash \{y \mapsto -\}\ [y] := t+1\ \{y \mapsto - \lhd y \rightsquigarrow t+1\}}\ \text{STORE}}{\vdash \{F_2 * y \mapsto -\}\ [y] := t+1\ \{F_2 * (y \mapsto - \lhd y \rightsquigarrow t+1)\}}\ \text{FRAME-SP}}{\vdash \{x \mapsto z * (y \mapsto - \wedge t = z)\}\ [y] := t+1\ \{x \mapsto z * (y \mapsto - \lhd y \rightsquigarrow z+1)\}}\ \text{CONS}$$

For the third and final specification, we begin with the axiom for the fence command:

$$\dfrac{}{\vdash \{\mathbf{emp}\}\ \mathsf{fence}\ \{\mathbf{bar}\}}\ \text{FENCE}$$

This small axiom allows for the introduction, from an empty system description, of the **bar** assertion. Intuitively, this assertion describes the effect of a barrier on the system state, in the sense that any writes that precede **bar** must necessarily be committed to memory. For example, we have the following logical equivalence:

$$y \mapsto t+1 \;\equiv\; y \rightsquigarrow t+1 \lhd \mathbf{bar}$$

This means that the description of y having value $t+1$ in memory is equivalent to the description of a buffered write to address y with value $t+1$ followed by a flush of that write to memory.[1] We wish to describe the effect of **bar** on the post-condition from the previous specification, so we extend the specification with the *temporal frame rule* FRAME-TM and the frame assertion $x \mapsto z * (y \mapsto - \lhd y \rightsquigarrow z+1)$, which we abbreviate for space below as F_2:

$$\dfrac{\dfrac{}{\vdash \{\mathbf{emp}\}\ \mathsf{fence}\ \{\mathbf{bar}\}}\ \textsc{fence}}{\vdash \{F_2 \lhd \mathbf{emp}\}\ \mathsf{fence}\ \{F_2 \lhd \mathbf{bar}\}}\ \textsc{frame-tm}$$

The empty assertion is a unit w.r.t. both the spatial and temporal separating conjunctions, so the pre-condition assertion is equivalent to one without the final temporal conjunct **emp**. And the post-condition is equivalent to one in which the buffered write has been flushed to memory, replacing the previous value in memory:

$$(x \mapsto z * (y \mapsto - \lhd y \rightsquigarrow z+1)) \lhd \mathbf{bar} \;\equiv\; x \mapsto z * y \mapsto z+1$$

So, once again, the derivation can be completed by an application of the rule of consequence:

$$\dfrac{\dfrac{\dfrac{}{\vdash \{\mathbf{emp}\}\ \mathsf{fence}\ \{\mathbf{bar}\}}\ \textsc{fence}}{\vdash \{F_2 \lhd \mathbf{emp}\}\ \mathsf{fence}\ \{F_2 \lhd \mathbf{bar}\}}\ \textsc{frame-tm}}{\vdash \{x \mapsto z * (y \mapsto - \lhd y \rightsquigarrow z+1)\}\ \mathsf{fence}\ \{x \mapsto z * y \mapsto z+1\}}\ \textsc{cons}$$

This completes the proof of the program specification of Equation 3.1. In

[1] In fact, the points-to assertion shall later on in the description of the sequential logic be made definitionally equal to the temporal conjunction of the analogous leads-to assertion and **bar**.

the following sections, the notions of program, system state, assertion, model, specification and proof are formally defined and discussed.

3.2 Expressions and Stacks

Expressions are terms that denote values, which in this development are just integers. Hence, they are also used later on to denote memory locations and processor identifiers.

The language of expressions, written **Expr**, is given by the following grammar:

$$\textbf{Expr}\ e \ ::= \ v \mid x \mid (e + e') \mid (e - e') \mid \ \ldots,$$

where $v \in \mathbb{V}$ and $x \in \mathbb{I}$. The possibility of additional operations is left open, as knowledge of the complete set of expressions is not particularly important for the purpose of describing the program logic.

The semantics of expressions is given w.r.t. *stacks*, which are total functions from $\mathbb{I}$ to $\mathbb{V}$. The collection of functions $\mathbb{I} \to \mathbb{V}$ is abbreviated **Stack**. The interpretation of an expression w.r.t. a stack s is given by the *extension* of a stack, written $\hat{s}$, which is a total function from **Expr** to $\mathbb{V}$ defined as follows:

$$\begin{aligned}
\hat{s}(v) &=_{df} v \\
\hat{s}(x) &=_{df} s(x) \\
\hat{s}(e + e') &=_{df} \hat{s}(e) + \hat{s}(e') \\
\hat{s}(e - e') &=_{df} \hat{s}(e) - \hat{s}(e')
\end{aligned}$$

Boolean expressions are terms that denote truth values. Their language is given by the following grammar:

$$\textbf{BExpr}\ b\ ::=\ \textbf{false} \mid \textbf{true} \mid (!b) \mid (e = e') \mid \ \ldots,$$

where $e, e' \in \textbf{Expr}$. For convenience, we represent truth values by the set $\{0, 1\}$ so the extension of a stack can also be used to interpret boolean expressions:

$$\begin{aligned}
\hat{s}(\textbf{false}) &=_{df} 0 \\
\hat{s}(\textbf{true}) &=_{df} 1 \\
\hat{s}(!b) &=_{df} 1 - \hat{s}(b) \\
\hat{s}(e = e') &=_{df} \begin{cases} 1 & \text{if } \hat{s}(e) = \hat{s}(e') \\ 0 & \text{otherwise.} \end{cases}
\end{aligned}$$

The set of *free variables* of a (boolean) expression e, written $\mathrm{fv}(e)$, is defined as usual:

$$\begin{aligned}
\mathrm{fv}(v) &=_{df} \emptyset \\
\mathrm{fv}(x) &=_{df} \{x\} \\
\mathrm{fv}(e + e') &=_{df} \mathrm{fv}(e) \cup \mathrm{fv}(e') \\
\mathrm{fv}(e - e') &=_{df} \mathrm{fv}(e) \cup \mathrm{fv}(e') \\
\mathrm{fv}(\textbf{false}) &=_{df} \emptyset \\
\mathrm{fv}(\textbf{true}) &=_{df} \emptyset \\
\mathrm{fv}(!b) &=_{df} \mathrm{fv}(b) \\
\mathrm{fv}(e = e') &=_{df} \mathrm{fv}(e) \cup \mathrm{fv}(e')
\end{aligned}$$

For stacks s, s' and $X \subseteq \mathbb{I}$, we write $s \sim_X s'$ if, for all $x \in X$, $s(x) = s'(x)$. The following basic lemma uses this relation to connect the static and dynamic semantics of expressions.

Lemma 1. *If $s \sim_{\mathrm{fv}(e)} s'$ then $\hat{s}(e) = \hat{s'}(e)$.*

Proof. By induction on the structure of the expression e. $\square$

3.3 Sequential Programs

In this chapter, programs are identified with *sequential commands*, which consist of compositions of *primitive commands* for testing, accessing and modifying state.

The syntax of primitive commands is given by the following grammar:

$$\begin{aligned} \textbf{PComm}\ p\ &::=\ \mathsf{skip} \mid \mathsf{assume}(b) \mid \mathsf{assert}(b) \mid x := e \mid x := [e] \mid \\ &\quad\ [e] := e' \mid \mathsf{fence} \end{aligned}$$

The informal meaning of the primitive commands is as follows.

- skip takes no evaluations steps;
- $\mathsf{assume}(b)$ evaluates to skip if b holds and becomes stuck otherwise;
- $\mathsf{assert}(b)$ evaluates to skip if b holds and aborts otherwise;
- $x := e$ assigns e to identifier x;
- $x := [e]$ assigns the value at memory address e to identifier x;
- $[e] := e'$ stores the value e' to memory address e; and
- fence commits any buffered writes to memory

The formal semantics of the successful execution of a primitive command p is given as a transition relation between machine states σ, σ'

$$p : \sigma \to \sigma'.$$

Informally, a triple p, σ, σ' belongs to this relation if the successful execution of p in state σ may yield state σ'. (A formal interpretation will be given later in the context of a formal semantics of full commands.) A primitive command may alternatively execute unsuccessfully, indicated as follows

$$p : \sigma \rightarrow \lightning .$$

Executions may be unsuccessful as a result of failed assertions—e.g., $\mathsf{assert}(\mathbf{false})$ is unsuccessful from any state—or attempts to access or modify a memory location outside a command's address space. We say that a primitive command *aborts* in such unsuccessful executions.

To define these relations formally, we must first define the notion of a uniprocessor memory system and a machine state.

Definition 1. A *uniprocessor memory system* is a pair (h, b), where:

- $h : \mathbb{L} \rightharpoonup \mathbb{V}$ is a *heap*, i.e., a partial function that represents the allocated locations of shared memory and their values; and
- $b : (\mathbb{L} \times \mathbb{V})$ list is a write buffer;

The set of uniprocessor memory systems is abbreviated as **Mem**. The pair that consists of a stack s, as defined in Section 3.2, which assigns values to identifiers, and a memory system μ is called a *state*, typically abbreviated by σ. The collection of states is written **State**. We often abuse notation by interchanging memory systems and states in definitions for which the stack is irrelevant.

Note that the notion of machine state given here differs from that used to define the memory model. First and most obviously, there is only a single write buffer, because in this section we are discussing only the execution of sequential programs on uniprocessor machines. Second, the global lock value from the memory model is omitted because there is only one write buffer, and hence commands have no need to claim sole ownership of the memory system. Third, the set of names (i.e., "registers," "variables," "identifiers," etc.) are global instead of local to each processor. This is for convenience only, and is not a technical restriction. The specification logic will be restricted to programs for which the names are partitioned among processes, except for those that are never modified. Another reasonable choice would have been to use local names only, and to share read-only values among processes in the shared memory. This has the advantage of codifying the above healthiness condition on programs directly into the model of the language and logic; it has the disadvantage of perhaps making the description of access to shared values slightly more awkward.

The definition of the semantic relation for primitive commands is given in Figure 3.1. By P-ASSUME, the primitive $\mathsf{assume}(b)$ takes an evaluation step only if the boolean expression b evaluates to 1 in the current state. Otherwise, it is effectively stuck. Later, when we describe the specification logic, we will see that such stuck executions are irrelevant to the truth of specifications. In fact, stuck executions are equivalent to diverging executions w.r.t. the specification logic. By P-ASSERT and P-ASSERT-A, $\mathsf{assert}(b)$ either evaluates normally if the boolean expression b evaluates to 1 in the current state, and aborts otherwise. An aborting command will later be shown to not satisfy any specification. By P-ASSIGN, the assignment primitive $x := e$ always evaluates successfully by assigning to x the value of the expression e in the current state. By P-LOAD, the load primitive $x := [e]$ assigns to x the value of the most recent buffered write to the address that is the value of expression e, if such a write exists, and otherwise gives the value in the current heap at that address. If the address that is the value of expression e has neither any buffered writes nor is defined in the heap (i.e., if $\hat{s}(e) \notin (h \backslash\backslash \overline{b})$) then by P-LOAD-A the load primitive aborts. By P-STORE, the store primitive $[e] := e'$ appends to the write buffer a new write with address the value of expression e and value the value of e', but only if the value of e is an address that is already allocated (i.e., if $\hat{s}(e) \in \mathrm{dom}(h \mathbin{\backslash\!\backslash} \bar{b})$). Otherwise, by P-STORE-A the store primitive aborts. Finally, by P-FENCE, the fence primitives evaluates without changing the state if the write buffer is empty. Later, when we describe the semantics of full commands, we will explain how the fence primitive can be thought to flush a non-empty buffer.

Structured commands consist of either a primitive command; a sequential composition of commands; a nondeterministic (internal) choice between commands; or an iteration of a command. The language of commands is defined by the following grammar:

$$\textbf{Comm } c \ ::= \ p_e \mid (c\,;c') \mid (c + c') \mid c^*,$$

where p is a primitive command.

The formal semantics of the successful execution of a command is given as a binary transition relation between command-state pairs:

$$c, \sigma \to c', \sigma'.$$

But a command's execution may abort unsuccessfully as well, as with primitive commands. Unsuccessful executions are modeled as a transition relation between command-state pairs and an erroneous pseudo-state, as for primitive commands:

$$c, \sigma \to \lightning.$$

We refer collectively to command-state pairs and the erroneous state $\lightning$ as *configurations*, and use $\mathcal{C}$ to indicate a configuration. A configuration $\mathcal{C}$ is considered *safe* if it does not abort: $\mathcal{C} \not\twoheadrightarrow \lightning$.

The semantics of commands also encompasses "silent" transitions, which represent the flushing of buffered writes to the shared memory as allowed by the

$$\frac{\text{if } \hat{s}(b) = 1}{\mathsf{assume}(b) : (s, h, b) \to (s, h, b)} \qquad (\text{P-ASSUME})$$

$$\frac{\text{if } \hat{s}(b) = 1}{\mathsf{assert}(b) : (s, h, b) \to (s, h, b)} \qquad (\text{P-ASSERT})$$

$$\frac{\text{if } \hat{s}(b) = 0}{\mathsf{assert}(b) : (s, h, b) \to \lightning} \qquad (\text{P-ASSERT-A})$$

$$\frac{}{x := e : (s, h, b) \to (s[x \leftarrow \hat{s}(e)]\,, h, b)} \qquad (\text{P-ASSIGN})$$

$$\frac{\text{if } (h \mathbin{\backslash\!\!\backslash} \bar{b})(\hat{s}(e)) = v}{x := [e] : (s, h, b) \to (s[x \leftarrow v]\,, h, b)} \qquad (\text{P-LOAD})$$

$$\frac{\text{if } (h \mathbin{\backslash\!\!\backslash} \bar{b})(\hat{s}(e)) = \bot}{x := [e] : (s, h, b) \to \lightning} \qquad (\text{P-LOAD-A})$$

$$\frac{\text{if } \hat{s}(e) \in \operatorname{dom}(h \mathbin{\backslash\!\!\backslash} \bar{b})}{[e] := e' : (s, h, b) \to (s, h, b +\!\!\!+ [\hat{s}(e), \hat{s}(e')])} \qquad (\text{P-STORE})$$

$$\frac{\text{if } \hat{s}(e) \notin \operatorname{dom}(h \mathbin{\backslash\!\!\backslash} \bar{b})}{[e] := e' : (s, h, b) \to \lightning} \qquad (\text{P-STORE-A})$$

$$\frac{\text{if } b = \varepsilon}{\mathsf{fence} : (s, h, b) \to (s, h, b)} \qquad (\text{P-FENCE})$$

Figure 3.1: Semantics of sequential primitive commands

memory model. This flushing is described by a relation $\underset{\tau}{\rightarrow}$ between memory systems[2] defined as follows:

$$(h, b +\!\!+ [(\ell, v)]) \underset{\tau}{\rightarrow} (h[\ell \leftarrow v], b)$$

We write $\preceq$ as shorthand for the converse of the reflexive-transitive closure $\underset{\tau}{\rightarrow}$:

$$\mu \preceq \mu' \equiv_{df} \mu' \overset{*}{\underset{\tau}{\rightarrow}} \mu.$$

The complete relation that defines the semantics of commands is given in Figure 3.2 below. The semantics of primitive commands is lifted directly to the level of commands by C-PRIM and C-PRIM-A. The silent transitions as defined by the $\underset{\tau}{\rightarrow}$ relation are also lifted directly to the level of commands by C-TAU. Sequential compositions $c_1 \,; c_2$ evaluate from left-to-right: by C-SEQ and C-SEQ-A if the left command c_1 evaluates or aborts, then so does the sequential composition; and by C-SEQ-S, once the left command c_1 has evaluated fully to skip it is dropped so that evaluation of the right side c_2 may proceed. By C-CH-1 and C-CH-2, the nondeterministic choice command $c_1 + c_2$ may evaluate to either c_1 or c_2, afterward continuing evaluation of the chosen command. Finally, by C-LOOP, a looping command c^* may always evaluate without modifying the state by expanding to a nondeterministic choice between doing nothing (i.e., exiting the loop) and sequential composition that consists of evaluating the loop body c once and then continuing with the loop.

The reflexive-transitive closure of the command evaluation relation, written $c, \sigma \overset{*}{\rightarrow} \mathcal{C}$, is defined as usual. The *range* of a configuration $\mathcal{C}$, written *range*$(\mathcal{C})$, is

[2]As noted earlier, we abuse notation by interchanging the concept of state and memory system in definitions for which the stack is irrelevant. Hence, the definition of the relation between memory systems also constitutes the definition of a relation between states such that $(s, \mu) \underset{\tau}{\rightarrow} (s, \mu')$ iff $\mu \underset{\tau}{\rightarrow} \mu'$.

defined as the set of states σ for which $\mathcal{C} \xrightarrow{*} \mathsf{skip}, \sigma$.

$$\frac{\text{if } p : \sigma \to \sigma' \text{ and } \sigma' \in \textbf{State}}{p, \sigma \longrightarrow \mathsf{skip}, \sigma'} \tag{C-PRIM}$$

$$\frac{\text{if } p : \sigma \to \lightning}{p, \sigma \longrightarrow \lightning} \tag{C-PRIM-A}$$

$$\frac{\text{if } \sigma \underset{\tau}{\to} \sigma'}{c, \sigma \longrightarrow c, \sigma'} \tag{C-TAU}$$

$$\frac{c, \sigma \longrightarrow c_0, \sigma'}{(c\,;c'), \sigma \longrightarrow (c_0\,;c'), \sigma'} \tag{C-SEQ}$$

$$\frac{c, \sigma \longrightarrow \lightning}{(c\,;c'), \sigma \longrightarrow \lightning} \tag{C-SEQ-A}$$

$$\frac{}{(\mathsf{skip}\,;c'), \sigma \longrightarrow c', \sigma} \tag{C-SEQ-S}$$

$$\frac{}{(c + c'), \sigma \longrightarrow c, \sigma} \tag{C-CH-1}$$

$$\frac{}{(c + c'), \sigma \longrightarrow c', \sigma} \tag{C-CH-2}$$

$$\frac{}{c^*, \sigma \longrightarrow (\mathsf{skip} + (c\,;c^*)), \sigma} \tag{C-LOOP}$$

Figure 3.2: Semantics of sequential commands

Sequential Command Abbreviations A few standard command abbreviations are shown in Figure 3.3. Some would benefit greatly from local variable declarations, which have not yet been added to the language.

Static Semantics The static semantics of expressions, primitive commands and commands, embodied here by functions fv(−) and mod(−) associating these objects to their sets of free and modified variables, respectively, are completely stan-

$$\begin{aligned}
\textsf{if } b \textsf{ then } c \textsf{ else } c' &=_{df} (\textsf{assume}(b)\,;c) + (\textsf{assume}(!b)\,;c') \\
\textsf{if } b \textsf{ then } c &=_{df} (\textsf{assume}(b)\,;c) + (\textsf{assume}(!b)\,;\textsf{skip}) \\
\textsf{while } b \textsf{ do } c &=_{df} (\textsf{assume}(b)\,;c)^*\,;\textsf{assume}(!b)
\end{aligned}$$

Figure 3.3: Sequential command abbreviations

dard. (Especially so because there are no name-hiding operations in the language, like the aforementioned missing local variable declaration command.) For example, $\mathrm{fv}(x := [y+1]) = \{x, y\}$ and $\mathrm{mod}(x := [y+1]) = \{x\}$.

The following lemma notes some basic facts about the relationship between the step relation $\rightarrow$ and the static semantics of commands.

Lemma 2. *If* $c, s, \mu \rightarrow c', s', \mu'$ *then:*

- $\mathrm{fv}(c') \subseteq \mathrm{fv}(c)$
- $\mathrm{mod}(c') \subseteq \mathrm{mod}(c)$
- $s \sim_{\mathbb{I} \setminus \mathrm{mod}(c)} s'$.

Proof. By a straightforward induction on the derivation of $c, s, \mu \rightarrow c', s\, \mu'$. $\square$

3.4 Locality and Separation

This section gives an informal introduction to the notion of local reasoning, including a rough sketch of the idea of a local command. The latter will be codified formally later in the semantics of program specifications in Section 3.6.

The idea of local reasoning is as follows. Perhaps we wish to show that the result of evaluating a command c in a particular state σ is always included in a set of states S—i.e., that the configuration c, σ does not abort, and that if $c, \sigma \overset{*}{\rightarrow} \textsf{skip}, \sigma'$ then $\sigma' \in S$. Suppose the elements of the set S are all composed of elements constructed from some other sets S_0 and S_1—i.e., $\sigma \in S$ iff, for some $\sigma_0 \in S_0$

and $\sigma_1 \in S_1$, $\sigma = \sigma_0 \bullet \sigma_1$ (where $\bullet$ indicates some unspecified state-constructing function)—and the initial state $\sigma = \sigma_0 \bullet \sigma_1$ with $\sigma_0 \in S_0$. In that case, we may wish to reduce the original problem to the potentially simpler task of showing that the result of evaluating c in state σ_1 is always included in S_1—i.e., showing that the configuration c, σ_1 does not abort and that if $c, \sigma_1 \xrightarrow{*} \mathsf{skip}, \sigma_1'$ then $\sigma_1' \in S_1$.

Under what circumstances is this reduction sound? First, it should be the case that if the command does not abort in the local state σ_1 then neither does it in the global state $\sigma = \sigma_0 \bullet \sigma_1$. Or, contrapositively, if $c, (\sigma_0 \bullet \sigma_1) \rightarrow \lightning$ then $c, \sigma_1 \rightarrow \lightning$. This is called the *safety monotonicity* property. Second, it should be the case that if $c, (\sigma_0 \bullet \sigma_1) \xrightarrow{*} \mathsf{skip}, \sigma'$, then there exists σ_1' such that $\sigma' = \sigma_0 \bullet \sigma_1'$ and $c, \sigma_1 \xrightarrow{*} \mathsf{skip}, \sigma_1'$. For then, by assumption $\sigma_0 \in S_0$, by the reduction $\sigma_1' \in S_1$, and by the structure of S $\sigma_0 \bullet \sigma_1' = \sigma' \in S$. This is called the *frame* property. A command that satisfies both the safety monotonicity and frame properties is called a *local command* [62].

For example, c is perhaps a command that reads and writes a particular set of memory addresses, and the starting state σ perhaps describes the value of some addresses that are superfluous to the execution of c. We may decompose σ by partitioning the memory addresses it describes into states σ_0 and σ_1 such that $(\sigma_0 \bullet \sigma_1)$, with σ_1 describing just the memory locations accessed by c and the remainder by σ_0. Because $(\sigma_0 \bullet \sigma_1)$ contains all the memory addresses of σ_1 and more, then if c can execute successfully (i.e., without aborting) from state σ_1 then it can also execute successfully from σ. And because the addresses of σ_0 are irrelevant to the evaluation of c, σ, those addresses will remain unchanged in the resultant state σ', and hence c, σ_1 also evaluates to σ_1' with $\sigma' = \sigma_0 \bullet \sigma_1'$.

This is the essence of local reasoning: leveraging some local property of a command in a local state to a related global property of the command in a global state. Of course, local reasoning is not possible for all properties—it relies crucially on the notion of decomposition $\bullet$—but when it is possible, it offers an especially direct path to showing the desired property. If the commands of a programming language are local, then the principle of local reasoning can be codified into a program logic by way of a *frame rule*, which allows inference from local to global program specifications. This will be considered in more detail in Section 4.5.

In the example above, the state was separated by memory address. But in the programming model described in Section 3.3, memory systems consist of both committed and buffered writes. How should we decompose (or separate) a memory system with buffered writes? Or, alternatively, how and when can we compose two partial memory systems? The goal of the next few sections is to carefully define a handful of different, useful notions of separation for which the sequential commands of the programming language are local. Each notions of separation will eventually yield a different frame rule, and hence a different principle of local reasoning about program specifications.

3.4.1 Spatial Separation

In this section we carefully define a notion of separation for uniprocessor states analogous to the one described in the previous section, in which states are decomposed according to memory address. This is accomplished by defining a notion of separation for memory systems, and then lifting that function to states that have identical stacks: i.e., given a definition of $\mu \bullet \mu'$, the lifted partial function on states $(s, \mu) \bullet (s', \mu')$ is defined as follows

$$(s, \mu) \bullet (s', \mu') =_{df} (s, \mu \bullet \mu') \text{ if } s = s' \text{ and } \mu \bullet \mu' \text{ is defined.}$$

In the sequel, we shall ignore the distinction between the function on memory systems and the lifted function on states.

States are typically separated by the resources they describe. In the traditional, strong-memory heap model of separation logic, the resource is a flat shared memory and the heaps are separated according to address:

$$h_0 \mathbin{\widetilde{*}} h_1 =_{df} \begin{cases} h_0 \uplus h_1 & \text{if } \mathrm{dom}(h_0) \cap \mathrm{dom}(h_1) = \emptyset \\ \bot & \text{otherwise.} \end{cases}$$

Note that the partial function is defined only when the heaps have disjoint domains. It is also, very obviously, commutative.

We wish to define an analogous notion of separation for memory systems, which consist of a heap-write buffer pairs. As in the case of separation logic, we add heaps with disjoint domains. But how should we combine the write buffers? To ensure commutativity of the operation, a natural choice is to *interleave* writes of the buffers:

$$(h_0, b_0) \mathbin{\widetilde{*}} (h_1, b_1) =_{df} \bigcup_{b \in b_0 \uplus b_1} \{(h_0 \uplus h_1, b) \mid \mathrm{dom}(h_0, b_0) \cap \mathrm{dom}(h_1, b_1) = \emptyset\}$$

Above, we write $\mathrm{dom}(h, b)$ as shorthand for $\mathrm{dom}(h) \cup \mathrm{dom}(b)$. The set $\mu_0 \mathbin{\widetilde{*}} \mu_1$ is called the *spatial separation* of μ_0 and μ_1 because it requires disjointness of the constituent domains, and does not constrain the order of of the buffered writes. (In a later section, we will weaken the disjointness requirement to yield a weaker notion of separation.)

Interleaving the write buffers results in a notion of separation in which the relative ordering between the writes in the constituent buffers, which necessarily have distinct memory locations, is irrelevant. For example, for $\mu_0 = (\emptyset, [(\ell, v)])$ and $\mu_1 = (\emptyset, [(m, u)])$, with $\ell \neq m$, we have both $(\emptyset, [(\ell, v), (m, u)]) \in (\mu_0 \mathbin{\widetilde{*}} \mu_1)$ and $(\emptyset, [(m, u), (\ell, v)]) \in (\mu_0 \mathbin{\widetilde{*}} \mu_1)$.

Unlike for the heap separation function, the heap-buffer separation function maps into the power-domain[3] of memory systems: for compatible pairs of memory

[3]Such algebras are called *non-deterministic monoids* [19].

systems, the separation yields a memory system for each possible interleaving of the individual write buffers. The resulting set is non-empty if and only if the domains of the constituent memory systems are disjoint.[4] This necessitates a slight change to the definition of a local command, as explained in Section 3.4. Given a notion of separation that yields a set of memory systems, the safety monotonicity and frame properties must hold for each possible result: i.e., for every $\mu \in \mu_0 \mathbin{\widetilde{*}} \mu_1$, if c, μ is safe then so is c, μ_1; and if $c, \mu \xrightarrow{*} \mathsf{skip}, \mu'$ then there exists μ_1' such that $c, \mu_1 \xrightarrow{*} \mathsf{skip}, \mu_1'$ with $\mu' \in \mu_0 \mathbin{\widetilde{*}} \mu_1'$.

We can informally observe locality as follows. For safety monotonicity, note that the domain of the memory systems $(\mu_0 \mathbin{\widetilde{*}} \mu_1)$ are supersets of the domain of μ_1, and so if, e.g., a load command does not abort with a memory error in μ_1 then neither will it in $(\mu_0 \mathbin{\widetilde{*}} \mu_1)$. For the frame property, a load in $\mu_0 \mathbin{\widetilde{*}} \mu_1$ does not change the memory system, and since we have assumed that it does not abort in μ_1 alone, then it will have the same result in μ_1 as in $(\mu_0 \bullet \mu_1)$. For example, with $\mu_0 = (\emptyset, [(\ell, v)])$, $\mu_1 = (\emptyset, [(m, u)])$ and the load command $c = x := [\ell]$, it is clear that, for any $\mu \in \mu_0 \mathbin{\widetilde{*}} \mu_1$, if $c, (s, \mu) \xrightarrow{*} \mathsf{skip}, (s', \mu')$ then also $c, (s, \mu_1) \xrightarrow{*} \mathsf{skip}, (s', \mu_1)$.

It is useful to to lift the definition of spatial separation from a partial function on states up to a total function on sets of states as follows:

$$S_0 \mathbin{\widetilde{*}} S_1 =_{df} \bigcup_{\sigma_0 \in S_0} \bigcup_{\sigma_1 \in S_1} \{\sigma_0 \mathbin{\widetilde{*}} \sigma_1\}.$$

By overloading notation in this way, this allows us to write, e.g., $\sigma_0 \mathbin{\widetilde{*}} (\sigma_1 \mathbin{\widetilde{*}} \sigma_2)$ as shorthand for $\bigcup \{\sigma_0 \mathbin{\widetilde{*}} \sigma_{12} \mid \sigma_{12} \in \sigma_1 \mathbin{\widetilde{*}} \sigma_2\}$. It also allows us to assert associativity of the lifted function:

$$\sigma_0 \mathbin{\widetilde{*}} (\sigma_1 \mathbin{\widetilde{*}} \sigma_2) = (\sigma_0 \mathbin{\widetilde{*}} \sigma_1) \mathbin{\widetilde{*}} \sigma_2$$

Spatial separation as well as its lifting are also commutative, they have as units

[4]This is more convenient than defining a partial function into the power-domain. In that case, both $\perp$ and $\emptyset$ would apparently both indicate incompatibility.

$(\emptyset, \varepsilon)$ and $\{(\emptyset, \varepsilon)\}$, respectively.

3.4.2 Temporal Separation

Spatial separation results in a set of states that encompasses all possible interleavings of the composed write buffers. Consequently, it is unsuitable for composing states with a particular interleaving in mind. Temporal separation does just this. Given memory systems μ_0 and μ_1, the *strong temporal separation*, $\mu_0 \mathbin{\widetilde{\blacktriangleleft}} \mu_1$ is the element of the set $\mu_0 \mathbin{\widetilde{*}} \mu_1$ in which the writes of μ_0 all precede the writes of μ_1. For example, for $\mu_0 = (\emptyset, [(\ell, v)])$ and $\mu_1 = (\emptyset, [(m, u)])$, the strong temporal conjunction $\mu_0 \mathbin{\widetilde{\blacktriangleleft}} \mu_1$ is given by:

$$\mu_0 \mathbin{\widetilde{\blacktriangleleft}} \mu_1 = (\emptyset, [(\ell, v), (m, u)]).$$

Instead of interleaving the constituent write buffers as in spatial separation, they are *concatenated* by temporal separation. For another example, let $\mu_0' = (\ell \mapsto v, \varepsilon)$. Then $\mu_0' \mathbin{\widetilde{\blacktriangleleft}} \mu_1 = (\ell \mapsto v, [(m, u)])$. Again the writes of μ_0' (which are committed) precede the writes to μ_1 in the composed state $\mu_0' \mathbin{\widetilde{\blacktriangleleft}} \mu_1$. As a final example, consider $\mu_1' = (m \mapsto u, \varepsilon)$ and the temporal separation $\mu_0 \mathbin{\widetilde{\blacktriangleleft}} \mu_1'$. The presumed result of this composition is $(m \mapsto u, [(\ell, v)])$. But this violates the property of having the writes of the left-hand side precede the writes of the right-hand side because, in the composition, the committed write $m \mapsto u$ implicitly precedes the buffered write (ℓ, v). Consequently, in the definition of $\widetilde{\blacktriangleleft}$ we explicitly rule out this case by requiring either that the left-side buffer or right-side heap be empty.

The complete definition of $\mu_0 \mathbin{\widetilde{\blacktriangleleft}} \mu_1$ is as follows:

$$(h, b) \mathbin{\widetilde{\blacktriangleleft}} (h', b') =_{df} \begin{cases} (h \uplus h', b +\!\!+ b') & \text{if } \mathrm{dom}(h, b) \cap \mathrm{dom}(h', b') = \emptyset \\ & \text{and } h' = \emptyset \vee b = \varepsilon \\ \bot & \text{otherwise.} \end{cases}$$

Clearly $\mu_0 \mathbin{\tilde{\blacktriangleleft}} \mu_1$ is defined when $\mu_0 \mathbin{\tilde{*}} \mu_1$ is defined and, because $b \mathbin{+\!\!+} b' \in (b \uplus b')$, $\mu_0 \mathbin{\tilde{\blacktriangleleft}} \mu_1 \in (\mu_0 \mathbin{\tilde{*}} \mu_1)$. The argument for locality w.r.t. temporal separation is consequently similar to that for spatial separation.

The requirement that the constituent domains of the temporal separation $\mu_0 \mathbin{\tilde{\blacktriangleleft}} \mu_1$ be disjoint is rather strong, however, and it is possible to eliminate this condition entirely. We refer, in the sequel, to $\mu_0 \mathbin{\tilde{\blacktriangleleft}} \mu_1$ as the *strong temporal separation* of μ_0 and μ_1, and now define a more relaxed operation $\mu_0 \mathbin{\tilde{\lhd}} \mu_1$, which we call a *weak temporal separation*. As before, we illustrate this separation with a few examples before showing the complete definition.

First, consider $\mu_0 = (\emptyset, [(\ell, v)])$ and $\mu_1 = (\emptyset, [(\ell, u)])$. Their weak temporal separation $\mu_0 \mathbin{\tilde{\lhd}} \mu_1$ simply concatenates the constituent write buffers, giving $(\emptyset, [(\ell, v), (\ell, u)])$. Note that the semantics of load ensures that the result of loading ℓ in the context of μ_1 is the same as for the context of $\mu_0 \mathbin{\tilde{\lhd}} \mu_1$ because only the value of the most recent write to a particular location is returned.

Next consider also $\mu_0' = (\ell \mapsto v, \varepsilon)$ and $\mu_1' = (\ell \mapsto u, \emptyset)$. The weak temporal separation $\mu_0' \mathbin{\tilde{\lhd}} \mu_1$ is defined as for the strong variant: $(\ell \mapsto v, [(\ell, u)])$. And the weak temporal separation $\mu_0 \mathbin{\tilde{\lhd}} \mu_1'$ is undefined as for the strong variant because of the potential for the ostensibly more recent committed write from μ_1' preceding the buffered write of μ_0.

Finally consider the weak temporal separation $\mu_0' \mathbin{\tilde{\lhd}} \mu_1'$. For the strong variant as well as for spatial separation this is, of course, undefined because the constituent domains are not disjoint. This is necessary because the result of adding the maps that represent heaps is undefined in this case; for what would be the result of applying the hypothetical map $(\ell \mapsto v) \uplus (\ell \mapsto u)$ to ℓ? Neither u nor v seem like suitable answers in general, but in the case of *temporal* separation, we can answer confidently: the result should be u, because the committed u write is more recent than the committed v write. Consequently we use the map overriding operation to combine heaps in the definition of weak temporal separation.

$$(h,b) \mathbin{\widetilde{\lhd}} (h',b') =_{df} \begin{cases} (h \mathbin{\backslash\!\backslash} h', b \mathbin{+\!\!+} b') & \text{if } h' = \emptyset \vee b = \varepsilon \\ \bot & \text{otherwise.} \end{cases}$$

One way to understand the choice of the overriding operation on heaps is w.r.t. an alternative, more concrete state model in which the heap is represented by a list of writes l instead of a partial function. The list is intended to capture the complete history of committed writes in the same way that the buffer captures the history of uncommitted writes. The model of state given in Section 3.3 uses a partial function h instead of a list of committed writes because only the most recent committed write to a particular location is relevant to the operational semantics. We can think of this model of state as an abstraction of the more concrete model in which committed writes are represented by lists. The abstraction function α that maps a concrete memory system (l,b) into an abstract memory system $\alpha(l,b)$ is defined as follows:

$$\alpha(l,b) =_{df} (\bar{l}, b),$$

where $\bar{l}$ is the lookup function for list l, as defined in Section 2.1.3.

Let us again consider the definition of weak temporal separation. In the context of concrete states, the definition is completely natural:

$$(l,b) \mathbin{\widetilde{\lhd}}_{\gamma} (l',b') =_{df} \begin{cases} (l \mathbin{+\!\!+} l', b \mathbin{+\!\!+} b') & \text{if } l' = \varepsilon \vee b = \varepsilon \\ \bot & \text{otherwise.} \end{cases}$$

This definition and the abstraction function given above provide a correctness criterion for a candidate definition of weak temporal separation on abstract states, namely that:

$$\alpha((l,b) \mathbin{\widetilde{\lhd}}_{\gamma} (l',b')) = \alpha(l,b) \mathbin{\widetilde{\lhd}} \alpha(l',b').$$

It is easy to see that the definition given above for weak temporal separation for abstract states satisfies this criterion:

$$\begin{array}{cl}
 & \alpha((l,b) \mathbin{\widetilde{\lhd}_{\gamma}} (l',b')) \\
= & \{\text{definition of } \widetilde{\lhd}_{\gamma}\} \\
 & \alpha(l \mathbin{+\!\!+} l', b \mathbin{+\!\!+} b') \\
= & \{\text{definition of } \alpha(-)\} \\
 & (\overline{l \mathbin{+\!\!+} l'}, b \mathbin{+\!\!+} b') \\
= & \{l \mathbin{+\!\!+} l' \in l \mathbin{\backslash\!\!\backslash} l' \text{ and } m \in l \mathbin{\backslash\!\!\backslash} l' \equiv_{df} \overline{m} = \bar{l} \mathbin{\backslash\!\!\backslash} \overline{l'}\} \\
 & (\bar{l} \mathbin{\backslash\!\!\backslash} \overline{l'}, b \mathbin{+\!\!+} b') \\
= & \{\text{definition of } \widetilde{\lhd}\} \\
 & (\bar{l}, b) \mathbin{\widetilde{\lhd}} (\overline{l'}, b') \\
= & \{\text{definition of } \alpha(-)\} \\
 & \alpha(l,b) \mathbin{\widetilde{\lhd}} \alpha(l',b').
\end{array}$$

It is easy to see that both temporal separators are associative and, as for spatial separation, have $(\emptyset, \varepsilon)$ as a unit. Furthermore, the weak variant is defined whenever the strong variant is defined; and when both are defined, they are equal. We may also lift these functions up to the power domain, as we did with spatial separation:

$$\begin{aligned}
S_0 \mathbin{\widetilde{\blacktriangleleft}} S_1 &=_{df} \bigcup_{\sigma_0 \in S_0} \bigcup_{\sigma_1 \in S_1} \left\{\sigma_0 \mathbin{\widetilde{\blacktriangleleft}} \sigma_1 \mid \mathsf{def}(\sigma_1 \mathbin{\widetilde{\blacktriangleleft}} \sigma_2)\right\} \\
S_0 \mathbin{\widetilde{\lhd}} S_1 &=_{df} \bigcup_{\sigma_0 \in S_0} \bigcup_{\sigma_1 \in S_1} \left\{\sigma_0 \mathbin{\widetilde{\lhd}} \sigma_1 \mid \mathsf{def}(\sigma_1 \mathbin{\widetilde{\lhd}} \sigma_2)\right\}
\end{aligned}$$

Finally, we note the fact that the strong temporal separation can be defined in terms of spatial separation and weak temporal separation:

$$\mu_0 \mathbin{\widetilde{\blacktriangleleft}} \mu_1 = \mu \;\Leftrightarrow\; \mu_0 \mathbin{\widetilde{\lhd}} \mu_1 = \mu \wedge \mu \in \mu_0 \mathbin{\widetilde{*}} \mu_1.$$

We will leverage this fact in the sequel to simplify the rest of the development.

3.4.3 Spatiotemporal Separation

Both spatial and temporal are restrictions of a more general, unifying notion of separation. We write $\mu_0 \mathbin{\widetilde{\#}} \mu_1$ for the *spatiotemporal separation* of memory systems μ_1 and μ_2, defined as follows:

$$(h_0, b_0) \mathbin{\widetilde{\#}} (h_1, b_1) \ =_{df} \bigcup_{b \in b_0 \mathbin{\backslash\!\backslash} b_1} \{(h_0 \mathbin{\backslash\!\backslash} h_1, b) \mid \mathrm{dom}(b_0) \cap \mathrm{dom}(h_1) = \emptyset\}\,.$$

For example, consider $\mu_0 = (\emptyset, [(\ell, 1), (m, 2)])$ and $\mu_1 = (\emptyset, [(\ell, 3), (n, 4)])$. Note that μ_0 and μ_1 are not strongly disjoint, and hence $\mu_0 \mathbin{\widetilde{*}} \mu_1$ and $\mu_0 \mathbin{\widetilde{\blacktriangleleft}} \mu_1$ are undefined. The weak temporal separation $\mu_0 \mathbin{\widetilde{\lhd}} \mu_1$ is, on the other hand, defined and equal to $(\varepsilon, [(\ell, 1), (m, 2), (\ell, 3), (n, 4)])$. The spatiotemporal separation $\mu_0 \mathbin{\widetilde{\#}} \mu_1$ includes additionally the following states:

- $(\varepsilon, [(\ell, 1), (\ell, 3), (m, 2), (n, 4)])$
- $(\varepsilon, [(\ell, 1), (\ell, 3), (n, 4), (m, 2)])$
- $(\varepsilon, [(\ell, 1), (\ell, 3), (m, 2), (n, 4)])$

Note in particular that the latter write to ℓ, with value 3, does not precede the earlier write to ℓ with value 1, but otherwise all other interleavings are included. This is crucial to a locality argument because it ensures that a load in state μ_1, if safe, will have the same result as a load in the separated state $(\mu_0 \mathbin{\widetilde{\#}} \mu_1)$.

It is easy to see that spatiotemporal separation generalizes both spatial and temporal separation. In the spatial case, the strongly disjoint definedness condition obviously implies the weakly disjoint definedness condition for spatiotemporal separation. And when the memory systems are strongly disjoint $h_0 \mathbin{\backslash\!\backslash} h_1 = h_0 \uplus h_1$ by

Lemma 1 and $b_0 \mathbin{\backslash\!\!\backslash} b_1 = b_0 \uplus b_1$ by Lemma 2. For the strong temporal case, the definedness conditions are again obviously stronger, and $b_0 +\!\!+ b_1 \in b_0 \mathbin{\backslash\!\!\backslash} b_1$. In the weak case, $b_0 = \varepsilon \vee h_1 = \emptyset$ implies $\mathrm{dom}(b_0) \cap \mathrm{dom}(h_1) = \emptyset$, and again $b_0 +\!\!+ b_1 \in b_0 \mathbin{\backslash\!\!\backslash} b_1$.

As with spatial separation, we lift spatiotemporal separation up to a function on the power domain of memory systems (and states), and abuse notation to refer to whichever function is appropriate in context:

$$S_0 \mathbin{\widetilde{\#}} S_1 =_{df} \bigcup_{\sigma_0 \in S_0} \bigcup_{\sigma_1 \in S_1} \left\{ \sigma_0 \mathbin{\widetilde{\#}} \sigma_1 \right\}.$$

Spatiotemporal separation is associative and has $(\emptyset, \varepsilon)$ as a unit as for the other separators, but it is not commutative.

3.4.4 Flushing Closure

In Section 3.4 we considered the task of showing, for some configuration C and set of states S, that C is safe and, if it evaluates to a configuration skip, σ then $\sigma \in S$. The second part of this task is equivalently restated as requiring that $\mathit{range}(C) \subseteq S$. The set $\mathit{range}(C)$ has a special structure worth noting: namely, it is *down-closed w.r.t. the flushing order.* That is, if $\sigma \in \mathit{range}(C)$ and $\sigma \underset{\tau}{\rightarrow} \sigma'$ then $\sigma' \in \mathit{range}(C)$ as well. This is a consequence of the fact that the nondeterministic flushing of buffered writes is incorporated in the evaluation semantics of programs; from C-TAU, if $\sigma \underset{\tau}{\rightarrow} \sigma'$ then $c, \sigma \rightarrow c, \sigma'$. Hence, if $C \xrightarrow{*} \mathsf{skip}, \sigma$ (by assumption that $\sigma \in \mathit{range}(C)$) and $\sigma \underset{\tau}{\rightarrow} \sigma'$, then

$$C \xrightarrow{*} \mathsf{skip}, \sigma \rightarrow \mathsf{skip}, \sigma'.$$

By transitivity, $C \xrightarrow{*} \mathsf{skip}, \sigma'$ and so by definition of the range of a configuration, $\sigma' \in \mathit{range}(C)$.

For example, the set $S_0 =_{df} \{(s, \emptyset, [(\ell, v)]) \mid s \in \mathbf{Stack}\}$, which consists of states that have a single buffered write, is not closed because that write, according to the definition of the flushing relation $\underset{\tau}{\rightarrow}$ in Section 3.3, may nondeterministically commit to memory as follows:

$$(\emptyset, [(\ell, v)]) \underset{\tau}{\rightarrow} (\ell \mapsto v, \varepsilon),$$

but there is no stack s such that $(s, \ell \mapsto v, \varepsilon) \in S_0$. On the other hand, the set $S_1 =_{df} \{(s, \ell \mapsto v, \varepsilon) \mid s \in \mathbf{Stack}\}$ is closed because each include state is completely flushed, and so none of the included states may take additional flushing steps beyond the bounds of S_1. The set $S_0 \cup S_1$ is also closed because the elements of S_0 may step to elements of S_1. Furthermore, it is easy to see that both the empty set and the set of all states are closed, and that closure is preserved by union and intersection.

Because the sets $range(C)$ are closed, we may focus our attention on showing the correctness of C w.r.t. sets S that are also closed. For if S does not have this special structure, then either it will not be the case that $range(C) \subseteq S$ (if, e.g., $C \xrightarrow{*} \mathsf{skip}, \sigma \rightarrow \mathsf{skip}, \sigma'$ with $\sigma \in S$ but $\sigma' \notin S$), or it will be possible to demonstrate a stronger property of the configuration; namely that $range(C) \subset S'$, for some closed set of states $S' \subseteq S$.

In Section 3.4 we also described a particular strategy for showing $range(C) \subseteq S$ that we referred to as local reasoning. That strategy relied on the ability to consider the set S as a composition of sets S_0 and S_1; i.e., for some notion of separation $\bullet$, it must be the case that $S = S_0 \bullet S_1$. Because we choose to restrict our attention to closed sets, it should be the case that the notion of separation preserves the property of being closed: if S_0 and S_1 are down-closed w.r.t. the flushing order, then S ought to be as well. And, in fact, for the three notions of separation introduced in Sections 3.4.1, 3.4.2 and 3.4.3, this turns out to be the case.

Proposition 3. *If S_0 and S_1 are closed w.r.t. the flushing order, then so are:*

1. *$S_0 \mathbin{\widetilde{*}} S_1$;*

2. $S_0 \mathrel{\widetilde{\lhd}} S_1$*; and*

3. $S_0 \mathrel{\widetilde{\#}} S_1$.

In each case, the proof follows from the fact that if $\mu \in \mu_0 \bullet \mu_1$ and $\mu' \preceq \mu$ then there exists μ'_0 and μ'_1 such that $\mu'_0 \preceq \mu_0$, $\mu'_1 \preceq \mu_1$ and $\mu' \in \mu'_0 \bullet \mu'_1$.

3.5 Sequential Assertions

Sequential assertions denote sets of uniprocessor machine states, and will later be used to express the pre- and post-conditions of commands in the specification logic. The language of sequential assertions is given by the following grammar:

$$\begin{aligned} \textbf{Asrt}\ P \ ::= \ & b \mid (P \vee P') \mid (P \wedge P') \mid (\exists x : P) \mid (\forall x : P) \mid \\ & \textbf{emp} \mid e \rightsquigarrow e' \mid \textbf{bar} \mid (P * P') \mid (P \lhd P') \end{aligned}$$

The informal meaning of the assertions above are as follows. The lifting of a boolean expression (**true**, **false**, $x = y$, etc.) to an atomic formula, disjunction, conjunction and quantifiers have the same basic meaning as in first-order logic: models for which the boolean expression b evaluates to 1; the models that satisfy either P or P', etc. The assertion **emp** describes states with an empty memory system (i.e., both an empty heap and an empty write buffer). $e \rightsquigarrow e'$ describes a single write to location e with value e', either buffered or flushed to memory. **bar** describes empty states in which preceding writes must have been committed to memory. The assertion $(P * P')$ describes spatial separation of the states of P and P', as described in Section 3.4.1. The assertion $(P \lhd P')$ describes temporal separation of the states of P and P', as described in Section 3.4.2. Note that the assertion language does not contain a logical negation operation; this shortcoming is discussed later in Section 3.5.1.

3.5.1 Sequential Satisfaction

The meaning of sequential assertions is given by a satisfaction relation $\mathcal{M} \models P$, relating *uniprocessor models* $\mathcal{M}$ to sequential assertions P. A model is a triple (s, μ, γ), in which (s, μ) is a uniprocessor state, as defined in Section 3.3, and γ is a boolean value. Informally, γ can be interpreted as indicating whether or not writes earlier than those explicitly described by the memory system μ must necessarily have been committed to memory. Note that the boolean γ is used in models of assertions but not in the definition of states because it is irrelevant to the execution of programs, and is only used to give meaning to assertions. We furthermore require that if the heap component of the memory system is non-empty—i.e., describes some writes that have flushed to memory—then $\gamma = \mathbf{t}$. This is because any writes that precede those explicitly described by the state must be flushed to memory, because they precede writes that have been flushed to memory. Because it deals with the flushing status of writes that precede those of a given state, the boolean γ, which we refer to as the *buffer-completeness flag*, is particularly important for modeling the **bar** assertion and the temporal separating conjunction. We call the pairs (μ, γ) that are part of a uniprocessor model $\mathcal{M}$ *generalized uniprocessor memory systems*, and refer to them by the symbol ν.

Definition 2. A *generalized uniprocessor memory system* (typically indicated by the symbol ν) is a triple, (h, b, γ), where

- (h, b) is a uniprocessor memory system; and
- γ is a boolean buffer-completeness flag;

such that if $\gamma = \mathbf{f}$ then also $h = \emptyset$.

The set of all models is abbreviated **Model**, and the satisfaction relation between models and assertions P is defined by recursion on the structure of P, as shown in Figure 3.4.

$$
\begin{array}{llll}
s, \nu & \models & b & \equiv_{df} & \hat{s}(b) = 1 \\
s, \nu & \models & P \vee Q & \equiv_{df} & s, \nu \models P \vee s, \nu \models Q \\
s, \nu & \models & P \wedge Q & \equiv_{df} & s, \nu \models P \wedge s, \nu \models Q \\
s, \nu & \models & \exists x : P & \equiv_{df} & \exists v \in \mathbb{V} : (s[x \leftarrow v], \nu) \models P \\
s, \nu & \models & \forall x : P & \equiv_{df} & \forall v \in \mathbb{V} : (s[x \leftarrow v], \nu) \models P \\
s, \nu & \models & \mathbf{emp} & \equiv_{df} & \nu = (\emptyset, \varepsilon, \mathbf{f}) \vee \nu = (\emptyset, \varepsilon, \mathbf{t}) \\
s, \nu & \models & \mathbf{bar} & \equiv_{df} & \nu = (\emptyset, \varepsilon, \mathbf{t}) \\
s, \nu & \models & e \rightsquigarrow e' & \equiv_{df} & \nu = (\emptyset, [(\hat{s}(e), \hat{s}(e'))], \mathbf{f}) \vee \\
 & & & & \quad \nu = (\emptyset, [(\hat{s}(e), \hat{s}(e'))], \mathbf{t}) \vee \\
 & & & & \quad \nu = (\hat{s}(e) \mapsto \hat{s}(e'), \varepsilon, \mathbf{t}) \\
s, \nu & \models & P_1 * P_2 & \equiv_{df} & \exists \nu_1, \nu_2 : \nu \in (\nu_1 \mathbin{\widehat{*}} \nu_2) \wedge \\
 & & & & \quad s, \nu_1 \models P_1 \wedge s, \nu_2 \models P_1 \\
s, \nu & \models & P_1 \lhd P_2 & \equiv_{df} & \exists \nu_1, \nu_2 : \nu = (\nu_1 \mathbin{\widehat{\lhd}} \nu_2) \wedge \\
 & & & & \quad s, \nu_1 \models P_1 \wedge s, \nu_2 \models P_1
\end{array}
$$

Figure 3.4: Sequential satisfaction relation

The meaning of the standard first-order logic formulas is as usual (i.e., classical). A model $\mathcal{M}$ satisfies **emp** if its memory system μ is empty. A model satisfies **bar** if its memory system is empty and its buffer-completeness flag γ is set. A model satisfies the leads-to assertion $e \rightsquigarrow e'$ either if its memory system consists of an empty heap and a write buffer with a single write $(\hat{s}(e), \hat{s}(e'))$; or if its buffer-completeness flag is set and its memory system consists of the single-point heap $\hat{s}(e) \mapsto \hat{s}(e')$ and an empty buffer. The two classes of model represent a write that is either buffered or committed. For the case in which the write has flushed, any previous writes must also have flushed, and so the buffer-completeness flag is set. A model satisfies the temporal (resp. spatial) separating conjunction if it admits a model-theoretic temporal (resp. spatial) separation, defined below, into models that satisfy the constituent conjunctions.

$$(h_1, b_1, \gamma_1) \mathbin{\widehat{*}} (h_2, b_2, \gamma_2) =_{\mathit{df}} \{(\mu, \gamma_1 \vee \gamma_2) \mid \mu \in (h_1, b_1) \mathbin{\widetilde{*}} (h_2, b_2)\}$$

$$(h_1, b_1, \gamma_1) \mathbin{\widehat{\lhd}} (h_2, b_2, \gamma_2) =_{\mathit{df}} \begin{cases} (\mu, \gamma_1 \vee \gamma_2) & \text{if } \mu = ((h_1, b_1) \mathbin{\widetilde{\lhd}} (h_2, b_2)) \wedge \\ & \qquad (b_1 = \varepsilon \vee \gamma_2 = \mathbf{f}) \\ \bot & \text{otherwise} \end{cases}$$

Note that the above model-theoretic separation functions lift the memory-system-theoretic separation functions defined in Section 3.4. In both cases, the buffer-completeness flag is set in the combined model iff it is set in either of the constituent models. Furthermore, the temporal function requires that the left-side buffer be empty whenever the right-side buffer-completeness flag is set, which corresponds to the informal requirement discussed above that writes which precede a buffer-complete state must be flushed.

The set of free variables of an assertion P, written $\mathrm{fv}(P)$, is defined as usual:

$$\begin{aligned} \mathrm{fv}(P \vee P') &=_{\mathit{df}} \mathrm{fv}(P) \cup \mathrm{fv}(P') & \mathrm{fv}(P \wedge P') &=_{\mathit{df}} \mathrm{fv}(P) \cup \mathrm{fv}(P') \\ \mathrm{fv}(\exists x : P) &=_{\mathit{df}} \mathrm{fv}(P) \setminus \{x\} & \mathrm{fv}(\forall x : P) &=_{\mathit{df}} \mathrm{fv}(P) \setminus \{x\} \\ \mathrm{fv}(\mathbf{emp}) &=_{\mathit{df}} \emptyset & \mathrm{fv}(e \rightsquigarrow e') &=_{\mathit{df}} \mathrm{fv}(e) \cup \mathrm{fv}(e') \\ \mathrm{fv}(P * P') &=_{\mathit{df}} \mathrm{fv}(P) \cup \mathrm{fv}(P') & \mathrm{fv}(P \lhd P') &=_{\mathit{df}} \mathrm{fv}(P) \cup \mathrm{fv}(P') \end{aligned}$$

The following lemma, analogous to Lemma 1, relates the set of free variables of an assertion and the stack-component of the states that satisfy it:

Lemma 3. *If $s \sim_{\mathrm{fv}(P)} s'$ then $s, \nu \models P$ iff $s', \nu \models P$.*

Proof. By induction on the structure of P, using Lemma 1 in the base cases. □

We write $[\![P]\!]$ for the set of all models that satisfy an assertion P,

$$[\![P]\!] =_{df} \{\mathcal{M} \mid \mathcal{M} \models P\},$$

and also $P \models P'$ and $P \equiv P'$ for semantic entailment and equivalence, respectively:

$$P \models P' \equiv_{df} [\![P]\!] \subseteq [\![P']\!]$$
$$P \equiv P' \equiv_{df} [\![P]\!] = [\![P']\!].$$

Assertions can thus be thought of as syntactic constructs that denote sets of uniprocessor models, and hence sets of uniprocessor machine states. We now extend the flushing order on memory systems described in Section 3.4.4 to generalized memory systems:

Definition 3. For generalized memory systems $\nu_1 = (\mu_1, \gamma_1)$ and $\nu_2 = (\mu_2, \gamma_2)$,

$$\nu_2 \xrightarrow[\hat{\tau}]{} \nu_1 \equiv_{df} \gamma_1 = \mathbf{t} \wedge \mu_2 \xrightarrow[\tau]{} \mu_1$$
$$\nu_1 \leq \nu_2 \equiv_{df} \nu_2 \xrightarrow[\hat{\tau}]{*} \nu_1$$

It is easy to see that this defines a partial order on generalized memory systems. Furthermore, the set of models denoted by assertions is closed with respect to this order.

Proposition 4. *If $s, \nu \models P$ and $\nu' \leq \nu$ then $s, \nu' \models P$.*

A corollary of this lemma is that the set of states denoted by an assertions is closed w.r.t. the flushing order.

Corollary 1. *If $s, \mu, \gamma \models P$ and $\mu' \preceq \mu$ then $s, \mu', \mathbf{t} \models P$.*

Proof. $(\mu', \mathbf{t}) \leq (\mu, \gamma)$ because $\mu' \preceq \mu \wedge \mathbf{t} = \mathbf{t}$; then Lemma 4. $\square$

An effect of this is that *assertions are oblivious to the nondeterministic flushing of buffered writes to memory* because they denote all possible memory systems that may result from such flushing. Intuitively, assertions may be thought to describe only the "initial" states, in which no nondeterministic flushing of writes has taken place, though the semantics encompasses all states reachable as a result these steps. We consider this property to be an important feature of the assertion language—and, hence, of the specification language.

Consider, as a significant example, the set of states that satisfy the atomic formula $e \rightsquigarrow e'$. These states may be classified as follows:

1. states that describe a single buffered write, and
2. states in which that write has been committed to memory.

Consider, as a less trivial example, the states that satisfy the compound assertion $1 \rightsquigarrow 3 \lhd 1 \rightsquigarrow 4$. The intuitive "initial" state is one with two successive buffered writes to location 1. The states of the earlier left-side write assertion include ones in which the buffered write has and has not committed, and similarly for the later right-side write assertion. When these two classes of states are combined with the sequential separation function, three classes of states result: those in which neither write has flushed, those in which only the earlier write has flushed, and those in which both have flushed. Crucially, the definition of weak sequential separation rules out the case in which the later write has flushed but not the earlier write. This is summarized in Figure 3.5. This results in a set of states that is closed w.r.t. the flushing relation.

The requirement that assertions denote sets of states that are closed w.r.t. flushing also explains why we have chosen atomic formulas that describe an empty buffer, **bar**, as well as an empty heap and empty buffer, **emp**. A conceivable alternative might be to use one atomic formula to describe empty heaps, say with

$1 \rightsquigarrow 3$	$1 \rightsquigarrow 4$	$1 \rightsquigarrow 3 \lhd 1 \rightsquigarrow 4$
$(\emptyset, [(1,3)], \mathbf{f})$	$(\emptyset, [(1,4)], \mathbf{f})$	$(\emptyset, [(1,3),(1,4)], \mathbf{f})$
$(\emptyset, [(1,3)], \mathbf{t})$	$(\emptyset, [(1,4)], \mathbf{f})$	$(\emptyset, [(1,3),(1,4)], \mathbf{t})$
$(1 \mapsto 3, \varepsilon, \mathbf{t})$	$(\emptyset, [(1,4)], \mathbf{f})$	$(1 \mapsto 3, [(1,4)], \mathbf{t})$
$(\emptyset, [(1,3)], \mathbf{f})$	$(\emptyset, [(1,4)], \mathbf{t})$	$\bot$
$(\emptyset, [(1,3)], \mathbf{t})$	$(\emptyset, [(1,4)], \mathbf{t})$	$\bot$
$(1 \mapsto 3, \varepsilon, \mathbf{t})$	$(\emptyset, [(1,4)], \mathbf{t})$	$(1 \mapsto 3, [(1,4)], \mathbf{t})$
$(\emptyset, [(1,3)], \mathbf{f})$	$(1 \mapsto 4, \varepsilon, \mathbf{t})$	$\bot$
$(\emptyset, [(1,3)], \mathbf{t})$	$(1 \mapsto 4, \varepsilon, \mathbf{t})$	$\bot$
$(1 \mapsto 3, \varepsilon, \mathbf{t})$	$(1 \mapsto 4, \varepsilon, \mathbf{t})$	$(1 \mapsto 4, \varepsilon, \mathbf{t})$

Figure 3.5: Sequential assertion semantics example

arbitrary buffers, say **heapemp**, and another to describe empty buffers with arbitrary heaps. (Then the assertion **emp** could be defined as a simple conjunction.) But **heapemp** is unsuitable because it does not describe a set of states that is closed under flushing. For if any write is flushed from a buffer in a state with an empty heap, the resulting state would have a heap that is nonempty.

The closure requirement also explains why we have omitted negation (and implication) from the assertion language. In a naive semantics, a state might be said to satisfy $\neg P$ if and only if it does not satisfy P; i.e., as a complement:

$$[\![\neg P]\!] =_{df} \mathbf{Model} \setminus [\![P]\!].$$

But the complement of a down-closed set is not generally closed itself, which would violate the predicate requirement.

Let us call the complement of a closed set an *open* set.[5] Open sets are rather curious; for example, an open set of memory systems that contains $(1 \mapsto 2, \emptyset, \mathbf{t})$ also contains every possible memory system that, when flushed, results in this memory system, e.g., $(\emptyset, [(1,2)])$, $(\emptyset, [(1,3),(1,2)])$, $(\emptyset, [(1,4),(1,3),(1,2)])$, etc. Open sets

[5]In fact, the open and closed sets described here appear to form a particular kind of topological space, called an *Alexandrov space*, which is characterized by the property that the property of being open or closed is preserved by arbitrary intersection, and not just finite intersections.

are thus not appropriate for describing individual writes. Sets that are both closed and open might well yield an elegant algebra, allowing full negation and implication, but it is not clear how such an algebra could be used for general program reasoning.

Also note that it is important to define the satisfaction relation so that closure of atomic formulas is immediate and closure is preserved by each connective, as opposed to closing the entire relation at once. For example, if $[\![e \rightsquigarrow e']\!]$ were not closed w.r.t. flushing, then

$$[\![e \rightsquigarrow e' \lhd \mathbf{bar}]\!] = [\![e \rightsquigarrow e']\!] \mathrel{\widehat{\lhd}} [\![\mathbf{bar}]\!] = \emptyset,$$

and hence so too would be its flushing closure.

3.5.2 Sequential Assertion Abbreviations

In this section we extend the language of assertions with additional constructs whose meaning can be defined in terms of the assertions already described.

The following abbreviation, analogous to the points-to formula of separation logic, describes the result of flushing a single write to memory:

$$e \mapsto e' \ =_{df} \ e \rightsquigarrow e' \lhd \mathbf{bar}$$

That is, we may describe the value of a location in memory by describing a buffered write to that location followed by a barrier assertion.

As indicated at the end of Section 3.4.2, we also introduce a strong temporal separating conjunction as the (additive) conjunction of spatial and weak-temporal conjunctions:

$$P \blacktriangleleft P' =_{df} (P \lhd P') \land (P * P').$$

Because the spatial separating conjunction $P * Q$ is commutative, there is no need to define its converse operation. The temporal separating conjunction, however, is not. Hence, we define $P \rhd Q$ as shorthand for $Q \lhd P$, and similarly for $P \blacktriangleright Q$:

$$P \rhd Q =_{df} Q \lhd P$$
$$P \blacktriangleright Q =_{df} Q \blacktriangleleft P$$

3.5.3 Separating Implications

Although not required for the sequential logic, the concept of a *separating implication* will be useful later. Thus, we shall introduce the concept first in the comparatively simple sequential setting to ease their discussion later.

Separating implications are related to separating conjunctions in the same way that the additive implication $(P \Rightarrow Q)$ is related to the additive conjunction $(P \wedge Q)$, namely:

$$(P \Rightarrow Q) \wedge P \models Q.$$

For example, the separating implication for spatial separation, written $P \mathrel{-\!\!*} Q$, describes states that, when spatially conjoined with another state that satisfies P, satisfy Q. Or, less formally, $P \mathrel{-\!\!*} Q$ describes Q states with P-shaped holes. For example, if Q describes a particular kind of record and P a field in that record, then $P \mathrel{-\!\!*} Q$ describes records that lack the P field. Consequently, adding such a field using the spatial separating conjunction, $(P \mathrel{-\!\!*} Q) * P$, yields a Q record:

$$(P \mathrel{-\!\!*} Q) * P \models Q.$$

The meaning of the spatial implication $P \mathrel{-\!\!*} Q$ is given as follows:

$$(s, \mu) \models P \mathrel{-\!\!*} Q \equiv_{df} \forall \mu_0, \mu_1 : s, \mu_0 \models P \wedge \mu_1 = \mu_0 * \mu \Rightarrow s, \mu_1 \models Q$$

Note that, if defined directly, the converse relation $P \mathrel{*\!\!-} Q$ would be equivalent to $Q \mathrel{-\!\!*} P$ because spatial separation is commutative. Consequently, we simply define $P \mathrel{*\!\!-} Q$ to be shorthand for $Q \mathrel{-\!\!*} P$. On the other hand, temporal separation is not commutative, and so we may define *two* notions of temporal implication: $P \mathrel{-\!\!\lhd} Q$ and $P \mathrel{\lhd\!\!-} Q$. The former describes states that, when temporally conjoined on the left side (i.e., the past) with a state that satisfies P, satisfies Q. The latter describes states that, when temporally conjoined on the right side (i.e., the future) with a state that satisfies P, satisfies Q. Formally:

$$(s, \mu) \models P \mathrel{-\!\!\lhd} Q \equiv_{df} \forall \mu_0, \mu_1 : s, \mu_0 \models P \wedge \mu_1 = \mu_0 \lhd \mu \Rightarrow s, \mu_1 \models Q$$
$$(s, \mu) \models Q \mathrel{\lhd\!\!-} P \equiv_{df} \forall \mu_0, \mu_1 : s, \mu_0 \models P \wedge \mu_1 = \mu \lhd \mu_0 \Rightarrow s, \mu_1 \models Q$$

The entailments that characterize the relationship between these implications and temporal conjunction are as follows:

$$P \lhd (P \mathrel{-\!\!\lhd} Q) \models Q$$
$$(Q \mathrel{\lhd\!\!-} P) \lhd P \models Q$$

As with spatial implication, we simply define the converse relations as shorthand:

$$P \mathrel{\rhd\!\!-} Q =_{df} Q \mathrel{-\!\!\lhd} P$$
$$P \mathrel{-\!\!\rhd} Q =_{df} Q \mathrel{\lhd\!\!-} P$$

The right-side temporal implication, and its characterizing entailment, is especially useful when used with the **bar** assertion. In particular, the formula $Q \mathrel{\lhd\!\!-}$ **bar** describes states which, if flushed, would satisfy Q. For example, $x \mapsto$

$1 \mathrel{\triangleleft\!\!-} \mathbf{bar}$ describes states with any number of buffered writes to address x, with the final one having value 1. One consequence is that:

$$(x \rightsquigarrow 2) \lhd (x \rightsquigarrow 1) \models (x \mapsto 1) \mathrel{\triangleleft\!\!-} \mathbf{bar}.$$

The ability to express such sets of states will be of crucial importance to the program logic for concurrent programs described in the following chapter.

3.5.4 Sequential Algebra

A few additional semantic equivalences and entailments are shown in Figures 3.6 and 3.7, respectively. If a formula contains instances of $\bullet$, then that is short-hand for the same formula in which the $\bullet$ has been consistently replaced by any of the four separating conjunctions.

$$\begin{aligned}
P \bullet \mathbf{emp} &\equiv P \\
\mathbf{emp} \bullet P &\equiv P \\
(P \bullet P') \bullet P'' &\equiv P \bullet (P' \bullet P'') \\
P * P' &\equiv P' * P \\
\mathbf{bar} \bullet \mathbf{bar} &\equiv \mathbf{bar} \\
(P \bullet P') \lhd \mathbf{bar} &\equiv (P \lhd \mathbf{bar}) \bullet (P' \lhd \mathbf{bar}) \\
P \blacktriangleleft \mathbf{bar} &\equiv P \lhd \mathbf{bar}
\end{aligned}$$

Figure 3.6: Sequential semantic equivalences

Each of the separating conjunctions is associative and has **emp** as a unit. Furthermore, the spatial separating conjunction is also commutative. The separating conjunctions are additive w.r.t. the barrier assertion **bar**, and **bar** also distributes fully through the separating conjunctions. Finally we note that the strong and weak temporal separating conjunction of a barrier assertion are equivalent because the models of **bar** have no allocated locations, which are what distinguish the two temporal conjunctions.

$$\begin{array}{rll}
\mathbf{bar} & \models \mathbf{emp} & \\
P \bullet P' & \models P'' \bullet P' & \text{if } P \models P'' \\
P \bullet P' & \models P \bullet P'' & \text{if } P' \models P'' \\
e \mapsto e' & \models e \rightsquigarrow e' & \\
P \blacktriangleleft P' & \models P * P' & \\
P \blacktriangleleft P' & \models P \lhd P' & \\
P \bullet (P' \circ P'') & \models (P \bullet P') \circ P'' & \text{for } P \bullet P' \models P \circ P' \\
(P \circ P') \bullet P'' & \models P \circ (P' \bullet P'') & \text{for } P \bullet P' \models P \circ P' \\
(P * P') \blacktriangleleft (P'' * P''') & \models (P \blacktriangleleft P'') * (P' \blacktriangleleft P''') &
\end{array}$$

Figure 3.7: Sequential semantic entailments

The first three entailments—that **bar** strengthens **emp** and monotonicity of the separating conjunctions—follow directly from the definition of the satisfaction relation. The next three entailments follow from their respective abbreviation expansions and by monotonicity. The three separating conjunctions naturally form a sort of lattice, and they satisfy (the second- and third-to-last entailments) what are known as small exchange laws [23]; the full exchange law (the final entailment) only holds for the spatial and strong temporal conjunctions. The full exchange law does not hold for the other combinations of separating conjunctions because, e.g., it implies commutativity of the main connective in the consequent, and the other conjunctions are not commutative.

3.6 Sequential Specifications

A *sequential program specification* is a three-tuple, written as follows:

$$\vdash \{P\}\ c\ \{Q\},$$

where c is a command and P, Q are assertions, referred to as the *pre-condition* and *post-condition*, respectively. The specifications considered here assert *partial correctness* of a command, which means that any specification of a nonterminating command is true. Their informal meaning is roughly analogous to that of separation logic: if c is evaluated in a state that satisfies P then: *1)* it does not abort, and *2)* if it evaluates fully, it terminates in a state that satisfies Q.

3.6.1 Sequential Proof Theory

We now present a proof theory for deriving true sequential program specifications. The axioms and inference rules are strongly inspired by Separation Logic; the axioms and inference rules of Hoare Logic are included verbatim.

$$\vdash \{P\}\ \mathsf{skip}\ \{P\} \qquad (\textsc{skip})$$

$$\vdash \{!b \vee P\}\ \mathsf{assume}(b)\ \{P\} \qquad (\textsc{assume})$$

$$\vdash \{b \wedge P\}\ \mathsf{assert}(b)\ \{P\} \qquad (\textsc{assert})$$

$$\vdash \{P\,[e/x]\}\ x := e\ \{P\} \qquad (\textsc{assign})$$

$$\vdash \{e \rightsquigarrow e' \blacktriangleleft P\}\ x := [e]\ \{(e \rightsquigarrow e' \blacktriangleleft P) \wedge x = e'\} \qquad (\textsc{load})$$

$$\vdash \{e \rightsquigarrow e'' \blacktriangleleft P\}\ [e] := e'\ \{(e \rightsquigarrow e'' \blacktriangleleft P) \lhd e \rightsquigarrow e'\} \qquad (\textsc{store})$$

$$\vdash \{\mathbf{emp}\}\ \mathsf{fence}\ \{\mathbf{bar}\} \qquad (\textsc{fence})$$

Figure 3.8: Sequential axioms

The axioms of the logic are given in Figure 3.8.[6] The axiom for skip indicates its evaluation preserves any pre-condition into a post-condition. The fact that assertions denote down-closed sets of states is crucial for the soundness of this rule, because a configuration skip, σ may well have non-trivial transitions. That is, it may

[6]The axioms and inference rules of the proof theory are of course correctly referred to as axiom schemas and rule schemas, because they must be instantiated metalogically with particular formulas, commands and expressions.

be the case that, for some $\sigma \models P$, that $\mathsf{skip}, \sigma \to \mathsf{skip}, \sigma'$ with $\sigma' \neq \sigma$. In this case it is clear from the operational semantics of commands that $\sigma' \preceq \sigma$, and so $\sigma' \models P$ as well because assertions denote sets that are closed w.r.t. the flushing relation.

The axioms for both the assume and assert commands ensure that the property P holds if the commands either evaluate successfully or diverge. In the case of $\mathsf{assume}(b)$, the command will execute successfully if the boolean expression b evaluates to 1 in the current state and P holds to begin with, and will diverge if b evaluates to 0. Hence, the pre-condition of $\mathsf{assume}(b)$ requires $(!b \vee P)$, which means that either the command will diverge or the property P holds to begin with. In the case of $\mathsf{assert}(b)$, if b evaluates to false then the command will abort, which would invalidate the axiom. Hence, the pre-condition requires both that b evaluate to true and that P holds to begin with; i.e., that $(b \wedge P)$ is true of the starting state.

The assignment axiom, as in Hoare logic, effectively transforms a formula with occurrences of the expression to be assigned into one with that expression replaced by the variable to which the expression was assigned. For example, $\vdash \{y \leq x+1\}\ \ x := x+1\ \ \{y \leq x\}$ can be derived from the assignment axiom by instantiating P by $y \leq x$, given that $(y \leq x)\,[x+1/x] = y \leq x+1$.

The axiom for load requires, in the pre-condition, that the value of the most recent write $e \rightsquigarrow e'$ be specified, along with any other writes P that temporally succeed it. These additional writes must be to locations other than e, for otherwise the specified write $e \rightsquigarrow e'$ would not be the most recent write. Consequently, we use the strong temporal separating conjunction to partition these additional writes in both space and time: $e \rightsquigarrow e' \blacktriangleleft P$. Note that if P *does* assert the existence of succeeding writes to location e, then the pre-condition is inconsistent and the specification becomes vacuously true. Evaluating the load command does not explicitly change the heap or buffer of the current state, just the value of the variable into which the value is loaded. Consequently, the post-condition is just the additive conjunction of the pre-condition and an equality $x = e'$ relating the loaded variable and the value of the most recent write.

The write described in the pre- and post-conditions of the load axiom is specified with the leads-to assertion. This assertion is used to describe writes that may or may not be buffered. If the write is buffered, it could flush to memory before or after the load has completed. But, again, because assertions denote sets of states that are closed w.r.t. flushing, the soundness of the rule is preserved.

Like the load command, the store command similarly requires that the address to be updated is already allocated. In this model, this simply means that there is an existing write to that address, either buffered or flushed. The pre-condition for the store axiom requires, as a witness to the allocation status of the address to be updated, specification of the most recent write $e \rightsquigarrow e'$. Of course, there may be other, more recent, buffered writes, which do not affect the behavior of the store command. These are described by the assertion P, and are combined using the strong temporal separating conjunction with the witness:[7] $e \rightsquigarrow e' \blacktriangleleft P$. Because e is asserted by the pre-condition to be an allocated address, the store command is safe, and as a result simply appends a new write after all others: $(e \rightsquigarrow e' \blacktriangleleft P) \lhd e \rightsquigarrow e''$.

Finally, the axiom for fence specifies an empty pre-condition, because the fence command may execute without any particular assumptions about the state. The result of the fence command, as specified by the post-condition, is simply to introduce the **bar** assertion. This specification is particularly useful when used in conjunction with a frame rule for a temporal separating conjunction, described below.

The inference rules of the sequential logic are given in Figure 3.9. The logical rules DISJ and EX allow the pre-condition to be weakened. Dually, the logical rules

[7]Use of the weak temporal separating conjunction here instead of the strong variant would also be sound, as the leads-to assertion in the pre-condition serves only as a witness to the allocation status of the location to which the command will store. The given axiom asks for the most recent write to that location, but in fact any previous write would be suitable as a witness.

$$\frac{\vdash \{P\}\ c\ \{Q\} \quad \vdash \{P'\}\ c\ \{Q\}}{\vdash \{P \vee P'\}\ c\ \{Q\}} \quad (\textsc{disj})$$

$$\frac{\vdash \{P\}\ c\ \{Q\} \quad x \notin \mathrm{fv}(c, Q)}{\vdash \{\exists x : P\}\ c\ \{Q\}} \quad (\textsc{ex})$$

$$\frac{\vdash \{P\}\ c\ \{Q\} \quad \vdash \{P\}\ c\ \{Q'\}}{\vdash \{P\}\ c\ \{Q \wedge Q'\}} \quad (\textsc{conj})$$

$$\frac{\vdash \{P\}\ c\ \{Q\} \quad x \notin \mathrm{fv}(c, P)}{\vdash \{P\}\ c\ \{\forall x : Q\}} \quad (\textsc{all})$$

$$\frac{\vdash \{P\}\ c\ \{Q\} \quad \mathrm{mod}(c) \cap \mathrm{fv}(R) = \emptyset}{\vdash \{R * P\}\ c\ \{R * Q\}} \quad (\textsc{frame-sp})$$

$$\frac{\vdash \{P\}\ c\ \{Q\} \quad \mathrm{mod}(c) \cap \mathrm{fv}(R) = \emptyset}{\vdash \{R \lhd P\}\ c\ \{R \lhd Q\}} \quad (\textsc{frame-tm})$$

$$\frac{\vdash \{P\}\ c\ \{Q\} \quad \mathrm{mod}(c) \cap \mathrm{fv}(R) = \emptyset}{\vdash \{R \blacktriangleleft P\}\ c\ \{R \blacktriangleleft Q\}} \quad (\textsc{frame-stm})$$

$$\frac{P \models P' \quad \vdash \{P'\}\ c\ \{Q'\} \quad Q' \models Q}{\vdash \{P\}\ c\ \{Q\}} \quad (\textsc{cons})$$

$$\frac{\vdash \{P\}\ c\ \{R\} \quad \vdash \{R\}\ c'\ \{Q\}}{\vdash \{P\}\ c\,;c'\ \{Q\}} \quad (\textsc{seq})$$

$$\frac{\vdash \{P\}\ c\ \{Q\} \quad \vdash \{P\}\ c'\ \{Q\}}{\vdash \{P\}\ c + c'\ \{Q\}} \quad (\textsc{choice})$$

$$\frac{\vdash \{P\}\ c\ \{P\}}{\vdash \{P\}\ c^*\ \{P\}} \quad (\textsc{loop})$$

Figure 3.9: Sequential inference rules

CONJ and ALL allow the post-condition to be strengthened.

The next three inference rules are variants of the frame rule of separation logic; one for each of the three separating conjunctions: spatial, temporal and strong temporal. Note that, although the strong temporal conjunction is *defined* as the additive conjunction of spatial and temporal conjunctions, its frame rule is not derivable from the other rules, which justifies the rule's inclusion. The spatial separating conjunction is commutative, but the others—which have a temporal aspect—are not. We have left-side frame rules only for these conjunctions.

The rule of consequence, as in Hoare Logic, allows arbitrary strengthening of the pre-condition and weakening of the post-condition. Typically the side conditions, which ensure that the pre-condition is strengthened and the post-condition is weakened, are described using syntactic entailment among assertions. But because in this project we have not developed a proof system for syntactic entailment—though some rules can be gleaned from Section 3.5.4—we describe these conditions using semantic entailment instead.

The remaining structural rules are exactly as in Hoare Logic. The rule for the sequential composition $c\,;c'$ requires identifying an intermediate assertion used as a post-condition for the former command and a pre-condition for the latter. A specification can be proved of the nondeterministic choice $c + c'$ if the same specification can be proved of both commands individually. Finally, an invariant specification holds for a loop if the body the loop maintains the invariant.

Derived and Alternative Axioms and Inference Rules

Some other useful rules may be derived from the set given in Figure 3.9. For example, it is possible to derive modified load and store axioms that describe flushed writes, with the points-to assertion, in the pre- and post-condition. Because the points-to assertion $e \mapsto e'$ is defined as $e \rightsquigarrow e' \lhd \mathbf{bar}$, we may instantiate the placeholder formula P in either axiom schemas with a leading barrier assertion. Here is a derived LOAD-MEM axiom schema

$$\dfrac{\overline{\vdash \{e \rightsquigarrow e' \blacktriangleleft (\mathbf{bar} \blacktriangleleft P)\}\ x := [e]\ \{(e \rightsquigarrow e' \blacktriangleleft (\mathbf{bar} \blacktriangleleft P)) \wedge x = e'\}}\ \textsc{load}}{\vdash \{e \mapsto e' \blacktriangleleft P\}\ x := [e]\ \{(e \mapsto e' \blacktriangleleft P) \wedge x = e'\}}\ \textsc{cons}$$

And a derived STORE-MEM axiom schema:

$$\dfrac{\overline{\vdash \{e \rightsquigarrow e' \blacktriangleleft (\mathbf{bar} \blacktriangleleft P)\}\ [e] := f\ \{(e \rightsquigarrow e' \blacktriangleleft (\mathbf{bar} \blacktriangleleft P)) \lhd e \rightsquigarrow f\}}\ \textsc{store}}{\vdash \{e \mapsto e' \blacktriangleleft P\}\ [e] := f\ \{(e \mapsto e' \blacktriangleleft P) \lhd e \rightsquigarrow f\}}\ \textsc{cons}$$

The rule of consequence is applied properly above because the strong temporal conjunction is associative, and because

$$e \rightsquigarrow e' \blacktriangleleft \mathbf{bar}\ \equiv\ e \rightsquigarrow e' \lhd \mathbf{bar}\ =_{df}\ e \mapsto e',$$

according to the equivalence from Figure 3.6 and the definition of the points-to assertions.

Another example of a useful derived rule is a "global" fence axiom, which describes the result of a fence operation on a potentially complete system state description:

$$\dfrac{\dfrac{\overline{\vdash \{\mathbf{emp}\}\ \mathsf{fence}\ \{\mathbf{bar}\}}\ \textsc{fence}}{\vdash \{P \lhd \mathbf{emp}\}\ \mathsf{fence}\ \{P \lhd \mathbf{bar}\}}\ \textsc{tm-frame}}{\vdash \{P\}\ \mathsf{fence}\ \{P \lhd \mathbf{bar}\}}\ \textsc{cons}$$

We can also use the sequential separating implication to give an alternative axiom for the fence command for backwards reasoning:

$$\overline{\vdash \{P \mathrel{\lhd\!\!-} \mathbf{bar}\}\ \mathsf{fence}\ \{P\}}\ \textsc{backwards-fence}$$

This axiom describes a sufficiently weak[8] pre-condition for the operation of the fence command in terms of an arbitrary post-condition P.

3.6.2 Semantics of Sequential Specifications

Following Vafeiadis [54], the formal semantics of specifications is given by a family of predicates, $\mathit{safe}_n(c, s, \mu, Q)$, parametrized by $n \in \mathbb{N}$, that relate a command c, state (s, μ) and post-condition Q according to the informal explanation above. Once these predicates are defined, we define truth of specifications as follows:

$$\models \{P\}\ c\ \{Q\} \equiv_{\mathit{df}} \forall (s, \mu) \in \mathbf{State}, n \in \mathbb{N} : s, \mu, \mathbf{t} \models P \Rightarrow \mathit{safe}_n(c, s, \mu, Q).$$

Note that we only consider models that satisfy the pre-condition with complete write buffers—i.e., with $\gamma = \mathbf{t}$.

The formal definition of $\mathit{safe}_n(c, s, \mu, Q)$ is given by natural number induction on n. $\mathit{safe}_0(c, s, \mu, Q)$ holds always. And for $n \in \mathbb{N}$, $\mathit{safe}_{n+1}(c, s, \mu, Q)$ holds iff the following conditions are true:

1. If $c = \mathsf{skip}$ then $(s, \mu, \mathbf{t}) \models Q$.
2. For all μ_0, μ_F such that $\mu_0 \in (\mu_F \mathbin{\widetilde{\#}} \mu)$ it is the case that $c, (s, \mu_0) \not\rightarrow \lightning$.
3. For all $\mu_0, \mu_F, c', s', \mu_1$ such that

   (i) $\mu_0 \in (\mu_F \mathbin{\widetilde{\#}} \mu)$,

   (ii) $c, (s, \mu_0) \rightarrow c', (s', \mu_1)$,

   there exists μ'_F, μ' such that

   (a) $\mu_1 \in (\mu'_F \mathbin{\widetilde{\#}} \mu')$,

   (b) $\mu'_F \preceq \mu_F$, and

[8] It seems likely that $P \lhd\!\!- \mathbf{bar}$ is in fact the weakest pre-condition, but this has not been proved.

(c) $\mathit{safe}_n(c', s', \mu', Q)$.

The first part ensures that if a command is fully evaluated (i.e., is skip) after $n+1$ steps of evaluation, then the state satisfies the post-condition. The second part ensures that the command, when evaluated in any more completely described state—as defined by the spatiotemporal notion of separation—does not abort after $n+1$ steps of evaluation. The third part ensures that the command, if safely evaluated for n steps, preserves safety for one additional step. The definition of the predicate above differs significantly from Vafeiadis' in that the frame state μ_F is allowed to change from one step of evaluation to the next, but only by making silent transitions. This represents the fact that the flushing of buffered writes is nondeterministic and not controlled by the command. If the frame memory system μ_F contains buffered writes, they may well flush to memory during evaluation; and if those writes succeed buffered writes described by the local memory system μ, then those writes too must have been flushed.

Note that the two conditions for locality—the safety monotonicity and frame properties, as described in Section 3.4—are implicit in this definition. Safety monotonicity requires that if the command does not abort in a memory system μ then it also does not abort in a more completely defined memory system $\mu_F \mathbin{\widetilde{\#}} \mu$. The second part requires that the command not abort in $\mu_F \mathbin{\widetilde{\#}} \mu$ for any memory system μ_F. If the command did abort in memory system μ, the specification would simply not be true. Otherwise, the second part ensures that it also does not abort in any more completely defined memory system. The third part is actually somewhat weaker than the frame property, requiring only that if the command can take a step in a more complete state then the command remains safe for the local state, instead of the stronger requirement that the command may take an analogous step from the local state. But this condition is sufficient to show the soundness of the spatiotemporal frame rule. It is also important to note that we use only the spatiotemporal notion of separation in the definition of the safety predicate. The soundness of the frame rules for the stronger (spatial and temporal) conjunctions will later be shown to follow from this more general definition.

Chapter 4

A Concurrent Program Logic

This chapter describes a program logic for a parallel programming language modeled w.r.t. a weak-memory multiprocessor system model. This is a significant generalization of the program logic for a sequential language described in Chapter 3.

4.1 An Example Concurrent Proof

Consider the following simple program, c_s, which updates a single memory location while holding a global memory lock:

$$c_s =_{df} \mathsf{lock}_0 \,; [d] := 1_0 \,; \mathsf{unlock}_0 \,.$$

Each primitive command in this program is annotated with the processor identifier 0, which indicates that, on a multiprocessor machine, it will be executed on the 0^{th} processor. The program first acquires a global lock with the lock primitive, then stores the value 1 to the location given by d, and then finally releases the lock with the unlock primitive.

Next consider the program c_r, which reads the same memory location while holding the global memory lock:

$$c_r \;=_{df}\; \mathsf{lock}_1\,;x := [d]_1\,;\mathsf{unlock}_1\,;\mathsf{if}\ x = 1\ \mathsf{then}\,([d] := 2_1\,;\mathsf{fence}_1).$$

Here, the value in memory at address d is loaded into x after acquiring the lock and before releasing it; if the result of the load was 1 (i.e., if $x = 1$) the value 2 is written to address d and flushed back to memory. The primitive commands are annotated with the processor identifier 1, which indicates that the command will be executed on the 1^{st} processor on a multiprocessor machine.

The parallel composition of these commands, $c_s \parallel c_r$, is a very simple message-passing program. The sending thread c_s communicates to the receiving thread by setting a flag at address d. If the receiving thread loads the address d and observes that the flag has been set, then it knows something about the progress of the sending thread: namely, that it completed execution up to and including the location of the flag setting in the program order. In this particular program, if the receiver observes that the flag has been set then the receiver may claim sole ownership of the address d and may read and write to it without concern of future interference, and hence without first having acquired the global memory lock.

We might like to prove such a specification: i.e., assuming $d \mapsto 0$ and $x = 0$ holds initially, that the composition is safe and that $d \mapsto x{+}1$ holds upon termination:

$$\vdash\ \{x = 0 \wedge d \mapsto 0\}\ c_s \parallel c_r\ \{d \mapsto x + 1\} \tag{4.1}$$

In the concurrent program logic, however, specifications distinguish between private state, which is only accessible to the specified command, and shared state, which is accessible to the environment at large. Private state is still described with pre- and post-conditions, but the shared state is described using a single assertion, which must be maintained as an invariant throughout the specified command's execution.

We write $J \vdash \{P\}\ c\ \{Q\}$ for the specification of a (possibly) concurrent command c with shared state described by invariant assertion J, local pre-condition P and local post-condition Q. The specification in Equation 4.1 is completely private to the parallel command $c_s \parallel c_r$, and so the shared invariant is simply **emp**, an assertion that describes no memory addresses. Thus, the revised specification is:

$$\mathbf{emp} \vdash \{x = 0 \wedge d \mapsto 0\}\ c_s \parallel c_r\ \{d \mapsto x + 1\} \tag{4.2}$$

From the perspective of the constituent commands c_s and c_r, however, the memory at address d must be considered shared because neither command makes sole use of the memory at that address. Hence, to prove specifications of the constituent commands, we must describe the evolution of the memory at address d using an invariant assertion and then show that both commands maintain that invariant.

To aid this description, it can help to augment the commands with auxiliary program variables that track the progress of their evaluation. In this case, it helps to augment the sender thread c_s with a variable assignment that indicates that the memory update has taken place. The in the augmented thread c'_s below, s is an auxiliary variable, the assignment to s is considered to be an auxiliary assignment, and that assignment does not appear in the original thread c_s:

$$c'_s =_{df} \mathsf{lock}_0\,; [d] := 1_0\,; s := 1_0\,; \mathsf{unlock}_0$$

We can now provide an invariant that is maintained by both the augmented sender c'_s and the given receiver c_r. Informally, the evolution of the value and ownership of the memory at address d can be described as follows: the memory at address d is shared between the threads and has value s until the value of x becomes 1, when the receiver takes sole ownership of d. And, crucially, it is impossible for $x = 1$ and $s = 0$ simultaneously because the receiver will only take ownership if the sender has set the flag. So, more formally, either the receiver has not taken ownership and d is shared with value s—i.e., $x = 0 \wedge d \mapsto s$—or the receiver takes sole ownership with $x = 1$ and $s = 1$ and the address is no longer shared—i.e., $x = 1 \wedge \mathbf{emp}$. Hence, we define the invariant I as follows:

$$I =_{df} (x = 0 \wedge (s = 0 \vee s = 1) \wedge d \mapsto s) \vee (x = 1 \wedge s = 1 \wedge \mathbf{emp}).$$

We may then try to prove the following constituent specifications, which maintain the invariant I:

$$I \vdash \{s = 0 \wedge \mathbf{emp}\}\ c_s'\ \{s = 1 \wedge \mathbf{emp}\}$$
$$I \vdash \{x = 0 \wedge \mathbf{emp}\}\ c_r\ \{(x = 0 \wedge \mathbf{emp}) \vee (x = 1 \wedge d \mapsto 2)\}$$

The first specification asserts that the invariant I is maintained by c_s' and that, if evaluated in a state with $s = 0$ with no thread-private memory, the command also finishes evaluation with no private memory addresses and $s = 1$. The second specification asserts that the invariant I is maintained by c_r and that, if evaluated in a state with $r = 0$ and with no thread-private memory, the command either finishes with no thread-private memory (when $x = 0$) or with private ownership of d (when $x = 1$).

If we are successful in proving these specifications, then we may use an inference rule (PAR) to derive a specification for the parallel composition of commands whose individual specifications agree to maintain the same invariant, as do the above specifications of c_s' and c_r:

$$\dfrac{\dfrac{\vdots}{I \vdash \{P_s\}\ c_s'\ \{Q_s\}} \quad \dfrac{\vdots}{I \vdash \{P_r\}\ c_r\ \{Q_r\}}}{I \vdash \{P_s * P_r\}\ c_s' \parallel c_r\ \{Q_s * Q_r\}}\ \textsc{par}$$

For space reasons, we abbreviate the pre-conditions of c'_s and c_r as P_s and P_r respectively, and the post-conditions as Q_s and Q_r respectively, repeated below:

$$\begin{aligned}
P_s &=_{df} s = 0 \wedge \mathbf{emp} \\
P_r &=_{df} x = 0 \wedge \mathbf{emp} \\
Q_s &=_{df} s = 1 \wedge \mathbf{emp} \\
Q_r &=_{df} (x = 0 \wedge \mathbf{emp}) \vee (x = 1 \wedge d \mapsto 2).
\end{aligned}$$

Next, an inference rule (SHARE) may be applied that allows shared state to be considered as local to the specified commands:

$$\dfrac{\dfrac{\dfrac{\vdots}{I \vdash \{P_s\}\ c'_s\ \{Q_s\}} \quad \dfrac{\vdots}{I \vdash \{P_r\}\ c'_r\ \{Q_r\}}}{I \vdash \{P_s * P_r\}\ c'_s \parallel c'_r\ \{Q_s * Q_r\}}\ \textsc{par}}{\mathbf{emp} \vdash \{I * P_s * P_r\}\ c'_s \parallel c'_r\ \{I * Q_s * Q_r\}}\ \textsc{share}$$

The pre-condition can then be strengthened and the post-condition weakened using the rule of consequence (CONS) with the following semantic entailments:

$$\begin{aligned}
s = 0 \wedge x = 0 \wedge d \mapsto 0 &\models I * P_s * P_r \\
I * Q_s * Q_r &\models d \mapsto (x + 1)
\end{aligned}$$

With a final inference rule (AUX)[1] we remove any mention of auxiliary variables from the commands and assertions, giving the following derivation of the desired

[1]No attempt is made here to formalize the notion of auxiliary variable or the auxiliary variable elimination rule.

specification from Equation 4.2:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{I \vdash \{P_s\}\ c'_s\ \{Q_s\}} \quad \dfrac{\vdots}{I \vdash \{P_r\}\ c_r\ \{Q_r\}}}{I \vdash \{P_s * P_r\}\ c'_s \parallel c_r\ \{Q_s * Q_r\}}\ \text{PAR}}{\mathbf{emp} \vdash \{I * P_s * P_r\}\ c'_s \parallel c_r\ \{I * Q_s * Q_r\}}\ \text{SHARE}}{\mathbf{emp} \vdash \{s = 0 \wedge x = 0 \wedge d \mapsto 0\}\ c'_s \parallel c_r\ \{d \mapsto x + 1\}}\ \text{CONS}}{\mathbf{emp} \vdash \{x = 0 \wedge d \mapsto 0\}\ c_s \parallel c_r\ \{d \mapsto x + 1\}}\ \text{AUX}$$

It remains to show that the sequential commands c'_s and c_r satisfy their individual specifications, each maintaining the invariant I along the way. The specification of c'_s follows from an application of the inference rule (ATOMIC) for well-locked commands:

$$\dfrac{\mathbf{emp} \vdash \{(\mathbf{lock}_0 * I * P_s) \lhd \mathbf{bar}_0\}\ c\ \{(\mathbf{lock}_0 * I * Q_s) \mathrel{\lhd\!-} \mathbf{bar}_0\}}{I \vdash \{P_s\}\ \mathsf{lock}_0 \,;c\,; \mathsf{unlock}_0\ \{Q_s\}}\ \text{ATOMIC}$$

where we abbreviate $[d] := 1_0 \,; s := 1_0$ as c above.

This rule, in essence, asserts that if a command c can be shown to maintain an invariant as part of its local specification, then the corresponding locked command must also maintain the invariant as part of the shared state. In particular, because the introductory lock_0 command acquires the global lock and the final unlock_0 command releases it, the pre- and post-conditions both assert ownership of the lock with the $\mathbf{lock}_0$ assertion. While the lock is held the command may temporarily violate the invariant, so long as it is repaired by the time the command has finished executing and before releasing the lock with the unlock primitive.

Similarly, the pre- and post-condition both assert private ownership of the invariant I, along with but distinct from the local pre-condition P_s and post-condition Q_s. The lock_0 command implicitly fences, flushing any pending writes described by the invariant or local state, and so the entire pre-condition in the antecedent is succeeded by $\mathbf{bar}_0$. The unlock_0 command also implicitly fences, so the post-condition of the antecedent is generalized to include states that *would* satisfy the invariant and local post-condition if they were succeeded by $\mathbf{bar}_0$. This generalization is accomplished with the *temporal separating implication*, $P \mathrel{\lhd\!\!-} Q$, which describes states that satisfy P if they are temporally combined on the right with states that satisfy Q. For example, assertion $d \mapsto 0 \mathrel{\lhd\!\!-} \mathbf{bar}_0$ describes states which, when processor 0 is flushed, satisfy $d \mapsto 0$; that is, states which include any number of writes to d, followed by a write to d with value 1. A proof sketch of the antecedent follows below:

$$
\begin{array}{l}
\{(\mathbf{lock}_0 * I * P_s) \lhd \mathbf{bar}_0\} \therefore \\
\{(\mathbf{lock}_0 * (x = 0 \land d \mapsto s) * (s = 0 \land \mathbf{emp})) \lhd \mathbf{bar}_0\} \therefore \\
\{(\mathbf{lock}_0 * (x = 0 \land \mathbf{emp}) * d \mapsto -) \lhd \mathbf{bar}_0\} \therefore \\
\{\mathbf{lock}_0 * (x = 0 \land \mathbf{emp}) * d \mapsto -\} \\
\qquad [d] := 1_0 \\
\{\mathbf{lock}_0 * (x = 0 \land \mathbf{emp}) * d \mapsto - \lhd d \rightsquigarrow_0 1\} \\
\qquad s := 1 \\
\{\mathbf{lock}_0 * (x = 0 \land s = 1 \land \mathbf{emp}) * d \mapsto - \lhd d \rightsquigarrow_0 1\} \therefore \\
\{(\mathbf{lock}_0 * x = 0 \land s = 1 \land d \mapsto 1) \mathrel{\lhd\!\!-} \mathbf{bar}_0\} \therefore \\
\{(\mathbf{lock}_0 * (x = 0 \land \land d \mapsto s) * (s = 1 \land \mathbf{emp})) \mathrel{\lhd\!\!-} \mathbf{bar}_0\} \therefore \\
\{(\mathbf{lock}_0 * I * Q_s) \mathrel{\lhd\!\!-} \mathbf{bar}_0\}
\end{array}
$$

In the proof sketch above, we separate successive assertions with $\therefore$ to indicate implication—weakening the pre-conditions and strengthening the post-conditions found in the triples—allowed by the rule of consequence. In the first triple, the pre-condition describes the memory address d is allocated but has an unknown value ($d \mapsto -$), while the post-condition of the store command $[d] := 1_0$ describes the addition of a buffered write on processor 0 ($d \mapsto - \lhd d \rightsquigarrow_0 1$). The post-condition of the assignment command $s = 1$ simply asserts the equality. The final sequence of implications indicates that, once a barrier is performed, the buffered write will be flushed, yielding the correct value in memory.

The specification for c_r can be proved similarly to that of c'_s. The locked portion of the command is proved using the atomic rule; and the rest by basic sequential reasoning. A proof sketch of the specification of c_r follows below:

```
$\{x = 0 \wedge \mathbf{emp}\}$
    $\mathsf{lock}_1\,; x := [d]_1\,; \mathsf{unlock}_1$
$\{(x = 0 \wedge \mathbf{emp}) \vee (x = 1 \wedge d \mapsto 1)\}$
    $\mathsf{if}(x)$
    $\{x = 1 \wedge d \mapsto 1\}$
        $[d] := 2_1$
    $\{(x = 1 \wedge d \mapsto 1) \lhd d \rightsquigarrow_1 2\}$
        $\mathsf{fence}_1$
    $\{x = 1 \wedge d \mapsto 2\}$
$\{(x = 0 \wedge \mathbf{emp}) \vee (x = 1 \wedge d \mapsto 2)\}$
```

This completes the proof of the concurrent program specification of Equation 4.2.

4.2 Concurrent Programs

In this section we describe a simplified C-like structured programming language with concurrency. The primitive commands closely resemble the basic memory events described by the memory model, while the composite commands are typical for high-level languages. In particular, note that this is not an assembly language. This particular language of commands was chosen to be simple to reason about, but also at a suitable level of detail for describing concurrent data structures. Such algorithms are typically expressed using high level constructs like loops and if-then-else statements, along with basic atomic constructs like compare-and-swap, indication of where fencing is required, etc.

The concurrent language differs from the sequential language described in Section 3.3 in two ways. First, commands may be composed in “parallel”, which indicates that there are no program-order dependencies between the primitives of the constituent commands. Second, each primitive must be annotated with an expression that indicates the processor on which the primitive will be executed. As is discussed below, this will allow us to express both concurrent and interleaving parallelism uniformly.

In this setting, programs are identified with *commands*, which consist of various compositions of *primitive commands* for accessing and modifying state. In order to restrict the scope of the project, dynamic memory management commands (e.g., memory allocation and disposal) have been omitted.[2]

The primitive commands for the concurrent language are, roughly, a superset of those for the sequential language, as described in Section 3.3:

$$\begin{array}{rcl}\mathbf{PComm}\ p & ::= & \mathsf{skip} \mid \mathsf{assume}(b) \mid \mathsf{assert}(b) \mid x := e \mid x := [e] \mid \\ & & [e] := e' \mid \mathsf{fence} \mid \mathsf{lock} \mid \mathsf{unlock}\end{array}$$

Compared to the primitives in the sequential language, the lock and unlock commands are new to the concurrent language, and serve to manipulate a single, global lock, which can either be free (available) or busy (held by a particular processor).

The formal semantics of a successful execution step by a primitive command p is given as a transition relation, parametrized by a processor identifier i, between some machine states σ and σ':

$$p : \sigma \underset{i}{\rightarrow} \sigma'.$$

Such a quadruple may be informally interpreted to mean that when primitive p is

[2]This commands were considered in earlier iterations of this project [56, 57], and are discussed in Section 5.3

executed on processor i in machine state σ, it may evaluate in a single, atomic step to yield state σ'. (A formal interpretation will be given later in the context of a formal semantics of full commands.) This differs from the relation used to describe the semantics of the primitives in the sequential language, described in Section 3.3, by requiring that the processor on which the primitive is to execute be specified explicitly.

A primitive command, in state σ on processor i, may alternatively abort upon execution. This is indicated as follows

$$p : \sigma \underset{i}{\rightarrow} \lightning .$$

For example, an assertion may fail or a process may attempt to access an unallocated (from the process's perspective) memory address.

To define the combined primitive evaluation relation formally, we must first define the notions of *memory system* and *machine state*.

Definition 4. A *multiprocessor memory system* is a triple (h, B, K), where:

- $h : \mathbb{L} \rightharpoonup \mathbb{V}$ is a *heap*, i.e., a partial function that represents the allocated locations of shared memory and their values;
- $B : \mathbb{P} \rightarrow (\mathbb{L} \times \mathbb{V})\ \mathsf{list}$ is an array of write buffers; and
- $K : \mathcal{P}(\mathbb{P})$ is a set of blocked processors.

The set of multiprocessor memory systems is abbreviated as **Mem**.

A pair that consists of a stack s, as defined in Section 3.2, which assigns values to identifiers, and a memory system μ is called a *multiprocessor machine state*, typically abbreviated by σ. The collection of states is written **State**. We often abuse notation by interchanging memory systems and states in definitions for which the stack is irrelevant.

The notion of machine state given here differs from the structure used to define the memory model, as described in Section 2.3.2, in the following ways. First, names (i.e., "registers," "variables," "identifiers," etc.) are global instead of local to a particular processor. This is for convenience only, and is not an important technical restriction. The specification logic we describe later will be restricted to those programs for which the names are partitioned among processes, except for those that are never modified. Another reasonable choice would have been to use local names only, and to share read-only values among processes in the shared memory. This has the advantage of codifying the above healthiness condition on programs directly into the model of the language and logic, but it has the disadvantage of perhaps making the description of access to shared values (stored, e.g., in the heap) and local names (which must be parametrized by a processor name) slightly more awkward.

Second, and more significant, the global lock from the memory model description is replaced in the machine model by a "blocked" set K of processor identifiers. This set describes the processors that are not allowed, in the given state, to access main memory. In the memory model, an available lock corresponds to an empty blocked set $K = \emptyset$ in which no processors are blocked; a lock that is held by processor i corresponds to a blocked set $K = \mathbb{P} \setminus \{i\}$ that includes all processor identifiers except for i, because all processors other than i are blocked.

The machine model described here is thus more general than the model used to describe the memory model in that it allows, via the blocked set K, an arbitrary subset of the available processors to be blocked from accessing memory. This is as opposed to the use of a traditional lock object, which can only indicate that either no processors are blocked (when the lock is available) or all but a single processor i are blocked (when the lock is held by i). The blocked sets thus represent partial information about the status of the true machine state. In particular, if the blocked set is non-empty then, according to the memory model, the global lock must be held by some processor, although it may not be clear which processor specifically. We call a memory system *lock-complete* when its set of blocked processors is a valid representation of a global lock—insofar as no processors are blocked when the lock is free, and all processors but i are blocked when i holds the lock:

$$\textit{lock-complete}(h, B, k) \equiv_{df} k = \emptyset \lor \exists i \in \mathbb{P} : k = \mathbb{P} \setminus \{i\} .$$

If a memory system is not necessarily complete, then it is called *partial*.

The definition of the semantic relation for primitive commands, which is given w.r.t. machine states, is given in Figure 4.1 below. The memory systems are in general partial, though some primitives require completeness. The semantics of the primitives that are shared with the sequential language is similar to the description given in Section 3.3. The assume, assert and assignment primitives are exactly the same; the load, store and fence primitives operate w.r.t. the *specified* buffer within the buffer array B, as opposed to the single buffer b present in the uniprocessor state used to describe the sequential semantics. The lock command, w.r.t. processor i, changes an empty blocked set to one in which every processor except for i is blocked (i.e., $\mathbb{P} \setminus \{i\}$), which indicates that i alone holds the global lock. Furthermore, the lock command is only enabled when the buffer of the processor on which it executes is empty, which is analogous to an implicit fence instruction. Conversely, the unlock primitive changes a state in which every processor but i is blocked to one in which no processor is blocked (i.e., $\emptyset$). And, like the lock command, the unlock primitive includes an implicit fence.

Structured commands consist of either a primitive command; a sequential or concurrent composition of commands; a nondeterministic choice between commands; or an iteration of a command. We assume that, as commands, primitive commands are annotated with the name of the processor on which they are to be ex-

$$\frac{\text{if } \hat{s}(b) = 1}{\mathsf{assume}(b) : (s,h,B,K) \underset{i}{\rightarrow} (s,h,B,K)} \qquad (\text{P-ASSUME})$$

$$\frac{\text{if } \hat{s}(b) = 1}{\mathsf{assert}(b) : (s,h,B,K) \underset{i}{\rightarrow} (s,h,B,K)} \qquad (\text{P-ASSERT})$$

$$\frac{\text{if } \hat{s}(b) = 0}{\mathsf{assert}(b) : (s,h,B,K) \underset{i}{\rightarrow} \lightning} \qquad (\text{P-ASSERT-A})$$

$$\frac{}{x := e : (s,h,B,K) \underset{i}{\rightarrow} (s[x \leftarrow \hat{s}(e)]\,,h,B,K)} \qquad (\text{P-ASSIGN})$$

$$\frac{\text{if } (h \mathbin{\backslash\!\backslash} \overline{B(i)})(\hat{s}(e)) = v \text{ and } i \notin K}{x := [e] : (s,h,B,K) \underset{i}{\rightarrow} (s[x \leftarrow v]\,,h,B,K)} \qquad (\text{P-LOAD})$$

$$\frac{\text{if } (h \mathbin{\backslash\!\backslash} \overline{B(i)})(\hat{s}(e)) = \bot \text{ and } i \notin K}{x := [e] : (s,h,B,K) \underset{i}{\rightarrow} \lightning} \qquad (\text{P-LOAD-A})$$

$$\frac{\text{if } \hat{s}(e) \in \mathrm{dom}(h \mathbin{\backslash\!\backslash} \overline{B(i)})}{[e] := e' : (s,h,B,K) \underset{i}{\rightarrow} (s,h,B[i \leftarrow B(i) +\!\!+ [\hat{s}(e), \hat{s}(e')]]\,,K)} \qquad (\text{P-STORE})$$

$$\frac{\text{if } \hat{s}(e) \notin \mathrm{dom}(h \mathbin{\backslash\!\backslash} \overline{B(i)})}{[e] := e' : (s,h,B,K) \underset{i}{\rightarrow} \lightning} \qquad (\text{P-STORE-A})$$

$$\frac{\text{if } B(i) = \varepsilon}{\mathsf{fence} : (s,h,B,K) \underset{i}{\rightarrow} (s,h,B,K)} \qquad (\text{P-FENCE})$$

$$\frac{\text{if } B(i) = \varepsilon}{\mathsf{lock} : (s,h,B,\emptyset) \underset{i}{\rightarrow} (s,h,B,\mathbb{P} \setminus \{i\})} \qquad (\text{P-LOCK})$$

$$\frac{\text{if } B(i) = \varepsilon}{\mathsf{unlock} : (s,h,B,\mathbb{P} \setminus \{i\}) \underset{i}{\rightarrow} (s,h,B,\emptyset)} \qquad (\text{P-UNLOCK})$$

Figure 4.1: Semantics of concurrent primitive commands

ecuted. Presumably, the components of a sequential command will all be scheduled to execute on the same processor, but this is not required and the semantics handles all cases uniformly. This generality is not likely to be practically useful for sequential composition, but is important to the semantics of concurrent composition, as will be discussed shortly.

The formal language of commands is defined by the following grammar:

$$\textbf{Comm}\ c \ ::=\ p_e \mid (c\,;c') \mid (c+c') \mid c^* \mid (c \parallel c'),$$

where p is a primitive command and e is an expression that indicates a processor identifier.

The formal semantics of a single, atomic, successful step of execution by a command is given as a transition relation between command-state pairs:

$$c, \sigma \rightarrow c', \sigma'.$$

Such an entry may interpreted to mean that a command c in state σ may take a step of evaluation, transforming to command c' in an updated state σ'. But the evaluation of a command may abort unsuccessfully as well, as for primitive commands. Unsuccessful executions are modeled by transitions from command-state pairs to the erroneous pseudo-state $\lightning$, again as for primitive commands:

$$c, \sigma \rightarrow \lightning.$$

We refer collectively to command-state pairs and the erroneous state $\lightning$ as *configurations*, and use $\mathcal{C}$ to indicate a configuration.

The semantics of commands also encompasses "silent" transitions, which represent the flushing of buffered writes to the shared memory as allowed by the memory model. This flushing is described by a relation $\underset{\tau}{\rightarrow}$ between memory systems[3] defined as follows:

$$(h, B, K) \underset{\tau,i}{\rightarrow} (h[\ell \leftarrow v]\,, B[i \leftarrow b]\,, K) \equiv_{df} B(i) = [(\ell, v)] +\!\!+\, b \wedge i \notin K$$

$$\mu \underset{\tau}{\rightarrow} \mu' \equiv_{df} \exists i \in \mathbb{P} : \mu \underset{\tau,i}{\rightarrow} \mu'$$

We write $\preceq$ for the reflexive-transitive closure of the converse of $\underset{\tau}{\rightarrow}$:

$$\mu' \preceq \mu \equiv_{df} \mu \overset{*}{\underset{\tau}{\rightarrow}} \mu'.$$

The complete relation that defines the semantics of commands is given in Figure 4.2 below.

The reflexive-transitive closure of command evaluation semantics, written $c, \sigma \overset{*}{\rightarrow} \mathcal{C}$, is defined as usual.

Command Abbreviations A few standard command abbreviations are shown in Figure 4.3. Some would benefit greatly from local variable declarations, which have not yet been added to the language.

It is interesting to note that the execution of the "locked" command $\langle c \rangle_e$ is not atomic. Threads on other processors may concurrently read and write variables (i.e., registers) during the execution of $\langle c \rangle_e$, and also perform store operations, which add new writes to the write buffer. What threads on other processors may *not* do is load from memory—whether from their respective buffers or from the shared memory—or update main memory by flushing their buffers. It is conceivable that a theorem could be proved that shows the operational semantics presented in this

[3]As noted earlier, we abuse notation by interchanging the concept of state and memory system in definitions for which the stack is irrelevant. Hence, the definition of the relation between memory systems also constitutes the definition of a relation between states such that $(s, \mu) \underset{\tau}{\rightarrow} (s, \mu')$ iff $\mu \underset{\tau}{\rightarrow} \mu'$.

$$\frac{\text{if } p : \sigma \underset{\hat{s}(e)}{\longrightarrow} \sigma' \text{ and } \sigma' \in \mathbf{State}}{p_e, \sigma \to \mathsf{skip}, \sigma'} \quad (\textsc{c-prim})$$

$$\frac{\text{if } p : \sigma \underset{\hat{s}(e)}{\longrightarrow} \lightning}{p_e, \sigma \to \lightning} \quad (\textsc{c-prim-a})$$

$$\frac{c, \sigma \to c_0, \sigma'}{(c \parallel c'), \sigma \to (c_0 \parallel c'), \sigma'} \quad (\textsc{c-par-1})$$

$$\frac{\text{if } \sigma \underset{\tau}{\longrightarrow} \sigma'}{c, \sigma \to c, \sigma'} \quad (\textsc{c-tau})$$

$$\frac{c, \sigma \to \lightning}{(c \parallel c'), \sigma \to \lightning} \quad (\textsc{c-par-1a})$$

$$\frac{c, \sigma \to c_0, \sigma'}{(c\,; c'), \sigma \to (c_0\,; c'), \sigma'} \quad (\textsc{c-seq})$$

$$\frac{}{(\mathsf{skip} \parallel c'), \sigma \to c', \sigma} \quad (\textsc{c-par-1s})$$

$$\frac{c, \sigma \to \lightning}{(c\,; c'), \sigma \to \lightning} \quad (\textsc{c-seq-a})$$

$$\frac{c', \sigma \to c_0, \sigma'}{(c \parallel c'), \sigma \to (c \parallel c_0), \sigma'} \quad (\textsc{c-par-2})$$

$$\frac{}{(\mathsf{skip}\,; c'), \sigma \to c', \sigma} \quad (\textsc{c-seq-s})$$

$$\frac{c', \sigma \to \lightning}{(c \parallel c'), \sigma \to \lightning} \quad (\textsc{c-par-2a})$$

$$\frac{}{(c + c'), \sigma \to c, \sigma} \quad (\textsc{c-ch-1})$$

$$\frac{}{(c \parallel \mathsf{skip}), \sigma \to c, \sigma} \quad (\textsc{c-par-2s})$$

$$\frac{}{(c + c'), \sigma \to c', \sigma} \quad (\textsc{c-ch-2})$$

$$\frac{}{c^*, \sigma \to (\mathsf{skip} + (c\,; c^*)), \sigma} \quad (\textsc{c-loop})$$

Figure 4.2: Semantics of concurrent commands

section is equivalent to one in which locked commands have a truly atomic semantics, but this has not been attempted.

Stability Consider a state $\sigma_0 = (s, h, B, \emptyset)$ in which $h = \ell \mapsto 0$, $B(j) = [(\ell, 1)]$ and $B(x) = \varepsilon$ for all $x \neq j$. From this state, a load on processor i may evaluate as follows:

$$y := [\ell]_i\,, (s, h, B, \emptyset) \to \mathsf{skip}, (s[y \leftarrow 0]\,, h, B, \emptyset).$$

Because j is not blocked, it is also possible for a flushing operation to take place:

$$y := [\ell]_i\,, (s, h, B, \emptyset) \to y := [\ell]_i\,, (s, h[\ell \leftarrow 1]\,, B[j \leftarrow \varepsilon]\,, \emptyset),$$

$$\begin{aligned}
\text{if } b \text{ then } c \text{ else } c' &=_{df} (\mathsf{assume}(b)\,;c) + (\mathsf{assume}(!b)\,;c')\\
\text{if } b \text{ then } c &=_{df} (\mathsf{assume}(b)\,;c) + (\mathsf{assume}(!b)\,;\mathsf{skip})\\
\text{while } b \text{ do } c &=_{df} (\mathsf{assume}(b)\,;c)^*\,;\mathsf{assume}(!b)\\
\langle c\rangle_e &=_{df} \mathsf{lock}_e\,;c\,;\mathsf{unlock}_e\\
\mathsf{inc}_e(e') &=_{df} \big\langle(x := [e']\,;[e'] := x+1)\big\rangle_e\\
\mathsf{cas}_e(f,g,g') &=_{df} \big\langle(x := [f]\,;\text{if } x = g \text{ then } [f] := g' \text{ else } r := 0)\big\rangle_e
\end{aligned}$$

Figure 4.3: Concurrent command abbreviations

and afterward for the load to evaluate as follows:

$$y := [\ell]_i\,,(s, h[\ell \leftarrow 1]\,, B[j \leftarrow \varepsilon]\,, \emptyset) \to \mathsf{skip}(s[y \leftarrow 1]\,, h[\ell \leftarrow 1]\,, B[j \leftarrow \varepsilon]\,, \emptyset).$$

Note that in the first evaluation the load resolves ℓ to 0, and in the second evaluation it resolves ℓ to 1, with the distinguishing characteristic of the latter being the preceding nondeterministic flushing operation.

By contrast, from the state $\sigma_1 = (s, h, B, \{j\})$, where h and B are defined as in σ_0, the *only* reduction of $y := [\ell]_i$ is:

$$y := [\ell]_i\,,(s, h, B, \emptyset) \to \mathsf{skip}, (s[y \leftarrow 0]\,, h, B, \emptyset).$$

This is because processor j is blocked, and so its buffered write may not commit to memory. As a consequence, it is not possible for the load on processor i to observe the write buffered on processor j.

We say that location ℓ is *unstable* in state σ_0 for processor i because the result of loading ℓ is determined by the relative ordering of the flushing operations. On the other hand, ℓ is *stable* in σ_1 for i because the load of ℓ is oblivious to the flushing operations.

A state is called *coherent* if each location has writes buffered by at most one processor:

$$\forall i, j \in \mathbb{P} \setminus K : i \neq j \Rightarrow \mathrm{dom}(B(i)) \cap \mathrm{dom}(B(j)) = \emptyset \,.$$

The memory locations in a coherent state may be partitioned among the processors, such that the locations of a partition are all stable for their respective processor.

Interleaving versus Parallelism A pleasant property of this semantics is the uniform description of both interleaving and parallel concurrency. Let c_i be a sequential command c in which each primitive has processor annotation i. Then, e.g., $(c_1 \parallel c'_1)$ describes the interleaving concurrent execution of commands c and c' on processor 1, while $(c_1 \parallel c'_2)$ describes the parallel concurrent execution of c and c' on processors 1 and 2, respectively. But one does not typically have control over the particular processor on which a command executes (e.g., c_1 instead of c_2). Thus, $(c_x \parallel c'_x)$ describes the interleaving concurrent execution of commands c and c' on some individual processor, denoted by the free variable x. For $x \neq y$, $(c_x \parallel c'_y)$ describes the parallel concurrent execution of c and c' on distinct processors given by x and y respectively. Furthermore, without any assumptions about the relationship between x and y, $(c_x \parallel c'_y)$ describes both interleaving and parallel executions of c and c'. This presumably is the most common situation with concurrent composition: it is up to the operating system to assign processors to threads, and correctness of a program ought to encompass any such assignment.

Static Semantics The static semantics of expressions, primitive commands and commands, embodied here by functions $\mathrm{fv}(-)$ and $\mathrm{mod}(-)$ associating these objects to their sets of free and modified variables, respectively, are completely standard. (Especially so because there is are no name-hiding operations in the language, like the aforementioned missing local variable declaration command.) For example, $\mathrm{fv}(x := [y+1]_z) = \{x, y, z\}$ and $\mathrm{mod}(x := [y+1]_z) = \{x\}$.

4.3 Separation

In this section we define a series of notions of separation, analogous to those defined in Chapter 3, but for multiprocessor states. As in the sequential, uniprocessor case, the goal for these notions of separation is to allow for local reasoning, by modeling separating conjunctions that have sound frame rules.

Unlike in the previous chapter, however, in which we defined separation in terms of memory systems and then later lifted those definitions to generalized memory systems, here we wish to define the notions of separation directly in terms of *generalized multiprocessor memory systems*, which are defined as follows.

Definition 5. A *generalized multiprocessor memory system* (typically indicated by the symbol ν) is a four-tuple, (h, B, K, Γ), where:

- (h, B, K) is a multiprocessor memory system; and
- $\Gamma : \mathcal{P}(\mathbb{P})$ is a *buffer-completeness* set.

We furthermore require that if $\Gamma = \emptyset$ then so is $h = \emptyset$.

Generalized multiprocessor memory systems are related to multiprocessor memory systems, which model parallel programs, in the same way that generalized sequential memory systems are related to uniprocessor memory systems, which model purely sequential programs. In particular, the buffer-completeness set Γ in the generalized multiprocessor memory system plays a role analogous to the buffer-completeness flag γ in the generalized uniprocessor memory system: inclusion of a processor identifier i in the set Γ means that the ith buffer is complete, and therefore there may exist no previous *buffered* writes on the ith write buffer; they must instead have been flushed to memory.

In the uniprocessor case, we required that if $\gamma = \mathbf{f}$ then $h = \emptyset$, because flushing can only occur when the buffer is complete. In the multiprocessor case, we require that $h = \emptyset$ only when *none* of the buffers are complete. Alternately, the heap may be non-empty as long as at least some of the buffers are complete.

We call a generalized multiprocessor memory system (h, B, K, Γ) *buffer-complete* when $\Gamma = \mathbb{P}$; i.e., when each buffer is complete. The semantics of assertions (given later in Section 4.4) shall be described in terms of generalized multiprocessor memory systems which are not necessarily buffer-complete, but the specifications (given later in Section 4.5) shall be described in terms of only buffer-complete generalized multiprocessor memory systems.

We now proceed with the definition of notions of separation in terms of generalized multiprocessor memory systems.

4.3.1 Spatial Separation

As in the uniprocessor case, described in Section 3.4.1, we begin with the simplest notion of separation, *spatial separation*, which describes decompositions of memory systems with disjoint sets of allocated locations. We write $\mu \in \mu_0 \mathbin{\widetilde{*}} \mu_1$ when memory system μ may be spatially separated in terms of memory systems μ_0 and μ_1. Similarly, we write $\nu \in \nu_0 \mathbin{\widehat{*}} \nu_1$ when generalized memory system ν may be spatially separated in terms of generalized memory systems ν_0 and ν_1. The latter operation on generalized memory systems will be defined in terms of the former operation on memory systems; hence we proceed to first define $\mu_0 \mathbin{\widetilde{*}} \mu_1$.

Informally, the spatial separation $\mu_0 \mathbin{\widetilde{*}} \mu_1$ denotes memory systems in which the write buffers of μ_0 and μ_1 are pointwise-interleaved: i.e., the ith buffer is some interleaving of the ith buffer of μ_0 and the ith buffer of μ_1. The formal definition $\mu_0 \mathbin{\widetilde{*}} \mu_1$ is as follows, where $\mu_0 = (h_0, B_0, K_0)$ and $\mu_1 = (h_1, B_1, K_1)$:

$$\mu_0 \mathbin{\widetilde{*}} \mu_1 =_{df} \{(h_0 \uplus h_1, B, K_0 \cup K_1) \mid B \in B_0 \uplus B_1 \wedge \mu_0 \smile_* \mu_1\},$$

where the *spatial compatibility relation* between multiprocessor memory systems

$\mu_0 \smile_* \mu_1$ is defined as follows:

$$\mu_0 \smile_* \mu_1 \;\equiv_{df}\; (\mathrm{dom}(h_0) \cup \mathrm{dom}(B_0)) \cap (\mathrm{dom}(h_1) \cup \mathrm{dom}(B_1)) = K_0 \cap K_1 = \emptyset\,.$$

In both definitions, we lift operations on lists to the analogous operations on functions into lists. That is, the domain of a buffer array B is the union of the domains of all the buffers in the array:

$$\mathrm{dom}(B) \;=_{df}\; \bigcup_{i \in \mathbb{P}} \mathrm{dom}(B(i)),$$

and the interleavings of buffer arrays B_0 and B_1 are buffers arrays for which all buffers are interleavings of their respective buffers in arrays B_0 and B_1:

$$B \in (B_0 \uplus B_1) \;\equiv_{df}\; \forall i \in \mathbb{P} : B(i) \in B_0(i) \uplus B_1(i).$$

It is easy to check that $\mu_0 \mathbin{\widetilde{*}} \mu_1$ yields a set of multiprocessor memory systems: because the compatibility relation requires disjointness of the domains, $h_0 \uplus h_1$ is well defined according to the definition of partial function summation in Section 2.1.2.

Next, we define the spatial separation of generalized memory system $\nu_0 \mathbin{\widehat{*}} \nu_1$ by lifting the spatial separation of the included memory systems and unioning the buffer-completeness sets. That is, for $\nu_0 = (\mu_0, \Gamma_0)$ and $\nu_1 = (\mu_1, \Gamma_1)$, the set $\nu_0 \mathbin{\widehat{*}} \nu_1$ is defined as follows

$$\nu_0 \mathbin{\widehat{*}} \nu_1 \;=_{df}\; \{(\mu, \Gamma_0 \cup \Gamma_1) \mid \mu \in \mu_0 \mathbin{\widetilde{*}} \mu_1\}\,.$$

Again, it is easy to check that $\nu_0 \mathbin{\widehat{*}} \nu_1$ yields a set of generalized multiprocessor memory systems: $h_0 \cup h_1 = \emptyset$ if $\Gamma_0 \cup \Gamma_1 = \emptyset$, as required by the definition.

The blocking sets and buffer-completeness sets in the separation are defined by union because, intuitively, the properties they describe can only accumulate with increasingly complete descriptions of memory systems. If a processor is blocked in some memory system, then it ought also to be blocked in a more complete description of a memory system—along with, perhaps, the requirement that some additional processors be blocked. Similarly, if a processor is buffer-complete in some partial description of a memory system, then it ought also to be buffer-complete in a more thorough description of that memory system. The requirement that blocking sets be disjoint is later used to give especially “small” axioms for the for the lock manipulation primitives.

Let us consider a small example. In the sequel, let $\mathcal{E} =_{df} \lambda x \,.\, \varepsilon$, the array of empty write buffers. Let $\nu_0 = (1 \mapsto 1, \mathcal{E}[0 \leftarrow [(1,2)]], \emptyset, \{0\})$ and $\nu_1 = (2 \mapsto 1, 1[(2,2)], \emptyset, \{1\})$ be generalized multiprocessor memory systems. Here, ν_0 consists of a single-point heap describing the memory address 1 with value 1; a write buffer array that is everywhere empty except for processor 0, which has a single write to address 1 with value 2; no blocked processors; and with processor 0 declared to be buffer-complete. Similarly, ν_1 consists of a single-point heap describing the memory address 2 with value 1; a write buffer array that is everywhere empty except for processor 1, which has a single write to address 2 with value 2; no blocked processors; and with processor 1 declared to be buffer-complete. Note that $\nu_0 \smile_* \nu_1$ holds because the domains of the memory systems are disjoint, as well as the blocked sets. Consequently, the set $\nu_0 \mathbin{\widehat{*}} \nu_1$ is non-empty; indeed, it is a singleton:

$$\nu_0 \mathbin{\widehat{*}} \nu_1 = \{1 \mapsto 1 \uplus 2 \mapsto 1, \mathcal{E}[0 \leftarrow [(1,2)], 1 \leftarrow [(2,2)]], \emptyset, \{0,1\}\},$$

in which the only resulting memory system has a two-point heap with addresses 1 and 2 defined; a buffer array that is empty everywhere except for processors 0 and 1; an empty blocking set; and in which processors 0 and 1 are buffer-complete. The set $\nu_0 \mathbin{\widehat{*}} \nu_1$ is a singleton because there is only a single possible interleaving of the memory systems' respective buffer arrays:

$$\mathcal{E}[0 \leftarrow [(1,2)]] \uplus \mathcal{E}[1 \leftarrow [(2,2)]] = \{\mathcal{E}[0 \leftarrow [(1,2)]\,, 1 \leftarrow [(2,2)]]\}\,.$$

Note that the result of a load in memory system ν_1 is the same as in any of the more completely defined memory systems of $\nu_0 \mathbin{\widehat{*}} \nu_1$. Note also that neither $\nu_0 \smile_* \nu_0$ nor $\nu_1 \smile_* \nu_1$ hold because in each case the domains overlap, and so $\nu_0 \mathbin{\widehat{*}} \nu_0 = \nu_1 \mathbin{\widehat{*}} \nu_1 = \emptyset$.

We overload for convenience the symbol $\widehat{*}$ to indicate the pointwise lifting of this function to sets of memory systems:

$$S_1 \mathbin{\widehat{*}} S_2 \;=_{df}\; \cup\{\nu_1 \mathbin{\widehat{*}} \nu_2 \mid \nu_1 \in S_1 \wedge \nu_2 \in S_2 \wedge \nu_1 \smile_* \nu_2\}\,.$$

We use these functions interchangeably when the intended meaning is clear from context, e.g.:

$$\nu_1 \mathbin{\widehat{*}} (\nu_2 \mathbin{\widehat{*}} \nu_3) = \cup\{\nu_1 \mathbin{\widehat{*}} \nu_{23} \mid \nu_{23} \in \nu_2 \mathbin{\widehat{*}} \nu_3\}\,.$$

In the following, let $\nu_u =_{df} (\emptyset, \mathcal{E}, \emptyset, \emptyset)$ be an empty generalized memory system. The following lemma asserts some algebraic properties of spatial separation.

Proposition 5. *For generalized multiprocessor memory systems ν_0, ν_1, ν_2:*

- $\nu_u \mathbin{\widehat{*}} \nu_0 = \{\nu_0\}$
- $\nu_0 \mathbin{\widehat{*}} \nu_1 = \nu_1 \mathbin{\widehat{*}} \nu_0$
- $\nu_0 \mathbin{\widehat{*}} (\nu_1 \mathbin{\widehat{*}} \nu_2) = (\nu_0 \mathbin{\widehat{*}} \nu_1) \mathbin{\widehat{*}} \nu_2$

4.3.2 Temporal Separation

As in the sequential, uniprocessor case, we also define a temporal notion of memory system separation, which we write $\mu_0 \mathrel{\widetilde{\lhd}} \mu_1$, and a temporal notion of generalized memory system separation, which we write $\nu_0 \mathrel{\widehat{\lhd}} \nu_1$. Informally, a temporal separation of memory systems partitions the writes of *each* write buffer. Unlike spatial separation, temporal separation does not require disjointness of the domains of memory addresses described by the states; so it can be used to describe sequences of writes (on a single buffer) to a particular memory address. However, to ensure locality and a sound (left-side) frame rule, the definedness condition for temporal separation must rule out, in some cases, memory systems with pending writes to a particular location on multiple write buffers.

Formally, the partial function $\mu_0 \mathrel{\widetilde{\lhd}} \mu_1$ is defined as follows, where $\mu_0 = (h_0, B_0, K_0)$ and $\nu_1 = (h_1, B_1, K_1)$:

$$\mu_0 \mathrel{\widetilde{\lhd}} \mu_1 =_{df} \begin{cases} (h_0 \backslash\!\backslash h_1, B_0 \mathbin{+\!\!+} B_1, K_0 \cup K_1) & \text{if } \mu_0 \smile_{\lhd} \mu_1 \\ \bot & \text{otherwise} \end{cases}$$

where the *temporal compatibility relation* $\smile_{\lhd}$ is defined below. In $\mu_0 \mathrel{\widehat{\lhd}} \mu_1$, the heap values defined in the later memory system override those in the earlier memory system, and the writes in each buffer of the earlier memory system are prepended to those in the later memory system. Above, we lift list concatenation to functions into lists as follows:

$$B_0 \mathbin{+\!\!+} B_1 =_{df} \lambda x \,.\, B_0(x) \mathbin{+\!\!+} B_1(x).$$

The temporal compatibility relation $\smile_{\lhd}$ is defined as follows:

$$\begin{aligned} \mu_0 \smile_{\lhd} \mu_1 \equiv_{df} {} & K_0 \cap K_1 = \emptyset \wedge \\ & \forall i \in \mathbb{P} \setminus (K_0 \cup K_1) : \mathrm{dom}(B_0(i)) \cap \mathrm{dom}(h_1) = \emptyset \wedge \\ & \forall i, j \in \mathbb{P} \setminus (K_0 \cup K_1) : i \neq j \Rightarrow \mathrm{dom}(B_0(i)) \cap \mathrm{dom}(B_1(j)) = \emptyset \,. \end{aligned}$$

The first conjunct above requires, as in the case of spatial separation, disjointness of the blocking sets. The second and third conjuncts describe the weak disjointness requirements required for soundness of the temporal frame rule. Specifically, the second requires that there be no preceding buffered writes to locations defined in the given memory, unless the buffers on which those writes exist are blocked. Similarly, the third conjunct requires that there be no preceding buffered writes to locations for which there currently pending writes, unless either those earlier pending writes are on the same buffer, or those earlier pending writes are on blocked buffers. Examples presented shortly will illustrate the necessity of these definedness conditions.

The temporal separation of generalized memory systems, $\nu_0 \mathbin{\widehat{\lhd}} \nu_1$ is defined as follows, where $\nu_0 = (\mu_0, \Gamma_0)$, $\mu_0 = (h_0, B_0, K_0)$ and $\nu_1 = (\mu_1, \Gamma_1)$:

$$\nu_0 \mathbin{\widehat{\lhd}} \nu_1 =_{df} \begin{cases} (\mu_0 \mathbin{\widetilde{\lhd}} \mu_1, \Gamma_0 \cup \Gamma_1) & \text{if } \mu_0 \smile_{\lhd} \mu_1 \text{ and } \forall i \in \Gamma_1 : B_0(i) = \varepsilon \\ \bot & \text{otherwise.} \end{cases}$$

The additional definedness condition in the temporal separation of generalized memory systems requires that, for the buffer-complete processors i of ν_1, any preceding writes (in ν_0) described on buffer i be flushed to memory.

The importance of the second and third conjuncts in the temporal compatibility relation above can be demonstrated with a few small examples. First, consider $\nu_1 = (1 \mapsto 2, \mathcal{E}, \emptyset, \{1\})$. A load by processor 1 of address 1 in this memory system clearly results in the value 2. Next consider $\nu_0 = (\emptyset, \mathcal{E}[0 \leftarrow [(1,3)]], \emptyset, \{0\})$, which includes a buffered write on processor 0 to the same memory address 1. The memory system ν_0 is not compatible with ν_1 due to the second conjunct in the temporal compatibility relation because processor 0 is not blocked in ν_0, but contains a buffered write to a location defined in the heap of ν_1. If this condition were omitted, then the value of $\nu_0 \mathbin{\widehat{\lhd}} \nu_1$ would be as follows:

$$\nu_0 \mathbin{\widehat{\lhd}} \nu_1 = (1 \mapsto 2, \mathcal{E}[0 \leftarrow [(1,3)]], \emptyset, \{0,1\}).$$

Although in the memory system $\nu_0 \mathbin{\widehat{\lhd}} \nu_1$ defined above it remains the case that a load by processor 1 of address 1 again results in the value 2, this is not a *stable* load: the write buffered on processor 0 may flush to memory before the load completes, resulting in the state $(1 \mapsto 3, \mathcal{E}, \emptyset, \{0,1\})$, from which a load by processor 1 certainly would not result in the value 2. This is a situation we wish to avoid, and so we rule out temporal conjunctions of memory systems like ν_0.

On the other hand, the temporal conjunction $\nu_0' \mathbin{\widehat{\lhd}} \nu_1$, with memory system $\nu_0' = (\emptyset, \mathcal{E}[0 \leftarrow [(1,3)]], \{0\}, \{0\})$, is perfectly safe because processor 2, on which the offending write is buffered, is blocked, and so there is no concern that it will flush to memory, becoming visible to other processors, and causing instability. Hence, the second conjunct above only rules out writes to conflicting locations on non-blocked write buffers.

The role of the third conjunct in the temporal compatibility relation is similar: to rule out temporal conjunctions that lead to instability. Consider now $\nu_1' = (\emptyset, \mathcal{E}[1 \leftarrow [(1,2)]], \emptyset, \{1\})$, which differs from ν_1 above in that the write that was flushed there is here still pending in the first buffer. Again, ν_0 is incompatible, by the third conjunct, with ν_1' because the buffered write on processor 0 in ν_0 conflicts with the buffered write on processor 1 in ν_1'. If, however, these memory systems were temporally compatible—i.e., if the third conjunct above were omitted in the temporal compatibility relation—then the value of $\nu_0 \mathbin{\widehat{\lhd}} \nu_1'$ would be as follows:

$$\nu_0 \mathbin{\widehat{\lhd}} \nu_1' = (\emptyset, \mathcal{E}[0 \leftarrow [(1,3)], 1 \leftarrow [(1,2)]], \emptyset, \{0,1\}).$$

The address 1 in this memory system is, again, unstable w.r.t. processor 1: if the write on buffer 1 flushes to memory, followed by the write on buffer 0 flushing to memory, a load on processor 1 might result in value 3 instead of 2. However, $\nu_0' \mathbin{\widehat{\lhd}} \nu_1'$ is well defined because the buffered write on processor 0 in memory system ν_0' is blocked, and so it is impossible for processor 1 to observe that write.

Note that writes committed to the memory are combined by the temporal separator using heap overriding (first defined in Section 2.1.2). This reflects the fact that previous committed writes are subsumed by more recent flushed writes. For example, consider $\nu_0 = (1 \mapsto 2, \mathcal{E}, \emptyset, \{0\})$ and $\nu_1 = (1 \mapsto 3, \mathcal{E}, \emptyset, \{0\})$. In the temporal separation $\nu_0 \mathbin{\widehat{\lhd}} \nu_1$, the later write overrides the earlier write because $(1 \mapsto 2) \mathbin{\backslash\!\!\backslash} (1 \mapsto 3) = 1 \mapsto 3$:

$$\nu_0 \mathbin{\widehat{\lhd}} \nu_1 = (1 \mapsto 3, \mathcal{E}, \emptyset, \{0\}).$$

Also note that, because there can be no ordering among writes on distinct buffers, none is created by temporal separation of such memory systems. For example, because $\nu_0' = (\emptyset, \mathcal{E}[0 \leftarrow [(1,3)]], \emptyset, \emptyset)$ and $\nu_1' = (\emptyset, \mathcal{E}[1 \leftarrow [(2,4)]], \emptyset, \emptyset)$ have writes only on distinct buffers, their temporal separation is symmetric:

$$\nu_0' \mathbin{\widehat{\lhd}} \nu_1' = (\emptyset, \mathcal{E}[0 \leftarrow [(1,3)], 1 \leftarrow [(2,4)]], \emptyset, \emptyset) = \nu_1' \mathbin{\widehat{\lhd}} \nu_0'.$$

In the following, let $\nu_u =_{df} (\emptyset, \mathcal{E}, \emptyset, \emptyset)$ be an empty generalized memory system as before. The following lemma asserts some algebraic properties of temporal separation.

Proposition 6. *For generalized multiprocessor memory systems ν_0, ν_1, ν_2:*

- $\nu_u \mathbin{\widehat{\lhd}} \nu_0 = \nu_0 \mathbin{\widehat{\lhd}} \nu_u = \nu_0$
- $\nu_0 \mathbin{\widehat{\lhd}} (\nu_1 \mathbin{\widehat{\lhd}} \nu_2) = (\nu_0 \mathbin{\widehat{\lhd}} \nu_1) \mathbin{\widehat{\lhd}} \nu_2$

Remark. Unlike traditional notions of separation (e.g., as outlined in work on abstract separation logic [11]), note that temporal separation is not a cancellative operation. That is, it is not the case that the following implication holds:

$$\mu \mathbin{\widetilde{\lhd}} \mu_0 = \mu \mathbin{\widetilde{\lhd}} \mu_1 \Rightarrow \mu_0 = \mu_1.$$

A simple counter-example is $\mu = \mu_0 = (1 \mapsto 2, \mathcal{E}, \emptyset)$ and $\mu_1 = (\emptyset, \mathcal{E}, \emptyset)$. Then $\mu \mathbin{\widetilde{\lhd}} \mu_0 = \mu = \mu \mathbin{\widetilde{\lhd}} \mu_1$, but obviously $\mu_0 \neq \mu_1$.

Strong Temporal Separation

As in the sequential case, we may define a strong variant of temporal separation—that requires both spatial and temporal separation—as the intersection of spatial and temporal separators. For $\nu_0 = (\mu_0, \Gamma_0)$ and $\nu_1 = (\mu_1, \Gamma_1)$, the strong temporal separation $\nu_0 \mathbin{\widehat{\blacktriangleleft}} \nu_1$ is defined as follows:

$$\nu_0 \mathbin{\widehat{\blacktriangleleft}} \nu_1 =_{df} \begin{cases} \nu_0 \mathbin{\widehat{\lhd}} \nu_1 & \text{if } \mu_0 \smile_{\lhd} \mu_1 \text{ and } \mu_0 \smile_{*} \mu_1 \\ \bot & \text{otherwise.} \end{cases}$$

Observe that if $\nu_0 \mathbin{\widehat{\blacktriangleleft}} \nu_1$ is defined then $\nu_0 \mathbin{\widehat{\lhd}} \nu_1$ is defined and is identical, and $\nu_0 \mathbin{\widehat{*}} \nu_1$ is defined with $\nu_0 \mathbin{\widehat{\blacktriangleleft}} \nu_1 \in \nu_0 \mathbin{\widehat{*}} \nu_1$.

4.3.3 Spatiotemporal Separation

Finally, we again define a *spatiotemporal* separation function on memory systems $\mu_0 \mathbin{\widetilde{\#}} \mu_1$, and a spatiotemporal separation function of generalized memory systems $\nu_0 \mathbin{\widehat{\#}} \nu_1$—so-called because it subsumes both spatial and temporal separation. Spatiotemporal separation subsumes spatial and temporal separation in the sense that, if $\nu_0 \mathbin{\widehat{\lhd}} \nu_1$ is defined, then $\nu_0 \mathbin{\widehat{\lhd}} \nu_1 \in \nu_0 \mathbin{\widehat{\#}} \nu_1$ and, similarly, $\nu_0 \mathbin{\widehat{*}} \nu_1 \subseteq \nu_0 \mathbin{\widehat{\#}} \nu_1$. This separation function is intended to be as weak as possible, describing a wide variety of memory systems while still maintaining locality w.r.t. the concurrent programming language. This weakness reduces the expressive power of the function, which limits its usefulness for describing particular sets of states like pre- and post-conditions (i.e., for modeling assertions). The weakness of spatiotemporal separation shall later (in Section 4.5.2) be extremely useful, though, for giving a semantics to specifications by outlining liberal conditions for sound frame rules. That is, specifications will later be considered true when they soundly admit spatiotemporal frames; and because spatiotemporal separation subsumes both spatial and temporal separation, specifications will also be shown to soundly admit spatial frames and temporal frames.

For memory systems $\mu_0 = (h_0, B_0, K_0)$ and $\mu_1 = (h_1, B_1, K_1)$, the spatiotemporal separation $\mu_0 \mathbin{\widetilde{\#}} \mu_1$ is defined as follows:

$$\mu_0 \mathbin{\widetilde{\#}} \mu_1 =_{df} \{(h_0 \backslash\!\!\backslash h_1, B, K_0 \cup K_1) \mid B \in B_0 \backslash\!\!\backslash B_1 \wedge \mu_0 \smile_{\lhd} \mu_1\},$$

in which the overriding operation on lists $b_0 \backslash\!\!\backslash b_1$ is lifted pointwise to functions into lists $B_0 \backslash\!\!\backslash B_1$:

$$B \in B_0 \backslash\!\!\backslash B_1 \equiv_{df} \forall i \in \mathbb{P} : B(i) \in B_0(i) \backslash\!\!\backslash B_1(i).$$

Note that the *temporal* compatibility relation is reused in the definition above.

The spatiotemporal separation $\nu_0 \mathbin{\widehat{\#}} \nu_1$ of generalized memory systems $\nu_0 = (\mu_0, \Gamma_0)$ and $\nu_1 = (\mu_1, \Gamma_1)$ is defined by lifting the spatiotemporal separation of memory systems and unioning the buffer-completeness sets as follows:

$$\nu_0 \mathbin{\widehat{\#}} \nu_1 =_{df} \left\{(\mu, \Gamma_0 \cup \Gamma_1) \mid \mu \in \mu_0 \mathbin{\widetilde{\#}} \mu_1\right\}.$$

It is easy to see that, when $\mu_0 \smile_{\lhd} \mu_1$ holds, $\mu_0 \mathbin{\widetilde{\lhd}} \mu_1 \in \mu_0 \mathbin{\widetilde{\#}} \mu_1$ because $B_0 +\!\!\!+ B_1$ is a member of the overriding $B_0 \backslash\!\!\backslash B_1$ (by Lemma 2); and hence also $\nu_0 \mathbin{\widehat{\lhd}} \nu_1 \in \nu_0 \mathbin{\widehat{\#}} \nu_1$. Similarly, if $\mu_0 \smile_{*} \mu_1$ then also $\mu_0 \smile_{\lhd} \mu_1$, because the strong disjointness requirements of the former imply the definedness conditions of the latter. And generally $\nu_0 \mathbin{\widehat{*}} \nu_1 \subseteq \nu_0 \mathbin{\widehat{\#}} \nu_1$ because, when $\mu_0 \smile_{*} \mu_1$, $h_0 \uplus h_1 = h_0 \backslash\!\!\backslash h_1$ (by Lemma 1) and $B_0 \backslash\!\!\backslash B_1 = B_0 \uplus B_1$ (by Lemma 2). Consequently, all the equations and inclusions in the examples of the previous two sections, for temporal and spatial separation, hold for spatiotemporal separation as well.

As an example of the weakness of spatiotemporal separation, consider $\nu_0 =$

$(\emptyset, \mathcal{E}[0 \leftarrow [(1,2)]], \emptyset, \emptyset)$ and $\nu_1 = (3 \mapsto 4, \mathcal{E}[0 \leftarrow [(1,3)]], \emptyset, \{0\})$. Observe that $\nu_0 \mathbin{\widehat{\lhd}} \nu_1$ is undefined because the buffer 0 is complete in ν_1 (i.e., $0 \in \Gamma_1$), but nonempty in ν_0. And $\nu_0 \mathbin{\widehat{*}} \nu_1$ is empty as well because of failed disjointness conditions (e.g., $1 \in \mathrm{dom}(B_0(0)) \cap \mathrm{dom}(B_1(0))$. But, in contrast, $\nu_0 \mathbin{\widehat{\#}} \nu_1$ is nonempty, with:

$$(3 \mapsto 4, \mathcal{E}[0 \leftarrow [(1,2),(1,3)]], \emptyset, \{0\}) \in \nu_0 \mathbin{\widehat{\#}} \nu_1.$$

Note that the result of a load of either location 1 or 3 on processor 0 in memory system ν_1 above is preserved in the expanded memory systems of $\nu_0 \mathbin{\widehat{\#}} \nu_1$. This is important because, as mentioned above, we wish for commands to be local w.r.t. spatiotemporal separation so that conjunctions derived from it may have sound corresponding frame rules.

As further explanation of the definition of spatiotemporal separation, consider a load of location ℓ on processor i in an arbitrary memory system ν_1 for which ℓ is defined. Let us consider the manners in which ν_1 can be extended while preserving the resultant value of the load.

1. We *may* augment ν_1 with additional buffered writes to locations distinct from ℓ regardless of their ordering with respect to writes to ℓ already present. The resultant value of the load command is not affected by buffered writes to locations not being loaded.

2. We *may* augment ν_1 with additional buffered writes to address ℓ on buffer i if those writes occur before the most recent writes to ℓ on i. The load command only returns the most recent buffered write, so additional earlier buffered writes will not affect the result. But we *may not* augment ν_1 with additional buffered writes to address ℓ on i that are more recent than those already present, for these additional writes cat affect the outcome of the load.[4]

[4]Of course, if additional later buffered writes have the same value of as the most recent buffered

3. We *may* augment ν_1 with additional committed writes to locations distinct from ℓ regardless of their ordering with respect to writes to ℓ already present. The resultant value of the load command is not affected by committed writes to locations not being loaded.

4. We *may* augment ν_1 with additional committed writes to address ℓ if those writes again precede previously committed writes to ℓ in ν_1. (Because committed writes implicitly precede all buffered writes, this is consistent with the previous scenario in which the ith buffer is safely augmented with earlier writes.) But we *may not* add additional committed writes that succeed previously committed writes, for those could be observed by the load.[5]

5. We *may* augment ν_1 with additional writes to locations distinct from ℓ on other buffers j, with $j \neq i$, as well, regardless of their ordering with respect to existing ℓ writes on buffer j. Although those writes may commit before or after the ℓ-writes being loaded by i, they do not affect the result.

6. Finally, consider writes to address ℓ on buffer j with $j \neq i$. In general, this may adversely affect the load on i because we are necessarily unable to predict the manner in which these writes buffered by j will commit to memory. For example, it is possible that they will commit after buffered writes on i have committed but before the load has completed, thus affecting the result of the load. So it would seem that such writes must be disallowed.

   But there is an important situation in which it is safe to augment ν_1 in this way: namely, when buffer j is blocked. Then, writes buffered by j will not be committed to memory, and so there is no risk that they will be made visible

write to ℓ then they may not affect the value of the load, but neither is that obvious nor does it seem worthwhile to consider seriously such a daring corner case in the definition of logical connective.

[5]In case there are buffered writes on i, it would also be safe to add committed writes to ℓ which succeed existing committed writes. But factoring in this behavior would make a compositional, associative definition of the logical connectives difficult.

to the load on i. Hence, we *may* augment other buffers with writes to ℓ when j is blocked.

We can now check the proposed definition of $\nu_0 \mathbin{\widehat{\#}} \nu_1$ for consistency with the scenarios above:

1. Buffered writes to locations other than ℓ on buffer i are ordered arbitrarily w.r.t. existing writes to ℓ by definition of buffer overriding, $B_0(i) \mathbin{\backslash\!\!\backslash} B_1(i)$, consistent with the corresponding scenario above.

2. Buffered ℓ-writes on buffer i necessarily precede any ℓ-writes already present on buffer i by definition of buffer overriding, $B_0(i) \mathbin{\backslash\!\!\backslash} B_1(i)$, consistent with the corresponding scenario above.

3. Committed writes to locations other than ℓ are ordered arbitrarily w.r.t. ℓ writes already present in the heap by definition of heap overriding $h_0 \mathbin{\backslash\!\!\backslash} h_1$, consistent with the corresponding scenario above.

4. Committed ℓ-writes necessarily precede any ℓ-writes already present in the heap by definition of heap overriding, $h_0 \mathbin{\backslash\!\!\backslash} h_1$, consistent with the corresponding scenario above.

5. Buffered writes to locations other than ℓ on another buffer j, with $j \neq i$, are ordered arbitrarily w.r.t. existing writes to ℓ by definition of buffer overriding, $B_0(j) \mathbin{\backslash\!\!\backslash} B_1(j)$, consistent with the corresponding scenario above. There are no ordering constraints among buffered writes on different write buffers by definition of overriding write buffer arrays, $B_0 \mathbin{\backslash\!\!\backslash} B_1$.

6. Finally, in the case of buffered ℓ writes on another buffer j, with $j \neq i$, the definition is consistent with the corresponding scenario above by way of the definedness conditions, which ensure that buffer j is blocked when ℓ writes are already present in buffer i or in the heap.

As with spatial separation, we overload for convenience the symbol $\widehat{\#}$ to indicate the pointwise lifting of this function to sets of memory systems:

$$S_1 \mathbin{\widehat{\#}} S_2 =_{df} \cup \left\{ \nu_1 \mathbin{\widehat{\#}} \nu_2 \mid \nu_1 \in S_1 \wedge \nu_2 \in S_2 \wedge \nu_1 \smile_{\#} \nu_2 \right\}.$$

We use these functions interchangeably when the intended meaning is clear from context, e.g.:

$$\nu_1 \mathbin{\widehat{\#}} (\nu_2 \mathbin{\widehat{\#}} \nu_3) = \cup \left\{ \nu_1 \mathbin{\widehat{\#}} \nu_{23} \mid \nu_{23} \in \nu_2 \mathbin{\widehat{\#}} \nu_3 \right\}.$$

In the following, let $\nu_u =_{df} (\emptyset, \mathcal{E}, \emptyset, \emptyset)$ be an empty generalized memory system as before. The following lemma asserts some algebraic properties of spatiotemporal separation.

Proposition 7. *For generalized multiprocessor memory systems ν_0, ν_1, ν_2:*

- $\nu_u \mathbin{\widehat{\#}} \nu_0 = \nu_0 \mathbin{\widehat{\#}} \nu_u = \{\nu_0\}$
- $\nu_0 \mathbin{\widehat{\#}} (\nu_1 \mathbin{\widehat{\#}} \nu_2) = (\nu_0 \mathbin{\widehat{\#}} \nu_1) \mathbin{\widehat{\#}} \nu_2$

4.4 Concurrent Assertions

Assertions are used to describe certain sets of multiprocessor models—and hence multiprocessor states and memory systems—and are used to write the pre- and post-conditions of commands in the specification logic. The core language is defined by the following grammar:

$$\begin{array}{lcl} \textbf{Assert}\ P & ::= & b \mid (P \vee P') \mid (P \wedge P') \mid (\exists x : P) \mid (\forall x : P) \mid \\ & & \textbf{emp} \mid \textbf{bar}_e \mid \textbf{lock}_e \mid e \rightsquigarrow_{e'} e'' \mid (P * P') \mid (P \mathrel{-\!\!*} P') \mid \\ & & (P \lhd P') \mid (P \mathrel{-\!\!\lhd} P') \mid (P \mathrel{\lhd\!\!-} P') \end{array}$$

The informal meaning of these assertions are as follows. The lifting of a boolean expression to an atomic formula, disjunction, conjunction and quantification have the usual meaning. **emp** describes states with an empty heap and all write buffers empty. $\mathbf{bar}_e$ describes states in which just the eth buffer is empty. $\mathbf{lock}_e$ asserts that processor e holds the lock. $e \rightsquigarrow_{e'} e''$ describes a single write to location e with value e'', either buffered on processor e' or flushed to memory. The temporal separating conjunction ($P \lhd P'$) describes per-write buffer concatenation of writes. The spatial separating conjunction ($P * P'$) requires disjointness of the locations described by P and P', and interleaves the described writes on each write buffer instead of concatenating them. The spatial separating implication ($P \mathrel{-\!\!*} P'$) and the left and right temporal separating implications ($P \mathrel{-\!\!\lhd} P'$) and ($P \mathrel{\lhd\!\!-} P'$) are analogous to the separating implications defined earlier in Section 3.5.3.

The set of free variables of an assertion, written $\mathrm{fv}(P)$, is defined as usual. For example, $\mathrm{fv}(\exists z : x \rightsquigarrow_z y \lhd \mathbf{bar}_z) = (\mathrm{fv}(x \rightsquigarrow_z y) \cup \mathrm{fv}(\mathbf{bar}_z)) \setminus \{z\} = (\{x, y, z\} \cup \{z\}) \setminus \{z\} = \{x, y\}$.

4.4.1 Concurrent Satisfaction

The meaning of assertions is given by a satisfaction relation $\mathcal{M} \models P$, relating *models* $\mathcal{M}$ to assertions P. A model is a pair (s, ν) consisting of a stack s and a generalized multiprocessor memory system ν. The satisfaction relation is defined by recursion on the structure of P below in Figure 4.4. For concision, the text of the definition makes use of the following auxiliary set definitions:

$$\begin{aligned}
\mathit{emp} &=_{\mathit{df}} \{(h, B, K, \Gamma) \mid h = \emptyset \wedge B = \mathcal{E} \wedge K = \emptyset\} \\
\mathit{bar}(i) &=_{\mathit{df}} \{(h, B, K, \Gamma) \in \mathit{emp} \mid i \notin \mathbb{P} \vee i \in \Gamma\} \\
\mathit{lock}(i) &=_{\mathit{df}} \{(h, B, K, \Gamma) \mid h = \emptyset \wedge B = \mathcal{E} \wedge i \in \mathbb{P} \wedge K = \mathbb{P} \setminus \{i\}\} \\
\mathit{pending}(i, \ell, v) &=_{\mathit{df}} \{(h, B, K, \Gamma) \mid h = \emptyset \wedge B = \mathcal{E}[i \leftarrow [(\ell, v)]] \wedge K \subseteq \{i\}\} \\
\mathit{flushed}(i, \ell, v) &=_{\mathit{df}} \{(h, B, K, \Gamma) \mid h = \ell \mapsto v \wedge B = \mathcal{E} \wedge K = \emptyset \wedge i \in \Gamma\} \\
\mathit{leads\text{-}to}(i, l, v) &=_{\mathit{df}} \{(h, B, K, \Gamma) \in \mathit{pending}(i, l, v) \cup \mathit{flushed}(i, l, v) \mid i \in \mathbb{P}\}
\end{aligned}$$

$$\begin{array}{llll}
s, \nu & \models & b & \equiv_{\mathit{df}} \quad \hat{s}(b) = 1 \\
s, \nu & \models & P \vee Q & \equiv_{\mathit{df}} \quad s, \nu \models P \vee s, \nu \models Q \\
s, \nu & \models & P \wedge Q & \equiv_{\mathit{df}} \quad s, \nu \models P \wedge s, \nu \models Q \\
s, \nu & \models & \exists x : P & \equiv_{\mathit{df}} \quad \exists v \in \mathbb{V} : s[x \leftarrow v], \nu \models P \\
s, \nu & \models & \forall x : P & \equiv_{\mathit{df}} \quad \forall v \in \mathbb{V} : s[x \leftarrow v], \nu \models P \\
s, \nu & \models & \mathbf{emp} & \equiv_{\mathit{df}} \quad \nu \in \mathit{emp} \\
s, \nu & \models & \mathbf{bar}_e & \equiv_{\mathit{df}} \quad \nu \in \mathit{bar}(\hat{s}(e)) \\
s, \nu & \models & \mathbf{lock}_e & \equiv_{\mathit{df}} \quad \nu \in \mathit{lock}(\hat{s}(e)) \\
s, \nu & \models & e \leadsto_{e'} e'' & \equiv_{\mathit{df}} \quad \nu \in \mathit{leads\text{-}to}(\hat{s}(e'), \hat{s}(e), \hat{s}(e'')) \\
s, \nu & \models & P * P' & \equiv_{\mathit{df}} \quad \exists \nu_0, \nu_1 : \nu \in \nu_0 \mathbin{\widehat{*}} \nu_1 \wedge \\
 & & & \qquad s, \nu_0 \models P \wedge s, \nu_1 \models P' \\
s, \nu & \models & P \lhd P' & \equiv_{\mathit{df}} \quad \exists \nu_0, \nu_1 : \nu = \nu_0 \mathbin{\widehat{\lhd}} \nu_1 \wedge \\
 & & & \qquad s, \nu_0 \models P \wedge s, \nu_1 \models P' \\
s, \nu & \models & P \mathrel{-\!\!*} P' & \equiv_{\mathit{df}} \quad \forall \nu_0, \nu_1 : s, \nu_0 \models P \wedge \nu_1 \in \nu_0 \mathbin{\widehat{*}} \nu \Rightarrow \\
 & & & \qquad s, \nu_1 \models P' \\
s, \nu & \models & P \mathrel{-\!\!\lhd} P' & \equiv_{\mathit{df}} \quad \forall \nu_0, \nu_1 : s, \nu_0 \models P \wedge \nu_1 = \nu_0 \mathbin{\widehat{\lhd}} \nu \Rightarrow \\
 & & & \qquad s, \nu_1 \models P' \\
s, \nu & \models & P \mathrel{\lhd\!\!-} P' & \equiv_{\mathit{df}} \quad \forall \nu_0, \nu_1 : s, \nu_1 \models P' \wedge \nu_0 = \nu \mathbin{\widehat{\lhd}} \nu_1 \Rightarrow \\
 & & & \qquad s, \nu_0 \models P
\end{array}$$

Figure 4.4: Concurrent satisfaction relation

The classical atomic formulas and logical operations are defined as usual. The **emp** assertion describes generalized memory systems with empty heaps, empty write buffers, and with no processors blocked. The $\mathbf{bar}_i$ assertion additionally requires that i be buffer-complete. The $\mathbf{lock}_i$ assertion is like $\mathbf{emp}$, but requires that all processors except for i be blocked. The leads-to assertion $e \rightsquigarrow_i f$ is modeled by two classes of generalized memory systems: those that are *pending*, with an empty heap, a single buffered write, and in which at most i is blocked; and those that are *flushed*, with a single-point heap, all buffers empty, and in which no processors are blocked. The separating conjunctions and implications are defined with the semantic functions described in Section 4.3.

Remark. The choice of blocked sets in the leads-to equation deserves additional explanation. In the case of models with a pending write on processor i, the blocked set may or may not contain i—the pending write may not have flushed to memory because the buffer on which it is enqueued is blocked; or, if the buffer is not blocked, it may simply have not yet flushed. In the case of models in which the write has flushed, i must not be blocked because, indeed, the write has flushed, which can only happen if the buffer is unblocked.

It is also instructive to consider the models of the leads-to assertion in separating-conjunction with the lock assertions: $e \rightsquigarrow_i e' * \mathbf{lock}_j$. The only blocked set of the models of $\mathbf{lock}_j$ is $\mathbb{P} \setminus \{j\}$, which is maximal w.r.t. set inclusion. (Operationally, no lock state corresponds to every processor being blocked.) Consequently, because the separating conjunction requires that the blocked sets of the constituent models be disjoint, the only models of $e \rightsquigarrow_i e'$ that are compatible with a model of $\mathbf{lock}_j$ are those with an empty blocked set; i.e., those in which i is not blocked. An informal interpretation of an empty blocked set is thus that of a model in which the lock status is wholly unconstrained.

Incompatibility of models of the leads-to assertion in which i is blocked is intuitive in case $i = j$: a blocked set that consists solely of i ought not to be compatible with one that consists of all processors except i. But in case $i \neq j$, the incompatibility is less obvious. Why must a model of a leads-to assertion which asserts that i is blocked be incompatible with a model of a lock assertion which asserts that many processors including i are blocked? The reason is that the lock status may be dynamically changed by the lock-manipulation primitives, which if compatibility were relaxed could lead to inconsistency. That is, even if a model of $e \rightsquigarrow_i e'$ in which i is necessarily blocked were allowed to be compatible with models of $\mathbf{lock}_i$, it surely ought not be compatible with models of $\mathbf{lock}_j$, with $j \neq i$; but the lock-manipulations primitives can effect just this change. The separating conjunctions are designed to isolate parts of the state from such changes, and such a relaxed notion of compatibility would lead to unsoundness of the frame rules in the context of lock manipulation primitives.

Similarly, even if the semantics were relaxed so that $\mathbf{lock}_i * \mathbf{lock}_i \equiv \mathbf{lock}_i$, we must still ensure that $\mathbf{lock}_i * \mathbf{lock}_j$ is inconsistent when $i \neq j$. But then we could derive an invalid specification as follows:

$$\cfrac{\cfrac{\cfrac{\vdots}{J \vdash \{\mathbf{lock}_0\}\ \mathsf{unlock}_0 \parallel \mathsf{lock}_1\ \{\mathbf{lock}_1\}} \quad \cdots}{J \vdash \{\mathbf{lock}_0 * \mathbf{lock}_0\}\ \mathsf{unlock}_0 \parallel \mathsf{lock}_1\ \{\mathbf{lock}_0 * \mathbf{lock}_1\}}\ \text{FRAME-SP}}{J \vdash \{\mathbf{lock}_0\}\ \mathsf{unlock}_0 \parallel \mathsf{lock}_1\ \{\mathbf{false}\}}\ \text{CONS}$$

$\square$

We write $[\![P]\!]$ for the set of models that satisfy P,

$$[\![P]\!] =_{df} \{\mathcal{M} \mid \mathcal{M} \models P\},$$

and also $P \models P'$ and $P \equiv P'$ for semantic entailment and equivalence, respectively:

$$P \models P' \equiv_{df} [\![P]\!] \subseteq [\![P']\!]$$
$$P \equiv P' \equiv_{df} [\![P]\!] = [\![P']\!].$$

4.4.2 Additional Concurrent Assertions

As before, we shall define the strong temporal separating conjunction as the additive conjunction of spatial and temporal separating conjunctions:

$$P \blacktriangleleft P' =_{df} (P * P') \wedge (P \lhd P').$$

Again, the spatial separating conjunction $P * Q$ is commutative, so there is no need to define its converse operation, but the temporal separating conjunction is not. Hence, we define $P \rhd Q$ as shorthand for $Q \lhd P$, and similarly for $P \blacktriangleright Q$:

$$P \rhd Q =_{df} Q \lhd P$$
$$P \blacktriangleright Q =_{df} Q \blacktriangleleft P.$$

In the sequential assertion logic we defined the points-to assertion $e \mapsto e'$ as an abbreviation of a leads-to assertion followed temporally by a barrier assertion. In the concurrent assertion logic, however, the situation is not as simple. The leads-to assertions are here annotated with expressions indicating the identifier of the processor on which they are buffered, and similarly for barrier assertions. We might try to define a points-to assertion as a leads-to assertion on a certain processor temporally conjoined with a barrier assertion on the same processor:

$$e \mapsto e' =_{df} \exists x : e \rightsquigarrow_x e' \lhd \mathbf{bar}_x$$

The intuition behind this abbreviation is that the result of flushing on one processor should be indistinguishable from the result of flushing the same write on a different processor, and hence the choice of a particular process from which the write originated is unimportant. But, in the model described in this document, this is only true of models for which all buffers are complete; and in general models of assertions need not have all buffers complete. As a consequence of the abbreviation above, among the models of $1 \rightsquigarrow_0 1 \lhd 2 \mapsto 2$ are included some in which the write to location 1 is buffered on processor 0, whereas there should arguably be none.

The correct definition instead temporally conjoins a leads-to formula with another that describes the flushing of *all* buffers:

$$e \mapsto e =_{df} (\exists x : e \rightsquigarrow_x e') \lhd (\forall x : \mathbf{bar}_x).$$

The left-hand conjunct describes a possibly buffered write on some processor, while the right-hand conjunction describes the result of flushing every processor. Hence, the models of $e \mapsto e'$ describe only a flushed write with all processors buffer-complete. Note that if x does not denote a processor identifier, then $e \rightsquigarrow_x e'$ is not satisfied by any model, whereas in the same circumstance $\mathbf{bar}_x$ is instead satisfied by the same models as **emp**.

4.4.3 Concurrent Algebra

A few additional semantic equivalences and entailments are shown in Figures 4.5 and 4.6, respectively. If a formula contains instances of $\bullet$ or $\circ$, then that is shorthand for the same formula in which the $\bullet$ has been consistently replaced by any of the separating conjunctions.

As in the sequential assertion logic, each of the separating conjunctions is associative and has **emp** as a unit. And the spatial separating conjunction is again commutative. The separating conjunctions are additive w.r.t. the barrier assertion $\mathbf{bar}_e$, and $\mathbf{bar}_e$ also distributes fully through the separating conjunctions. Again, the strong and weak temporal separating conjunctions of barrier assertions are equivalent. The separating conjunctions are multiplicative w.r.t. the lock assertion $\mathbf{lock}_e$ and also symmetric. $\mathbf{bar}_e$ also commutes with $\mathbf{lock}_e$, even for the sequential conjunctions. Note that the result of flushing successive writes to the same location is

$$\begin{aligned}
P \bullet \mathbf{emp} &\equiv P \\
\mathbf{emp} \bullet P &\equiv P \\
(P \bullet P') \bullet P'' &\equiv P \bullet (P' \bullet P'') \\
P * P' &\equiv P' * P \\
P \bullet \mathbf{lock}_e &\equiv \mathbf{lock}_e \bullet P \\
\mathbf{lock}_e \bullet \mathbf{lock}_f &\equiv \mathbf{false} \\
\mathbf{bar}_e \bullet \mathbf{bar}_e &\equiv \mathbf{bar}_e \\
(P \bullet P') \lhd \mathbf{bar}_e &\equiv (P \lhd \mathbf{bar}_e) \bullet (P' \lhd \mathbf{bar}_e) \\
P \blacktriangleleft \mathbf{bar}_e &\equiv P \lhd \mathbf{bar}_e \\
\mathbf{lock}_e \lhd \mathbf{bar}_e &\equiv \mathbf{lock}_e * \mathbf{bar}_e \\
e \rightsquigarrow_{e'} e'' * e \rightsquigarrow_{f'} f'' &\equiv \mathbf{false} \\
e \rightsquigarrow_{e'} e'' \lhd e \rightsquigarrow_{e'} f \lhd \mathbf{bar}_{e'} &\equiv e \rightsquigarrow_{e'} f \lhd \mathbf{bar}_{e'}
\end{aligned}$$

Figure 4.5: Concurrent semantic equivalences

equivalent to flushing just the most recent write.

$$\begin{aligned}
\mathbf{bar}_e &\models \mathbf{emp} \\
P \bullet P' &\models P'' \bullet P' && \text{if } P \models P'' \\
P \bullet P' &\models P \bullet P'' && \text{if } P' \models P'' \\
e \mapsto f &\models e \rightsquigarrow_{e'} f \\
P \blacktriangleleft P' &\models P * P' \\
P \blacktriangleleft P' &\models P \lhd P' \\
P \bullet (P' \circ P'') &\models (P \bullet P') \circ P'' && \text{for } P \bullet P' \models P \circ P' \\
(P \circ P') \bullet P'' &\models P \circ (P' \bullet P'') && \text{for } P \bullet P' \models P \circ P' \\
(P * P') \blacktriangleleft (P'' * P''') &\models (P \blacktriangleleft P'') * (P' \blacktriangleleft P''')
\end{aligned}$$

Figure 4.6: Concurrent semantic entailments

The first three entailments in Figure 4.6—that $\mathbf{bar}_e$ strengthens **emp** and monotonicity of the separating conjunctions—follow directly from the definition of the satisfaction relation. Points-to strengthens leads-to by its abbreviation expansion and monotonicity of the weak temporal separating conjunction, along with the fact that $(\forall x : \mathbf{bar}_x) \models \mathbf{emp}$. The next two entailments follow from their respective abbreviation expansions and by monotonicity of $\wedge$. The three separating conjunctions naturally form a sort of lattice, and they satisfy the small exchange laws. The full exchange law only holds for the spatial and strong temporal conjunctions. The law does not hold for the other separating conjunctions because, e.g., it implies commutativity of the principal connective in the consequent, and the other conjunctions are not commutative.

4.4.4 Flushing Closure

Assertions can thus be thought of as syntactic constructs that denote sets of multiprocessor models, and hence sets of generalized multiprocessor machine states. These sets of models have a special structure: namely, they are down-closed w.r.t. a partial order, which encompasses the flushing partial order on multiprocessor memory systems. This section formalizes this property.

The flushing order on multiprocesor memory systems, described in Section 3.4.4, is extended to a generalized order on generalized multiprocessor memory systems as follows:

Definition 6. For generalized multiprocessor memory systems $\nu_1 = (\mu_1, \Gamma_1)$ and $\nu_2 = (\mu_2, \Gamma_2)$,

$$\begin{aligned}
\nu_2 \underset{\widehat{\tau},i}{\rightarrow} \nu_1 &\equiv_{df} \Gamma_2 \subseteq \Gamma_1 \wedge i \in \Gamma_1 \wedge \mu_2 \underset{\tau,i}{\rightarrow} \mu_1 \\
\nu_2 \underset{\widehat{\tau}}{\rightarrow} \nu_1 &\equiv_{df} \exists i \in \mathbb{P} : \nu_2 \underset{\widehat{\tau},i}{\rightarrow} \nu_1 \\
\nu_1 \leq \nu_2 &\equiv_{df} \nu_2 \underset{\widehat{\tau}}{\overset{*}{\rightarrow}} \nu_1
\end{aligned}$$

Informally, $\nu_1 \leq \nu_2$ if ν_2 can flush its buffer-complete processors to arrive at ν_1. The fact that this order on generalized multiprocessor memory systems encompasses the flushing order on multiprocessor memory systems is formalized by the following lemma.

Proposition 8. *If $\mu_0 \preceq \mu_1$ then for all Γ_1 there exists Γ_0 such that $\Gamma_0 \supseteq \Gamma_1$ and $(\mu_0, \Gamma_0) \leq (\mu_1, \Gamma_1)$.*

Proof. If $\mu_0 \preceq \mu_1$, then there exists some sequence of memory systems such that $\mu'_0 \underset{\tau}{\rightarrow} \cdots \underset{\tau}{\rightarrow} \mu'_n$, with $\mu'_0 = \mu_1$ and $\mu'_n = \mu_0$. For each step in the above sequence, there exists a processor identifier i such that $\mu'_m \underset{\tau,i}{\rightarrow} \mu'_{m+1}$. Let Γ' be this set of these identifiers, and $\Gamma_0 = \Gamma_1 \cup \Gamma'$. □

As in the uniprocessor semantics, a central claim is that the denotation $[\![P]\!]$ of each assertion P is closed w.r.t. the generalized flushing order.

Proposition 9 (Flushing Closure)**.** *If $s, \nu \models P$ and $\nu' \leq \nu$ then $s, \nu' \models P$.*

Sketch. By induction on the shape of P. For the separating conjunctions, see, e.g., Lemma 7 in Section A.1. □

An effect of this is that *assertions are oblivious to the nondeterministic flushing of buffered writes to memory.*

Corollary 2. *If $s, \mu, \Gamma \models P$ and $\mu' \preceq \mu$ then there exists Γ' such that $\Gamma' \supseteq \Gamma$ and $s, \mu', \Gamma' \models P$.*

Sketch. By Propositions 9 and 8. □

Intuitively, assertions may be thought to describe only the "initial" states without concern for the nondeterministic flushing of writes that may have taken place, though the semantics also encompasses all states reachable as a result these steps. This is an important feature of the assertion language because flushing buffered writes—which cannot be controlled—never changes the satisfiability of an assertion. This is also extremely important for the specification logic, as is described in the next section.

Remark. Earlier versions of the semantic functions that model the separating conjunctions incorporated directly the notion of flushing closure. In order to ensure that an assertion $P \bullet Q$ denoted a closed sets of states, the corresponding semantic function $\widetilde{\bullet}$ was defined so that $\mu \,\widetilde{\bullet}\, \mu'$ was itself a closed set. For example, the spatial conjunction of two states with buffered writes included not just the result of each possible interleaving of the buffers, but also each possible interleaving of the buffers after arbitrary flushing. Furthermore, the definition of the semantic function for temporal separation was relaxed to be total instead of partial. In particular, the result of temporally conjoining an earlier, left-side state that contained buffered writes to a later, right-side buffer-complete state (e.g., a model of a **bar** assertion)—a composition that would be undefined according to the current definition—was equivalent to first completely flushing the left-side state and only then temporally combining the states as as in the current definition. That is, the definition of the semantic function included an explicit application of a function $\mathit{flush}(h, b)$, which calculates the result of flushing a buffer b into a heap h.

The relaxations described above made closure arguments straightforward, but yielded function definitions that were otherwise difficult to reason about. The semantic functions described in Section 4.3 are simpler and stronger than the earlier definitions sketched here, but the limited scope and stronger definedness conditions of the current definitions make them easier to reason about. And the lemmas and propositions described earlier in this section assert that the simpler definitions are indeed sufficient for flushing closure.

4.5 Concurrent Specifications

The language of specifications is given by the following schema:

$$J \vdash \{P\}\ c\ \{Q\}\,,$$

where c is a command and J, P, Q are assertions, referred to as the *invariant*, *pre-condition* and *post-condition*, respectively.

4.5.1 Concurrent Proof Theory

The axioms of the logic are given in Figure 4.7.

$$J \vdash \{P\}\ \mathsf{skip}_i\ \{P\} \qquad (\textsc{skip})$$

$$J \vdash \{!b \vee P\}\ \mathsf{assume}(b)_i\ \{P\} \qquad (\textsc{assume})$$

$$J \vdash \{b \wedge P\}\ \mathsf{assert}(b)_i\ \{P\} \qquad (\textsc{assert})$$

$$J \vdash \{P\,[e/x]\}\ x := e_i\ \{P\} \qquad (\textsc{assign})$$

$$J \vdash \{e \rightsquigarrow_i e' \blacktriangleleft P\}\ x := [e]_i\ \{(e \rightsquigarrow_i e' \blacktriangleleft P) \wedge x = e'\} \qquad (\textsc{load})$$

$$J \vdash \{e \rightsquigarrow_i e'' \blacktriangleleft P\}\ [e] := e'_i\ \{(e \rightsquigarrow_i e'' \blacktriangleleft P) \lhd e \rightsquigarrow_i e'\} \qquad (\textsc{store})$$

$$J \vdash \{\mathbf{emp}\}\ \mathsf{fence}_i\ \{\mathbf{bar}_i\} \qquad (\textsc{fence})$$

$$J \vdash \{\mathbf{emp}\}\ \mathsf{lock}_i\ \{\mathbf{lock}_i\} \qquad (\textsc{lock})$$

$$J \vdash \{\mathbf{lock}_i\}\ \mathsf{unlock}_i\ \{\mathbf{emp}\} \qquad (\textsc{unlock})$$

Figure 4.7: Concurrent axioms

The axioms for skip, $\mathsf{assume}(b)$, $\mathsf{assert}(b)$ and $x := e$ are as in Hoare logic, identical to the corresponding axioms of the sequential program logic except for the presence of the (arbitrary) invariant J. The load and store axioms are also similar to those of the sequential program logic, but the writes described by the pre- and post-condition are annotated with the processor identifier that annotates the command, which indicates the processor on which the writes are buffered. And similarly for the fence command, which introduces a barrier assertion annotated with the appropriate processor identifier. Finally, the lock and unlock commands produce and consume, respectively, a lock assertion annotated by the appropriate processor identifier. The logical and structural inference rules of the logic are given in Figure 4.8. The logical rules require little explanation: besides the addition of the invariant assertion, they are unchanged from the previous chapter. Note again that, although the strong temporal conjunction is *defined* as the additive conjunction of spatial and temporal conjunctions, its frame rule is not derivable from the other rules, which justifies the rule's inclusion. The spatial separating conjunction is commutative, but the others—which have a temporal aspect—are not. As in the sequential program logic, we have left-side frame rules only for these conjunctions.

Unlike the sequential logic, there is no conjunction rule in the concurrent logic. This is because there is a known connection [20] between the soundness of the conjunction rule and a semantic property called *precision*, which we have not yet identified in this model.

The structural rules for sequential composition, nondeterministic choice, loops and concurrency are similar in spirit to those in Concurrent Separation Logic. In particular, the concurrency rule requires syntactic agreement between the two processes on the shared invariant, and the strong interleaving separating conjunction is used to partition the local states. Note that both this conjunction and the concurrent composition command are commutative. The sharing rule allows us to infer that, if a portion of the state is invariant w.r.t. the assertion J, then J may also be assumed to hold initially and upon termination.

Only the invariant rule differs significantly from Concurrent Separation Logic. Here, we require that the lock be held before accessing the shared state; and that, under the assumption that the shared state satisfied the invariant initially, upon completion of the command the shared state must again be shown to satisfy the invariant. In the subsection below, we discuss a number of variations on this important rule.

$$\frac{J \vdash \{P\}\ c\ \{Q\} \quad J \vdash \{P'\}\ c\ \{Q\}}{J \vdash \{P \vee P'\}\ c\ \{Q\}} \qquad (\textsc{disj})$$

$$\frac{J \vdash \{P\}\ c\ \{Q\} \quad x \notin \mathrm{fv}(c, Q)}{J \vdash \{\exists x : P\}\ c\ \{Q\}} \qquad (\textsc{ex})$$

$$\frac{J \vdash \{P\}\ c\ \{Q\} \quad \mathrm{mod}(c) \cap \mathrm{fv}(R) = \emptyset}{J \vdash \{R * P\}\ c\ \{R * Q\}} \qquad (\textsc{frame-sp})$$

$$\frac{J \vdash \{P\}\ c\ \{Q\} \quad \mathrm{mod}(c) \cap \mathrm{fv}(R) = \emptyset}{J \vdash \{R \lhd P\}\ c\ \{R \lhd Q\}} \qquad (\textsc{frame-tm})$$

$$\frac{J \vdash \{P\}\ c\ \{Q\} \quad \mathrm{mod}(c) \cap \mathrm{fv}(R) = \emptyset}{J \vdash \{R \blacktriangleleft P\}\ c\ \{R \blacktriangleleft Q\}} \qquad (\textsc{frame-stm})$$

$$\frac{P \models P' \quad J \vdash \{P'\}\ c\ \{Q'\} \quad Q' \models Q}{J \vdash \{P\}\ c\ \{Q\}} \qquad (\textsc{cons})$$

$$\frac{J \vdash \{P\}\ c\ \{R\} \quad J \vdash \{R\}\ c'\ \{Q\}}{J \vdash \{P\}\ c\,;c'\ \{Q\}} \qquad (\textsc{seq})$$

$$\frac{J \vdash \{P\}\ c\ \{Q\} \quad J \vdash \{P\}\ c'\ \{Q\}}{J \vdash \{P\}\ c+c'\ \{Q\}} \qquad (\textsc{choice})$$

$$\frac{J \vdash \{P\}\ c\ \{P\}}{J \vdash \{P\}\ c^*\ \{P\}} \qquad (\textsc{loop})$$

$$\frac{J \vdash \{P_i\}\ c_i\ \{Q_i\} \quad \mathrm{fv}(P_i, c_i, Q_i) \cap \mathrm{mod}(c_{1-i}) = \emptyset \,\text{for}\ i \in \{0,1\}}{J \vdash \{P_0 * P_1\}\ c_0 \,\|\, c_1\ \{Q_0 * Q_1\}} \qquad (\textsc{conc})$$

$$\frac{J \vdash \{P\}\ c\ \{Q\}}{\mathbf{emp} \vdash \{J * P\}\ c\ \{J * Q\}} \qquad (\textsc{share})$$

$$\frac{\mathbf{emp} \vdash \{J * P * \mathbf{lock}_i\}\ c\ \{J * Q * \mathbf{lock}_i\}}{J \vdash \{P * \mathbf{lock}_i\}\ c\ \{Q * \mathbf{lock}_i\}} \qquad (\textsc{inv})$$

Figure 4.8: Concurrent inference rules

Derived and Alternative Axioms and Inference Rules

Loading flushed writes Observe that the write in the pre-condition of both the load and store axioms need not be buffered: the placeholder assertion P can be instantiated with an appropriate barrier assertion to indicate that the write has been flushed:

$$J \vdash \left\{e \leadsto_i e' \blacktriangleleft (\mathbf{bar}_i \lhd P')\right\}\ x := [e]_i\ \left\{(e \leadsto_i e' \blacktriangleleft (\mathbf{bar}_i \lhd P')) \wedge x = e'\right\}$$

It is interesting to note though that this only describes the result of loading a write in memory that originated from the processor performing the load. We can, however, derive variants of the load and store axioms which allow us to reason about writes flushed by an arbitrary processor in case all buffers are complete—i.e., variants of the load and store axioms that make use of points-to assertions instead of leads-to assertions, as follows:

$$J \vdash \{e \mapsto e' \blacktriangleleft P\}\ x := [e]_i\ \{(e \mapsto e' \blacktriangleleft P) \wedge x = e'\} \qquad \text{(LOAD-P)}$$

$$J \vdash \{e \mapsto e' \blacktriangleleft P\}\ x := [e]_i\ \{(e \mapsto e' \blacktriangleleft P) \blacktriangleleft e \leadsto_i e'\} \qquad \text{(STORE-P)}$$

These axioms can be derived by instantiating P by $(\forall x : \mathbf{bar}_x) \blacktriangleleft P$ and with the rule of consequence, using the following equivalence:

$$e \leadsto_i e' \blacktriangleleft ((\forall x : \mathbf{bar}_x) \blacktriangleleft P) \equiv ((\exists x : e \leadsto_x e') \lhd (\forall x : \mathbf{bar}_x)) \blacktriangleleft P$$

Stronger lock-manipulation axioms The given axioms for the lock and unlock commands are sound but rather weak insofar as they do not reflect the implicit fencing that is coincident with these commands. The axioms, however, can be strengthened to take this into account as follows:

$$J \vdash \{\mathbf{emp}\}\ \mathsf{lock}_i\ \{\mathbf{lock}_i * \mathbf{bar}_i\} \qquad \text{(LOCK)}$$

$$J \vdash \{\mathbf{lock}_i\}\ \mathsf{unlock}_i\ \{\mathbf{bar}_i\} \qquad \text{(UNLOCK)}$$

Using the frame rules, we can also derive "global" axioms for lock and unlock, as well as fence:

$$J \vdash \{P\}\ \mathsf{fence}_i\ \{P \lhd \mathbf{bar}_i\} \qquad \text{(FENCE-G)}$$

$$J \vdash \{P\}\ \mathsf{lock}_i\ \{\mathbf{lock}_i * P \lhd \mathbf{bar}_i\} \qquad \text{(LOCK-G)}$$

$$J \vdash \{P * \mathbf{lock}_i\}\ \mathsf{unlock}_i\ \{P \lhd \mathbf{bar}_i\} \qquad \text{(UNLOCK-G)}$$

We can also derive "backward" variations of the global rules, using the temporal and spatial separating implications:

$$J \vdash \{P \lhd\!\!- \mathbf{bar}_i\}\ \mathsf{lock}_i\ \{P\} \qquad \text{(FENCE-B)}$$

$$J \vdash \{(P \lhd\!\!- \mathbf{bar}_i) -\!\!* \mathbf{lock}_i\}\ \mathsf{lock}_i\ \{P\} \qquad \text{(LOCK-B)}$$

$$J \vdash \{(P \lhd\!\!- \mathbf{bar}_i) * \mathbf{lock}_i\}\ \mathsf{unlock}_i\ \{P\} \qquad \text{(UNLOCK-B)}$$

Locked and atomic commands Using the invariant rule and the axioms for lock and unlock, we can derive the following rule for reasoning about "locked" commands, such as those implemented on x86 like atomic increment or compare-and-swap:

$$\frac{\mathbf{emp} \vdash \{(J * P * \mathbf{lock}_i) \lhd \mathbf{bar}_i\}\ c\ \{J * Q * \mathbf{lock}_i\}}{J \vdash \{P\}\ \mathsf{lock}_i\,; c\,; \mathsf{unlock}_i\ \{Q \lhd \mathbf{bar}_i\}} \qquad \text{(LOCKED)}$$

A shortcoming of this derived rule specifically, and the existing locking axioms and invariant inference rules generally, is the shared invariant must be general enough to describe buffered writes that may never be observed because of the fencing implicit with the lock commands. For example, the following specification is true but not provable with the current rules:

$$x \mapsto 1 \vdash \{\mathbf{emp}\}\ \mathsf{lock}_0\,; [x] := 1_0\,; \mathsf{unlock}_0\ \{\mathbf{emp}\}$$

To prove this with the derived LOCKED rule above, we would have to prove the following specification:

$$\mathbf{emp} \vdash \{x \mapsto 1 * \mathbf{lock}_0\}\ [x] := 1_0\ \{x \mapsto 1 * \mathbf{lock}_0\}\,,$$

but this is false because the post-condition of the store command yields a buffered write that has not necessarily flushed, yet the post-condition requires that the buffer be empty.

We remedy this by providing an alternative, stronger invariant axiom that uses the expansion of the invariant, and relies upon commands being well-locked:

$$\frac{\mathbf{emp} \vdash \{(J * P * \mathbf{lock}_i) \lhd \mathbf{bar}_i\}\ c\ \{(J * Q * \mathbf{lock}_i) \mathrel{\lhd\!\!-} \mathbf{bar}_i\}}{J \vdash \{P\}\ \mathsf{lock}_i\,;c\,;\mathsf{unlock}_i\ \{Q\}}\ (\textsc{atomic})$$

(We call this the ATOMIC rule to distinguish it from the LOCKED derived rule, though they apply to the same commands.) This inference rule allows for a stronger invariant, such as in the previous example.

Daring rules for accessing shared state We may also consider additional "daring" inference rules, the soundness of which may well be quite difficult to demonstrate.

$$\frac{\mathbf{emp} \vdash \{J * P\}\ x := [e]_i\ \{J * Q\}}{J \vdash \{P\}\ x := [e]_i\ \{Q\}}\qquad (\textsc{daring-load})$$

$$\frac{\mathbf{emp} \vdash \{J * P\}\ [e] := e'_i\ \{J * Q\}}{J \vdash \{P\}\ [e] := e'_i\ \{Q\}}\qquad (\textsc{daring-store})$$

These rules differ from the invariant rule because they allow reasoning about the behavior of individual load and store instructions in which the value of the lock is unspecified. Intuitively, the shared load rule might be shown to be true because a load may only proceed on a live processor, and so will never access shared state while it is being modified by another process, which holds the lock. On the other hand, the shared store axiom might be shown to be true because although a store may take place while another process is modifying the shared state—and hence while the shared state does not satisfy the stated invariant J—the buffered write will not commit until the other process has released the lock and repaired the shared state, restoring the invariant.

A problem with the daring store rule is that it requires the invariant J to encompass the newly added buffered write; but heap-only invariants are easier to describe and more general. A possible relaxation of the daring store rule is thus as follows:

$$\frac{\mathbf{emp} \vdash \{J * P\}\ [e] := e'_i\ \{(J \lhd\!\!- \mathbf{bar}_i) * Q\}}{J \vdash \{P\}\ [e] := e'_i\ \{Q\}}\ (\textsc{daring-store-2})$$

This relaxation would allow a daring store if the new write *when flushed* would maintain the invariant. Unfortunately, this rule is almost certainly not sound w.r.t. the semantics of specification described in the next section, because it is simply not the case that the state necessarily satisfies the invariant after the store; only that it is somehow observationally equivalent, in that a load of the shared region shall be compatible with the result of a load that exactly satisfies the stated invariant. Explorations of the soundness of these rules is thus considered future work.

The daring rules may be needed to reason about, e.g., x86 spinlock implementations. The spinlock is typically acquired using a compare-and-swap instruction, which in this language is simply a locked if-the-else command. The invariant rule and lock axioms thus should be sufficient for demonstrating correctness of spinlock acquisition. But the spinlock is released by writing to a shared memory address without first acquiring the global lock or fencing. This obviates the invariant rule, but not the shared write rule, and so there is yet hope.

4.5.2 Semantics of Concurrent Specifications

A specification asserts the partial correctness of a command. Its informal meaning is roughly analogous to that of Concurrent Separation Logic: if c is evaluated in a state that satisfies $J * P$, then: *1)* it does not abort, *2)* it maintains the invariant J during execution, and *3)* if it evaluates fully, it terminates in a state that satisfies $J * Q$.

Following Vafeiadis [54], the formal semantics of specifications is given by a family of predicates, $\mathit{safe}_n(c, s, \mu, J, Q)$, parametrized by $n \in \mathbb{N}$, that relate a command c, state (s, μ), invariant assertion J and post-condition Q according to the informal explanation above. Once these predicates are defined, we define truth of specifications as follows:

$$J \models \{P\}\ c\ \{Q\} \equiv_{\mathit{df}} \forall s : \forall \mu : (s, \mu, \mathbb{P}) \models P \Rightarrow \forall n \in \mathbb{N} : \mathit{safe}_n(c, s, \mu, J, Q).$$

Observe that only states (s, μ) for which *buffer-complete models* $(s, \mu, \mathbb{P})$ of the pre-condition are relevant to the meaning of specifications.

In the sequel, let $\mathit{locked}(\mu)$ indicate that some processor holds the lock in memory system μ:

$$\mathit{locked}(h, B, K) \equiv_{\mathit{df}} \exists i \in \mathbb{P} : k = \mathbb{P} \setminus \{i\}\,.$$

We now give a formal definition of $\mathit{safe}_n(c, s, \mu, J, Q)$ by natural number induction on n. $\mathit{safe}_0(c, s, \mu, J, Q)$ holds always. And for $n \in \mathbb{N}$, $\mathit{safe}_{n+1}(c, s, \mu, J, Q)$ holds iff the following conditions are true:

1. If $c = \mathsf{skip}$ then $(s, \mu, \mathbb{P}) \models Q$.
2. For all μ_0, μ_J, μ_F such that

   (i) $\mu_0 \in (\mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} \mu))$,

(ii) $\mathit{lock\text{-}complete}(\mu_0)$, and

(iii) either $(s, \mu_J, \mathbb{P}) \models J$ or $\mathit{locked}(\mu_0)$,

$c, (s, \mu_0) \not\twoheadrightarrow \lightning$.

3. For all $\mu_0, \mu_1, \mu_J, \mu_F, c', s'$ such that

   (i) $\mu_0 \in (\mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} \mu))$,

   (ii) $\mathit{lock\text{-}complete}(\mu_0)$,

   (iii) either $(s, \mu_J, \mathbb{P}) \models J$ or $\mathit{locked}(\mu_0)$, and

   (iv) $c, (s, \mu_0) \rightarrow c', (s', \mu_1)$,

   there exists μ'_J, μ'_F, μ' such that

   (a) $\mu_1 \in (\mu'_J \mathbin{\widetilde{*}} (\mu'_F \mathbin{\widetilde{\#}} \mu'))$,

   (b) $\mu'_F \preceq \mu_F$,

   (c) either $(s', \mu'_J, \mathbb{P}) \models J$ or $\mathit{locked}(\mu_1)$, and

   (d) $\mathit{safe}_n(c', s', \mu', J, Q)$.

The definition of the predicate above differs from Vafeiadis' in three ways. First, the separating conjunctions are obviously different and, in particular, there are two different separating conjunctions used: spatiotemporal separation for framing and spatial separation for partitioning the local from the shared state. The spatiotemporal separator is used for framing because it subsumes spatial and temporal frames, as well as any combination of spatial and temporal frames. It may be possible to make use of $\widetilde{\#}$ uniformly in the definition, which would yield a stronger notion of specification, but recent theoretical results [23], which describes the importance of a commutative notion of separation for the concurrency rule, makes this seem unlikely. (Recall that spatial separation is commutative, but not spatiotemporal separation is not.)

Second, the frame state is allowed to change from one step to another, but only by making silent transitions. This reflects the fact that specifications are oblivious to the nondeterministic flushing transitions required by the memory model and described by the program semantics—the program logic allows for reasoning about specifications at a level that is generally higher than the level of nondeterministic buffer flushing.

Third, because there is no inherent notion of atomicity in this language and memory model, we cannot require that the system state *always* be partitionable so that one cell satisfies the invariant assertion. For even while one processor holds the lock, others may well continue to execute by, e.g., storing to their write buffers, assigning to identifiers, etc. Hence, the invariant condition from Vafeiadis' safety predicate is weakened to require only that the invariant holds while the lock is available. This allows a process to temporarily violate the invariant after acquiring the lock. In these states, other processes of course cannot rely on the shared state satisfying the invariant, but this is generally not a concern because, while the other processes can continue to execute while the invariant is violated, they cannot read the shared state; instead, they are blocked until the lock is released.

4.5.3 Soundness

For the sake of a soundness theorem about the proof system described above, we say that a *proof* is a tree of specifications, in which the leaves are instances of axiom schemas, and the internal specification nodes are instances of the conclusion of some inference rule, with the children of that node as instances of the hypotheses of the inference rule. We write $\overline{J \vdash \{P\}\ c\ \{Q\}}$ to indicate that there exists some proof for which the root of the tree is labeled with $J \vdash \{P\}\ c\ \{Q\}$. The soundness theorem asserts that provable specifications are true:

Theorem 1 (Soundness). $\overline{J \vdash \{P\}\ c\ \{Q\}}$ *only if* $J \models \{P\}\ c\ \{Q\}$.

Proof. By induction on the structure of an arbitrary proof tree, using the soundness lemmas in Section A.2. $\square$

Chapter 5

Loose Ends

This chapter describes open questions about the current program logic, and also some notable features that were explored but not been adopted either because they were considered narrowly unsuitable, or not sufficiently well understood, or simply due to lack of time.

5.1 Top Assertions

Previous iterations of the logic included an additional right-side frame rule for the strong temporal separating conjunction:

$$\frac{J \vdash \{P\}\ c\ \{Q\} \quad \mathrm{fv}(R) \cap \mathrm{mod}(c') = \emptyset}{J \vdash \{P \blacktriangleleft R\}\ c\ \{Q \blacktriangleleft R\}} \qquad (\textsc{frame-stm-r})$$

The intuition behind this rule is that additional, more recent writes to distinct locations are irrelevant to the load command. In fact, this frame rule allows for the following very small load axiom, which does not require parametrization by any additional formulas:

$$J \vdash \{e \rightsquigarrow_{e'} e''\}\ x := [e]_{e'}\ \{e \rightsquigarrow_{e'} e'' \wedge x = e''\} \qquad (\textsc{load-sm})$$

The load axiom defined in Chapter 4 is then derivable from the smaller load axiom and the right-side strong temporal frame rule:

$$\dfrac{\dfrac{\dfrac{}{J \vdash \{e \rightsquigarrow_{e'} e''\}\ x := [e]_{e'}\ \{e \rightsquigarrow_{e'} e'' \wedge x = e''\}}\ \text{LOAD-SM}}{J \vdash \{e \rightsquigarrow_{e'} e'' \blacktriangleleft P\}\ x := [e]_{e'}\ \{(e \rightsquigarrow_{e'} e'' \wedge x = e'') \blacktriangleleft P\}}\ \text{FRAME-STM-R}}{J \vdash \{e \rightsquigarrow_{e'} e'' \blacktriangleleft P\}\ x := [e]_{e'}\ \{(e \rightsquigarrow_{e'} e'' \blacktriangleleft P) \wedge x = e''\}}\ \text{CONS}$$

Unfortunately this frame rule is not sound for the store command, which only adds new writes to the "top" of the write buffer and never in the middle. For example, the following derived specification is false:

$$\dfrac{\dfrac{}{J \vdash \{x \mapsto -\}\ [x] := 1_0\ \{x \mapsto 0 \lhd x \rightsquigarrow_0 1\}}\ \text{STORE}}{J \vdash \{x \mapsto - \blacktriangleleft y \rightsquigarrow_0 2\}\ [x] := 1_0\ \{(x \mapsto 0 \lhd x \rightsquigarrow_0 1) \blacktriangleleft y \rightsquigarrow_0 2\}}\ \text{FRAME-STM-R}$$

This specification is false because, from a state with a buffered write $y \rightsquigarrow_0 2$, a store command will always add a succeeding write, not a preceding write as above.

We can work around this problem with a new assertion, $\mathbf{top}_e$, which describes an empty write buffer that can only be extended with preceding and not succeeding writes. For example, $x \rightsquigarrow_0 1$ describes a part of the write buffer 0, which may be extended using either the left- or right-side separating conjunctions. But $x \rightsquigarrow_0 1 \lhd \mathbf{top}_0$ describes specifically the top part of write buffer 0, and may be extended only with preceding writes. This is accomplished by augmenting yet again the notion of generalized multiprocessor memory system with an additional set of "top-completeness" flags, analogous to the buffer-completeness flags Γ, and redefining the separating conjunctions so that, e.g., $\mathbf{top}_0 \lhd y \rightsquigarrow_0 2$ is inconsistent. We then update the axiom for the store command to specifically require that the top of the write buffer be described in the pre-condition, so that a new top may be specified in the post-condition:

$$J \vdash \{(e \rightsquigarrow_{e'} e'' \blacktriangleleft P) \lhd \mathbf{top}_{e'}\}\ [e] := f_{e'}\ \{(e \rightsquigarrow_{e'} e'' \blacktriangleleft P) \lhd e \rightsquigarrow_{e'} f \lhd \mathbf{top}_{e'}\} \tag{STORE-TOP}$$

The fence and lock-manipulation axioms require similar updates.

Although this model simplifies the load axiom and strengthens the proof theory, it also significantly complicates the assertion language—the addition of top-completeness flags would complicate an already complex model of assertions. Furthermore, the details of the redefinition of the separating conjunction are far from clear. For example, what is the meaning, if any, of $\mathbf{top}_e \lhd \mathbf{bar}_e$? Consequently, although the idea of symmetric temporal frame rules is interesting, it is not clear that this would be an improvement overall compared to the assertions and proof theory described in Chapter 4.

5.2 Additive Barrier Assertions

The current definition of the barrier assertions $\mathbf{bar}_i$ yields an expressive logic, but at the expense of a somewhat more complicated model. In particular, the buffer-completeness flags Γ are present in the semantics of assertions solely to model the $\mathbf{bar}_i$ assertion. This is the best solution of those that have been explored, but a simpler, less expressive alternative semantics for $\mathbf{bar}_i$ is possible. In particular, instead of distinguishing between states and more elaborate models, we may give a semantics to assertions directly in terms of states. In this simplified semantics, the meaning of most assertions is unchanged (though, of course, without the buffer-completeness flags), and the meaning of $\mathbf{bar}_i$ is the set of states in which buffer i is fully flushed:

$$(s, h, B, K) \models \mathbf{bar}_i \equiv_{df} B = \mathcal{E}[\hat{s}(i) \leftarrow \varepsilon]\,.$$

Note that, in this definition, the heap is arbitrarily defined, as well as all the write buffers other than i. In this model, $\mathbf{bar}_i$ has more in common with the additive unit **true** than the multiplicative unit **emp**. To describe a flushed write, instead of using a multiplicative conjunction $e \rightsquigarrow_i f \lhd \mathbf{bar}_i$, we use an additive conjunction $e \rightsquigarrow_i f \wedge \mathbf{bar}_i$.

By unifying states and models, the semantics of assertions and specifications is drastically simplified, but reduced expressiveness of the assertion language makes the program logic more complicated and less effective. For one, it is not possible to give a "small" axiom to the fence, lock and unlock commands. The given axiom for fence:

$$J \vdash \{\mathbf{emp}\}\ \mathsf{fence}_i\ \{\mathbf{bar}_i\} \qquad (\textsc{fence})$$

is sound but drastically weakened: the post-condition, after all, describes an arbitrary set of heap-allocated locations. To restore the original spirit of the axiom, we must make it "large," incorporating the entire (relevant) system state as follows:

$$J \vdash \{P\}\ \mathsf{fence}_i\ \{P \wedge \mathbf{bar}_i\} \qquad (\textsc{fence-lg})$$

Such large axioms, formed using the additive conjunction, are less useful in the context of a separation-style logic because there is no additive frame rule.

In the case of the ATOMIC inference rule, the situation is even worse:

$$\frac{\mathbf{emp} \vdash \{(J * P * \mathbf{lock}_i) \lhd \mathbf{bar}_i\}\ c\ \{(J * Q * \mathbf{lock}_i) \mathrel{\lhd\!\!-} \mathbf{bar}_i\}}{J \vdash \{P\}\ \mathsf{lock}_i\,; c\,\mathsf{unlock}_i\ \{Q\}} \ (\textsc{atomic})$$

Soundness of this rule relies crucially on the use of the temporal separating implication in the post-condition of the antecedent, in which, in the current model, $P \mathrel{\lhd\!\!-} \mathbf{bar}_i$ denotes the set of states which, when flushed on processor i, satisfy P. But in the simplified model, the separating implication $P \mathrel{\lhd\!\!-} \mathbf{bar}_i$ does not have the intended meaning.[1] The problem lies in the incompatibility between the multiplicative separating conjunctions and implication, and the additive semantics of the

[1]Or, really, any coherent or intuitive meaning.

barrier assertion. We might also consider using an additive implication $P \Leftarrow \mathbf{bar}_i$, but neither is this intuitively correct (it describes either flushed states that satisfy P states, or arbitrary non-flushed states), nor is additive implication (along with negation) technically feasible for the reasons previously discussed in Section 3.5.1.

5.3 Permissions

In Concurrent Separation Logic, the concept of *permissions* (or *shares*) has been fruitful. The idea originated from the ownership interpretation of separation logic assertions, in which $e \mapsto f$ is read as an assertion of complete ownership of the address e, thus effectively granting the command which has this assertion as a pre-condition full permission to access and modify the location e. There can be no concurrent modification of the value at e because, in order to do so, a command would also require $e \mapsto -$ in its pre-condition, but the parallel composition rule requires the pre-conditions of parallel commands to be disjoint. For example, consider the following CSL command specifications:

$$J \vdash \{x \mapsto -\}\ [x] := 1\ \{x \mapsto 1\}$$
$$J \vdash \{y \mapsto -\}\ [y] := 2\ \{y \mapsto 2\}$$

Using the rule of of composition we can derive the following combined specification for $[x] := 1 \parallel [y] := 2$:

$$J \vdash \{x \mapsto - * y \mapsto -\}\ [x] := 1 \parallel [y] := 2\ \{x \mapsto 1 * y \mapsto 2\}$$

This rule is sound because the first command has sole ownership of x and the second of y. If the commands were to share ownership of a single address—i.e., if $x = y$—then there would be a data race on that address. This is ruled out by the pre-condition which requires that $x \neq y$.

Consider, however, a pair of load commands that share an address:

$$J \vdash \{x \mapsto 1\}\ t := [x]\ \{x \mapsto 1 \wedge t = 1\}$$

$$J \vdash \{x \mapsto 1\}\ u := [x]\ \{x \mapsto 1 \wedge u = 1\}$$

We might like to prove the following combined specification of $t := [x] \parallel u := [x]$:

$$J \vdash \{x \mapsto 1\}\ t := [x] \parallel u := [x]\ \{x \mapsto 1 \wedge t = 1 \wedge u = 1\}$$

Unfortunately this is not provable. The parallel composition rule, in particular, yields the following specification:

$$J \vdash \{x \mapsto 1 * x \mapsto 1\}\ t := [x] \parallel u := [x]\ \{(x \mapsto 1 \wedge t = 1) * (x \mapsto 1 \wedge u = 1)\}$$

This is vacuously true because the pre-condition is inconsistent, and certainly not equivalent to the desired specification above.

The problem with Concurrent Separation Logic is that both load commands must claim sole ownership of the address x. This requirement is designed to ensure that only race-free commands have derived specifications, but clearly in this case there are no data races. In the fractional permission model of Concurrent Separation Logic—first introduced by Boyland and Bornat [8, 7] and later refined by Parkinson and Dockins et al. [40, 16]—each memory address is associated with a real-numbered permission r with $0 < r \leq 1$. The operational semantics is adjusted so that a store command requires full permission to execute safely, while the load command only requires non-zero permission. The separating conjunction simply combines permissions by adding them, with the operation being undefined if the sum is greater than 1. For example, $x \overset{0.5}{\mapsto} 1 + x \overset{0.5}{\mapsto} 1 \equiv x \overset{1}{\mapsto} 1$ and $x \overset{0.5}{\mapsto} 1 + x \overset{1}{\mapsto} 1 \equiv \textbf{false}$.

Using the fractional permission model, we can prove the desired specification of $t := [x] \parallel u := [x]$. Instead of assigning each command full permission to the address x we give each half, which is sufficient to show that each command individually has the desired effect:

$$J \vdash \left\{x \overset{0.5}{\mapsto} 1\right\}\ t := [x]\ \left\{x \overset{0.5}{\mapsto} 1 \land t = 1\right\}$$
$$J \vdash \left\{x \overset{0.5}{\mapsto} 1\right\}\ u := [x]\ \left\{x \overset{0.5}{\mapsto} 1 \land u = 1\right\}$$

The parallel composition rule is then used to derive

$$J \vdash \left\{x \overset{0.5}{\mapsto} 1 * x \overset{0.5}{\mapsto} 1\right\}\ t := [x] \parallel u := [x]\ \left\{(x \overset{0.5}{\mapsto} 1 \land t = 1) * (x \overset{0.5}{\mapsto} 1 \land u = 1)\right\}$$

And now the pre-condition of this specification is equivalent to $x \overset{1}{\mapsto} 1$ and the post-condition to $x \overset{1}{\mapsto} 1 \land t = 1 \land u = 1$, as desired.

An alternative to the fractional model of permissions is the *counting model.* In this model, permissions are integers, the constituent integers are added in the model of a separating conjunction, and the full permission, which is required for writing, is modeled by 0. Conceptually, a points-to assertion with counting permission $n \geq 0$ is thought of as a "source" from which n read-only permissions have been derived. A characteristic equivalence of the counting permission model is as follows:

$$e \overset{n}{\mapsto} e' \land n \geq 0 \Leftrightarrow e \overset{n+1}{\mapsto} e' * e \overset{-1}{\mapsto} e'$$

Some (strong-memory) concurrent programs are more amenable to verification using counting permissions versus fractional splitting permissions depending on the interaction among threads. Part of the goal of this dissertation project is to explore what aspects of the memory model—which implicitly and concurrently interacts with all programs—can be hidden or incorporated directly into a weak-memory program logic. From this standpoint, both splitting and counting permissions have been a fruitful source of ideas, some of which are described informally in the following sections.[2]

5.3.1 A Use for Splitting Permissions

A variation on splitting permission could significantly extend the programs about which the concurrent weak-memory logic is capable of reasoning: namely, to more racy programs. Consider a program that performs no locking whatsoever, in which at most one thread at a time may write to a shared region of memory, while the other threads may only read. If the shared invariant relates more than one memory location in a non-trivial way, then the "daring" rules from Section 4.5.1, which allow individual reads and writes to shared memory without requiring that the global lock be acquired first, may be insufficient because it may not be possible to show that individual writes necessarily preserve the invariant.

For example, consider a message-passing program with two shared memory locations: address d, which contains a data value; and address r which contains a "ready" flag. The invariant, informally, is that whenever the ready flag is set the value of the data flag is also set. Formally, let J be the following invariant:

$$J =_{df} \exists r' : \exists d' : r \mapsto r' * d \mapsto d' \wedge (r' \neq 1 \vee d' = 1).$$

There may be multiple copies of a reading (or receiving) thread, which may load r and d at any time without first acquiring a lock. The "daring" load axiom is sufficient to describe the results of these loads. There is also a single writing (or sending) thread c_s, which updates these locations in accordance to the invariant, defined as follows:

$$c_s =_{df} [d] := 1 ; [r] := 1.$$

[2]And also in a longer paper [57].

Intuitively, it is clear that this thread maintains the invariant, because it only sets the ready flag after first setting the data flag; and no other threads may interfere, resetting the data flag between these two writes.

The "daring" store axiom, however, is not sufficient to show that this thread maintains the invariant J. The first store is not problematic: regardless of whether $r' \neq 1$ or $d' = 1$ to begin with, after setting d, it will be the case that $d' = 1$, which satisfies the invariant. But an attempt to apply the daring store axiom to the second write will fail, because it is only safe to set the ready flag when it is known that $d = 1$. But this knowledge was lost after the first application of the daring store axiom; again, we only know that one of $r' \neq 1$ and $d' = 1$ holds.

Suppose, however, we had a modified notion of splitting permissions such that, as in Bornat's model, only locations with permission 1 are writable, but also in which only locations with permission not less than $\frac{1}{2}$ are allowed to have buffered writes. Let us parametrize the invariant J by some permission p as follows:

$$J_p =_{df} \exists r' : \exists d' : r \overset{p}{\mapsto} r' * d \overset{p}{\mapsto} d' \wedge (r' \neq 1 \vee d' = 1).$$

This allows us to separate the invariant into a read-only part $J_{\frac{1}{4}}$ that can be shared among the reading threads, and a "bufferable" part $J_{\frac{3}{4}}$ that can be incorporated into the private state of the writing thread, yielding the following equivalence:

$$J_1 \equiv J_{\frac{1}{4}} * J_{\frac{3}{4}}$$

The daring store axiom can then be used to acquire temporary, lock-free access to the read-only part of the invariant and thus, when combined with the bufferable part, giving the writing thread sufficient permission to perform the write. A proof sketch is as follows:

$\{J_{\frac{3}{4}}\}$

$$
\begin{array}{l}
\qquad [d] := 1_0 \\
\{J_{\frac{3}{4}} \lhd d \rightsquigarrow_0 1\} \\
\qquad [r] := 1_0 \\
\{J_{\frac{3}{4}} \lhd d \rightsquigarrow_0 1 \lhd r \rightsquigarrow_0 1\}
\end{array}
$$

The reason that this proof sketch might be expanded into a full proof—assuming, hypothetically, the existence of a suitable notion of permission—is that the second write can be shown to satisfy the invariant because it takes place in the context of the first write, $d \rightsquigarrow_0 1$.

The existence of a suitable permission model has not yet been demonstrated, however. A significant problem that has arisen with candidate models is the requirement that assertions denote sets of states that are closed w.r.t. the flushing order. With the naive model, this property is easily violated. For example, the models of the assertion $x \overset{\frac{1}{4}}{\mapsto} 1 * (x \overset{\frac{3}{4}}{\mapsto} 1 \lhd x \overset{\frac{3}{4}}{\rightsquigarrow}_0 2)$ are not closed, because the models of the left-hand assertion are compatible with only the pending-write models of the right-hand assertion (because the heap values match), but not the flushed models (because the heap values, 1 and 2 respectively, do not match). It is not yet clear how to overcome this problem.

5.3.2 A Use for Counting Permissions

Counting permissions have a different potential use in the weak-logic: namely, for axiomatization of memory management commands, and unification of the strong and weak temporal conjunctions.[3] With the assertions given thus far, it is possible to describe successive writes to the same location with the weak temporal conjunction, but not of course with the strong temporal conjunction; to do so would violate the

[3]This section is adapted from an earlier paper [57].

disjointness requirements of the separating conjunctions. For example,

$$e \mapsto e' \blacktriangleleft e \leadsto_0 e'' \;\equiv\; \mathbf{false}$$

because no models of the first conjunct are compatible with any models of the second—the allocated location e cannot be separated across the sequential separating conjunction.

To unify the strong and weak temporal conjunctions with permissions, let us associate with each allocated location an element of an *asymmetric counting* permission model, in which permissions are integers and permissions are combined by a function $\oplus$. The model is inspired by the previously described counting permissions of Bornat et al. [7], but here, instead of addition, the operation $\oplus$ is partial and asymmetric, defined as follows:

$$a \oplus b = \begin{cases} a + b & \text{if } a < 0 \wedge (b < 0 \vee -b \leq a) \\ \perp & \text{otherwise.} \end{cases}$$

The strong temporal conjunction is redefined with a relaxed notion of compatibility: locations allocated in both states must have compatible (i.e., not $\perp$) permissions. (The spatial separating conjunction remains unchanged, requiring disjointness of allocated locations.) Syntactically, points-to and leads-to assertions are annotated with integers (e.g., $e \leadsto_n e'$) that denote their location's permission value.

To understand the intuition behind asymmetric counting permissions, first recall Bornat's original counting permission model. There, negative integer annotations denote read-only permission for the location, and nonnegative integer annotations ("source" annotations) indicate the number of read-only permissions that have been split off. In particular, a zero annotation indicates full permission. For example,

$$x \overset{-1}{\mapsto} 1 * x \overset{-1}{\mapsto} 1 * x \overset{2}{\mapsto} 1,$$

is a consistent formula in Bornat's logic with full permission ($-1 + -1 + 2 = 0$), in which two read-only assertions have been split off of the original assertion.

The asymmetric model can be derived from Bornat's counting model by replacing the separating conjunction with the sequential conjunction, and requiring that permissions are combined with $\blacktriangleleft$ in the order shown in the example above, from negative to positive. Then we can interpret a nonnegative annotation as denoting the most-recently written (top-most) value, where the particular value indicates the number of prior values, if any. Such prior values are indicated by negative annotations, and the full permission by zero. For example, the following is a consistent formula that describes two successive writes on buffer i to location x:

$$x \overset{-1}{\rightsquigarrow}_i 1 \blacktriangleleft x \overset{1}{\rightsquigarrow}_i 2,$$

because $-1 \oplus 1 = 0$. The following, on the other hand, are inconsistent:

$$x \overset{-1}{\rightsquigarrow}_i 1 \blacktriangleleft x \overset{-1}{\rightsquigarrow}_i 2 \blacktriangleleft x \overset{1}{\rightsquigarrow}_i 3 \quad \text{and} \quad x \overset{0}{\rightsquigarrow}_i 1 \blacktriangleleft x \overset{-1}{\rightsquigarrow}_i 2$$

because the top-most write, annotated with $n \geq 0$, must succeed no more than n earlier writes ($(-1 \oplus -1) \oplus 1 = -2 \oplus 1 = \bot$), and no writes may follow the top write, annotated with a non-negative integer ($0 \oplus -1 = \bot$).

Using the asymmetric counting permission described above, we may consider axiomatization of primitives for dynamic memory management; namely, dynamic allocation via the command $x := \mathsf{new}(e)$, and dynamic disposal via the command $\mathsf{free}(e)$. The semantics of these commands are not defined at the level of the memory model, so there is some choice about what operational meaning to give them. In practice, these commands are typically implemented using fence commands to ensure system-wide consistency. In contrast, we have chosen to extricate barriers from their meaning, primarily to explore the circumstances in which barriers are actually needed. (The typical semantics of allocation and disposal can be recovered by explicitly adding fence instructions before and after these commands.)

Absent a succeeding barrier, allocation is perfectly natural; it simply adds a new write to an unallocated location to the top of the write buffer:

$$J \vdash \{\mathbf{emp}\}\ x := \mathsf{new}(e)_i\ \left\{x \overset{0}{\leadsto}_i e\right\} \tag{NEW}$$

Note that the write in the post-condition has full permission, which ensures that earlier writes, framed from the left, have distinct locations. Otherwise, this could result in a duplicate allocation.

The meaning of the free command absent a preceding barrier is less clear. We axiomatize a conservative semantics: if a location has at most one value in the system, it may be deallocated. In particular, we make no attempt to describe the outcome of deallocation without a barrier when there are multiple pending writes. Perhaps in this case the command should fault, or perhaps all pending writes should be removed from the system, leaving writes to other locations unaffected. In any case, the following axiom describes the conservative semantics and does not allow anything to be proved in the less clear cases:

$$J \vdash \left\{e \overset{0}{\leadsto}_i -\right\}\ \mathsf{free}(e)_i\ \{\mathbf{emp}\} \tag{FREE}$$

Symmetric to the case for allocation, the write in the pre-condition has full permission, which in this case prevents earlier writes to the same location from being framed on from the left, yielding a double disposal. By using the rule of consequence, the pre-condition may be strengthened from a leads-to assertion to a points-to assertion, thus axiomatizing disposal of shared memory.

5.4 Shared Variables

In Concurrent Separation Logic, threads may communicate not just through the heap, but through shared variables as well. Elaborate rules govern these interactions, which must be well synchronized, to ensure soundness of the logics. Even for traditional, strong-memory logics, this is notoriously difficult; indeed, a counter-example to the soundness of CSL was recently (and quite unintentionally) discovered by the author—and described in detail by Brookes along with a minor revision to the logic to avoid the problem [10], and also by Reddy accompanying a more radical revision to the logic [46]—which exploited a subtle problem with the inference rules that govern variable communication.

In the weak-memory logic described in this dissertation, communication via variables is completely prohibited—the rule for concurrent composition explicitly requires that the modified variables of each thread are distinct from the free variables of the others. This is a conservative condition that rules out many interesting programs, with the goal of simplifying the program logic and focusing the project on the study of communication via the weak memory system. Furthermore, there is reason to believe that allowing variable communication in the weak-memory logic would be even more subtle and problematic than in traditional logics. Of course, communication among threads executed on different processors can only happen by loading and storing through the memory system, and there is little reason to think that variable assignment is, in general, a suitable abstraction of this process. It seems likely though that under some conditions—namely, when shared-variable assignments are consistently protected by fence or lock instructions—it is possible to reason soundly about programs with such interaction, but it remains to be determined exactly what those conditions are.

5.5 Invariant Expansion

In the semantics of specifications given in Section 4.5.2, the shared invariant component J of a specification $J \vdash \{P\}\ c\ \{Q\}$ is interpreted rather literally: if the lock is not held, then there must be a portion of the complete state which satisfies J exactly. To reason about the specification of commands that accesses shared state, it is therefore necessary to prove always that, once the shared access is finished, there is some portion of the state that satisfies the invariant. For many programs, however, this is difficult for the following reason. It is often most convenient to specify the shared invariant by describing the legal values in the heap—i.e., with spatial conjunctions of points-to formulas. But points-to formulas do not simply constrain values of the heap; they also specify emptiness of write buffers. Consequently, a program which buffers writes to the shared state without first acquiring the lock cannot possibly satisfy a heap-only shared invariant. For example, even a very simple invariant like $x \mapsto 1$ cannot be satisfied by the program $[x] := 1_i$ because it is not the case that:

$$x \mapsto 1 \lhd x \leadsto_i 1 \models x \mapsto 1.$$

It seems clear, however, that in at least some circumstances the above command ought to satisfy its simple specification because, observationally, a process cannot distinguish a state that satisfies $x \mapsto 1 \lhd x \leadsto_i 1$ from one that satisfies the specification $x \mapsto 1$. In both cases the address x is allocated, and the result of loading x on any processor is 1. Also note that the command does not satisfy the more liberal specification $x \mapsto 1 \lhd x \leadsto_i 1$ for the same reason, which is that the following entailment is not true:

$$x \mapsto 1 \lhd x \leadsto_i 1 \lhd x \leadsto_i 1 \models x \mapsto 1 \lhd x \leadsto_i 1.$$

An alternative semantics of specifications might relax the interpretation of the shared invariant so that, if the lock is not held, there is always a portion of the state such that, as its buffered writes flush to memory, the heap part satisfies the invariant J. More formally, let us write *trim* for the function that replaces the write buffers in a memory system with empty buffers:

$$\mathit{trim}(h, B, K) =_{\mathit{df}} (h, \mathcal{E}, K).$$

We might then redefine the semantics of specifications such that there must always be a portion μ_J of the state such that $\mathit{trim}(\mu_J)$ satisfies the invariant J. Note that, because of the implicit flushing steps incorporated into the semantics of commands, this implies that regardless of which flushing steps take place, that trimming the substate μ_J satisfies J. Hence, the heap part of the shared substate always satisfies, in this relaxed way, the invariant J as it commits.

Under this semantics of specifications, a more liberal rule for invariant reasoning could be admitted, in which the command must ensure that a portion of state always satisfies the *expansion* of the invariant J, written $\mathbf{exp}(J)$. Informally, a state satisfies the expansion of J if, as in the relaxed semantics, as it flushes its heap part always satisfies J. Formally:

$$s, \mu, \Gamma \models \mathbf{exp}(J) \equiv_{\mathit{df}} \forall \mu', \Gamma' : (\mu', \Gamma') \leq (\mu, \Gamma) \Rightarrow s, \mathit{trim}(\mu'), \Gamma' \models J.$$

The relaxed invariant rule is as follows:

$$\frac{\mathbf{emp} \vdash \{\mathbf{exp}(J) * P\}\ c\ \{\mathbf{exp}(J) * Q\}}{J \vdash \{P\}\ c\ \{Q\}} \quad (\textsc{inv-exp})$$

Note that this invariant rule does not require that lock acquisition precede shared state manipulation.

While this semantics and invariant rule seems, in principle, superior to those presented in Chapter 4, it has disadvantages as well. Most significantly, it is not clear how to reason about the $\mathbf{exp}(J)$ formula. It seems as though it should be possible to load and store in the context of $\mathbf{exp}(J)$ just as in the context of J, but it is not clear how to connect this intuition with the assertion logic. Consider again the invariant $x \mapsto 1$. If the rule INV-EXP is used to access this shared state in the guise of $\mathbf{exp}(x \mapsto 1)$ then to usefully apply the load axiom as presented in Section 4.5.1 we must exhibit an assertion P such that $\mathbf{exp}(J) \models x \mapsto 1 \blacktriangleleft P$ and $(x \mapsto 1 \blacktriangleleft P) \lhd x \leadsto_i 1$. Considering the relatively loose definition of $\mathbf{exp}(J)$, it is not clear what such an assertion would be.

Furthermore, in those situations for which access to shared state is mediated by the global lock, such a relaxed semantics is undesirable. When the threads of a program take care to serialize their access to shared resources, flushing their buffers before and after such accesses, it is an entirely unnecessary complication to assume otherwise. Ideally a semantics and proof theory would be found which allows reasoning about both kinds of programs, but that remains as future work.

Chapter 6

Related Work

Technical inspiration for this project comes primarily from work on separation logic [47, 36, 35] and abstract separation logic [11], as well as concurrent separation logic [34, 9], which this program logic resembles insofar as it strives to enable local (instead of global) reasoning about shared-state invariants (instead of two-place state relations). Earlier iterations of the machine and programming language models was influenced by work on graphical models [58, 25, 26] and the pomset model of true concurrency from Pratt [43, 44]. The style of semantics of specifications, and the associated soundness proof, is taken almost directly from Vafeiadis' recent soundness proof of concurrent separation logic [54]. Vafeiadis' excellent dissertation has also been an invaluable guide [53].

The following sections discuss other work related to the topic of weak-memory program reasoning and verification.

6.1 Weak-Memory Program Transformations

The approach to weak-memory program reasoning that is perhaps technically simplest is based on program transformations. The basic idea is to use existing techniques to reason about programs that have been modified to directly incorporate the behavior of the memory model on which they are to be executed. For example, in the context of x86, programs would be modified to include as additional state a series of in-memory FIFO queues which represent write buffers, each load and store operation is replaced by calls to subroutines which respectively search the process's queue for an appropriate write and append a new write to the process's queue, and finally an additional thread is composed in parallel which commits writes, with arbitrary timing, from the encoded buffers to memory. This is essentially the tack taken by relatively early work in this area by Ridge [48], in which he uses the Isabell/HOL framework and logic to reason directly about operational semantics of modified Caml programs. In particular, a mechanically verified proof in this style is presented for Peterson's mutual exclusion algorithm [42]. The potential drawback to the program transformation approach is the difficulty of applying a local reasoning principle and, thus, a useful frame rule. Because Ridge uses higher-order logic to reason about the semantics directly, those proofs are undertaken without a frame rule.

It is also conceptually possible to use a separation logic with a strong-memory locality principle and associated frame rule to reason about transformed programs. In the author's early experiments, this approach seemed cumbersome at best. With write buffers encoded as program state, the locations of the writes are encoded as data. This encoding obviates the notion of "separation" that makes separation logic so useful, since it is always possible to frame onto a specification writes which, when flushed, violate the intended specification. And because the nondeterministic flushing of writes is not embedded into the logic, it must be encoded as a disjunction of possible buffered states and reasoned about on a case-by-case basis.

6.2 Program Logics for Weak-Memory Reasoning

Also of note are two works that present solutions to the same weak-memory reasoning problem, both developed much more fully than the logic described in this dissertation. First is Ridge's rely-guarantee program logic for the x86-TSO memory model [49]. Ridge's logic is formalized in HOL and has been demonstrated with proofs of a number of interesting algorithms, including Simpsons's 4-slot non-blocking buffer. In contrast to this dissertation project, Ridge's is a logic for the x86 assembly language, whereas the logic described in Chapter 4 targets a higher level, structured language. Additionally, Ridge's logic is not inherently local, and offers nothing like the frame rule of separation logic.

A second work of note is Cohen and Schirmer's [15] reduction from x86-TSO to sequential consistency for certain programs. This is notable because the class of programs they consider is larger than just the well locked programs. They show that many concurrent programming paradigms, although racy, in fact remain sequentially consistent. They furthermore provide a method of syntactic restriction for an Owicki-Gries-style program logic that allows sound reasoning about such programs. Although they describe some useful programs that fall outside of this boundary, this seems to be a work of great practical importance. Although their logic also offers no frame rule, Cohen has suggested in private communication that a similar restriction may be applied concurrent separation logic for sound local reasoning.

Related but less relevant to the current problem, compared to the previous two papers, is work by Ferreira, Feng and Shao which gives soundness proofs for concurrent separation logic in a variety relaxed-memory settings [18]. As with the original soundness proof by Brookes [9], their theorem applies to well locked programs only, which are necessarily sequentially consistent.

6.3 Algebraic Models of Concurrency

A final thread of related work by Hoare et al. considers algebraic models of concurrent programs and logics for reasoning about such programs [58]. The thread begins with the introduction of *graphical models*, a true-concurrency model, like Pratt's pomset model [43, 44] or Winskel's event structures [59]. Graphical models are a simple but amazingly expressive formalism; recent work has even shown how they can be used to give weak-memory semantics of various kinds to concurrent programs [27].

A graphical model is a finite partial order over some set of events. The labels on events are chosen based on the features of the programming language one wishes to model; examples include variable assignments; memory loads, stores and fences; dynamic memory allocation and disposal; lock acquisitions and releases; reads and writes to communication channels; and thread forks and joins. Directed edges between events represent control flow and data flow. Each graph represents a single execution of a program; a program is represented by a set of graphs indicating its possible executions; and program features are described in terms of constraints on the graphs.

For example, a variable assignment $x := e$ might be described as a single event labeled by $x := v$, where v is the valuation of e, with incoming edges labeled with the names, and respective values, of the variables found in the expression e, and outgoing edges labeled by x and the value v. In this case, the incoming edges represented data flows needed to evaluate the expression e, and the outgoing edges represent data flows of the new value of x that can be used to evaluate later expressions. Another example is the sequential composition of commands $c_0 ; c_1$, which are modeled by graphs that can be partitioned such that there are no "backward" edges from an event of c_1 to an event of c_2.

Graphical models have also been used to show the soundness of concurrent program logics [58]. When modeling a program logic, assertions simply denote sets graphical models just the same as programs—a simplification that yields soundness much more easily than with traditional models. Later, models of program logics were generalized to an abstract algebraic characterization of graphical models. These algebras are known as *Concurrent Kleene Algebras* [25, 24], so-called because they extend traditional Kleene Algebras with operators for both sequential and concurrent composition.

The laws of Concurrent Kleene Algebras are further relaxed in a broader study of algebras for concurrent reasoning [23], which resulted in the discovery of *locality bimonoids*, found to be sufficient for modeling logics that embody a local reasoning principle. Some results from work on locality bimonoids influenced this work. In particular, those results showed that, while frame rules do not require commutative logical operations, concurrency rules do—this indicated that the semantics of specifications, which describes the separation of the shared state from the private state using commutative spatial separation can likely not be strengthened by substituting a weaker non-commutative notion of separation, e.g., spatiotemporal separation.

Chapter 7

Conclusion

This dissertation project has presented a new program logic for reasoning about the behavior of structured, C-like concurrent programs w.r.t. a weak, x86-like model of memory. The programming language has a novel semantics that unifies the description of parallel and interleaved execution; the novel assertion language provides a natural and concise language for describing x86-like system states, and is oblivious to the non-deterministic flushing of buffered writes; and the program logic—similar in shape and meaning to Concurrent Separation Logic—provides small axioms for the programming language's primitives, including memory fences, and embodies in its frame rules a novel x86-specific principle of local reasoning.

Motivations Reconsidered Why bother building a program logic? The original motivation was as follows. Although program logics are reasonable systems in which to construct hand proofs of arbitrary program properties, they have more recently been shown to be amenable to automation of relatively shallow properties, like memory safety or shape properties. But existing logics cannot be soundly applied to certain fine-grained concurrent programs like concurrent data structures. This is because these programs are typically not well locked and contain races, and so cannot rely on the underlying computer architecture to ensure that their interaction with memory is sequentially consistent. Sequential consistency however is a deep assumption in most existing program logics, hence their inapplicability.

As further motivation, concurrent data structures are inarguably important to computer science given the decline of single-threaded processor performance improvements and the concomitant proliferation of hardware parallelism. At the same time, correctness arguments for concurrent data structures are subtle enough to make informal reasoning extremely difficult. Additionally, these programs are of only modest size, which perhaps gives cause for optimism about their amenability to automated or semi-automated verification. Altogether, this would appear to be an excellent opportunity for the application of an automated formal method.

Now, however, I am less confident in the practical value of this project than I was at the outset, having identified a number of errors of judgment in the original motivation. First, it was wrong to consider the small size of these programs as increasing the viability of their fully automatic reasoning. This is exactly backwards: because these subtle and important programs are so small, it may be entirely practical to consider expert-constructed formal proofs of their correctness using proof assistants like ACL2, Coq or Isabelle. And although constructing these proofs is difficult, surely it is less so than developing a general technique for doing so.

Second, although such programs are clearly racy, it is not clear that their interactions with memory fall outside the bounds of sequentially consistency. And for sequentially consistent programs, it seems unlikely that an approach of such high fidelity w.r.t. the memory model (e.g., with explicit write buffers) will turn out to be the most effective.

Nonetheless, the project still has real scientific merit. First, it is not clear that it is possible to reason about small concurrent data structures in isolation from the clients that access them. If, in fact, this is not possible—if such programs cannot soundly be analyzed by their components—then it could be the case that a weak memory program logic that smoothly generalizes simpler traditional logics could be applied to whole programs, working like a traditional program logic for the less complex aspects of the program, but general enough to handle the racy interactions made possible by the concurrent data structures. The logic described here is not yet up to that task, but may constitute a step in that direction.

Furthermore, this work faces the problem of local reasoning about the behavior of programs executing on a more complicated machine quite directly and gives some indication of how this can be done without relying on simplifying assumptions about memory. Local reasoning techniques can, of course, be quite useful even for hand-constructed formal proofs. In the best case, this work could provide a foundation for practically useful reasoning about a class of difficult programs. In the worst case, it sheds some light on the problem of local program reasoning in general by providing an additional—fairly extreme—data point in the space of program logics, illustrating the difficulty and complexity of reasoning about the behavior of programs w.r.t. a widespread and weak memory model.

Local Reasoning and Sequential Consistency A final note has to do with the relationship between local reasoning and sequential consistency. The logic described in this dissertation was designed up from the memory model; that is, soundness w.r.t. the x86 memory model was a primary requirement, and strongly directed the shape of the assertion and specification languages and proof theory. The second primary requirement was a logic that embodied a local reasoning principle. Because the original goal was to be able to reason about racy concurrent data structures, at every step care was taken to be as liberal as possible while maintaining both goals. For example, the definition of the spatiotemporal conjunction was relaxed over and over again, incorporating more and more possible state configurations, until it seemed as though any additional relaxation would yield an unsound frame rule, thus violating the goal of having a logic suitable for local reasoning. And yet, in the end, it seems—though has not been proven—that the logic that resulted from this design process is incapable of expressing proofs about non-sequentially consistent programs. This was certainly was not intended; indeed, it came about despite direct intention.

Although one could imagine changing or improving many aspects of this logic, it is far from clear how one might proceed to relax its definition further still to incorporate non-sequentially consistent programs given the local reasoning requirement. One is led to wonder whether these two requirements of the logic—both that it directly describes weak-memory systems, and that it admits a useful principle of local reasoning—are not just at odds, but even mutually inconsistent. The apparent failure of this logic to accomplish both goals simultaneously is not, of course, a cogent argument for the impossibility of this task, but it does hint toward the difficulty of doing so.

Appendix A

Additional Lemmas, Proofs and Conjectures

This chapter contains proofs of various results about the assertion and program logics. Section A.1 describes results about closure properties of the models of the separating conjunctions, and Section A.2 describes results about the soundness of the program logic w.r.t. the model given in Chapter 4.

A.1 Flushing Closure Proofs

The following lemma describes a crucial relationship between spatiotemporal separation and the flushing relation.

Lemma 4. *If $\mu \in \mu_1 \mathbin{\widetilde{\#}} \mu_2$ and $\mu \underset{\tau}{\rightarrow} \mu'$, then either there exists μ'_1 such that $\mu' \in \mu'_1 \mathbin{\widetilde{\#}} \mu_2$, or there exists μ'_2 such that $\mu' \in \mu_1 \mathbin{\widetilde{\#}} \mu'_2$.*

Proof. Without loss of generality we may assume that $\mu = (h, B[i \leftarrow (\ell, v) :: b], K)$ for some $i \notin K$, and thus $\mu' = (h[\ell \leftarrow v], B[i \leftarrow b], K)$. Furthermore, we have $h = h_1 \mathbin{\backslash\!\backslash} h_2$, $B(i) = (\ell, v) :: b \in B_1(i) \mathbin{\backslash\!\backslash} B_2(i)$ and $K = K_1 \uplus K_2$, assuming $\mu_1 =$

(h_1, B_1, K_1) and $\mu_2 = (h_2, B_2, K_2)$. The least-recent write of $B(i)$, (ℓ, v), is either the least-recent write of $B_1(i)$ or $B_2(i)$.

In the first case, $B_1(i) = (\ell, v) :: b'_1$, and $b \in b'_1 \backslash\backslash B_2(i)$. Let

$$\mu'_1 = (h_1[\ell \leftarrow v], B_1\left[i \leftarrow b'_1\right], K_1).$$

Because $i \notin K = K_1 \uplus K_2$, it is also the case that $i \notin K_1$, and so $\mu_1 \underset{\tau}{\rightarrow} \mu'_1$. By definedness of $\mu_1 \mathbin{\widetilde{\#}} \mu_2$, we know that if $\ell \in \mathrm{dom}(h_2) \cap \mathrm{dom}(B_1(i))$ then $i \in K$. Hence $\ell \notin \mathrm{dom}(h_2)$, which means that $(h_1 \backslash\backslash h_2)[\ell \leftarrow v] = h_1[\ell \leftarrow v] \backslash\backslash h_2$. It follows that

$$\begin{aligned}\mu' &= (h[\ell \leftarrow v], B[i \leftarrow b], K)\\ &= (h_1[\ell \leftarrow v] \backslash\backslash h_2, B[i \leftarrow b], K_1 \uplus K_2)\\ &\in (h_1[\ell \leftarrow v], B_1\left[i \leftarrow b'_1\right], K_1) \mathbin{\widetilde{\#}} (h_2, B_2, K_2)\\ &= \mu'_1 \mathbin{\widetilde{\#}} \mu_2.\end{aligned}$$

In the second case, $B_2(i) = (\ell, v) :: b'_2$, and $b \in B_1(i) \backslash\backslash b'_2$. Let

$$\mu'_2 = (h_2[\ell \leftarrow v], B_2\left[i \leftarrow b'_2\right], K_2).$$

Again, $i \notin K_2$, and so $\mu_2 \underset{\tau}{\rightarrow} \mu'_2$. Because $(h_1 \backslash\backslash h_2)[\ell \leftarrow v] = h_1 \backslash\backslash (h_2[\ell \leftarrow v])$, it follows that

$$\begin{aligned}\mu' &= (h[\ell \leftarrow v], B[i \leftarrow b], K)\\ &= (h_1 \backslash\backslash (h_2[\ell \leftarrow v]), B[i \leftarrow b], K_1 \uplus K_2)\\ &\in (h_1, B_1, K_1) \mathbin{\widetilde{\#}} (h_2[\ell \leftarrow v], B_2\left[i \leftarrow b'_2\right], K_2)\\ &= \mu_1 \mathbin{\widetilde{\#}} \mu'_2.\end{aligned}$$

□

Lemma 5. *If $\mu \in \mu_1 \mathbin{\widetilde{*}} \mu_2$ and $\mu \underset{\tau}{\to} \mu'$, then either there exists μ_1' such that $\mu' \in \mu_1' \mathbin{\widetilde{*}} \mu_2$, or there exists μ_2' such that $\mu' \in \mu_1 \mathbin{\widetilde{*}} \mu_2'$.*

Proof. Without loss of generality we may assume that $\mu = (h, B[i \leftarrow (\ell, v) :: b], K)$ for some $i \notin K$, and thus $\mu' = (h[\ell \leftarrow v], B[i \leftarrow b], K)$. Furthermore, we have $h = h_1 \uplus h_2$, $B(i) = (\ell, v) :: b \in B_1(i) \uplus B_2(i)$ and $K = K_1 \uplus K_2$, assuming $\mu_1 = (h_1, B_1, K_1)$ and $\mu_2 = (h_2, B_2, K_2)$. The least-recent write of $B(i)$, (ℓ, v), is either the least-recent write of $B_1(i)$ or $B_2(i)$.

In the first case, $B_1(i) = (\ell, v) :: b_1'$, and $b \in b_1' \uplus B_2(i)$. Let

$$\mu_1' = (h_1[\ell \leftarrow v], B_1\left[i \leftarrow b_1'\right], K_1).$$

Because $i \notin K = K_1 \uplus K_2$, it is also the case that $i \notin K_1$, and so $\mu_1 \underset{\tau}{\to} \mu_1'$. By $\mu_1 \smile_* \mu_2$ and $\ell \in \mathrm{dom}(\mu_1)$ we have $\ell \notin \mathrm{dom}(h_2)$, which means that $(h_1 \uplus h_2)[\ell \leftarrow v] = h_1[\ell \leftarrow v] \uplus h_2$. It follows that

$$\begin{aligned}
\mu' &= (h[\ell \leftarrow v], B[i \leftarrow b], K) \\
&= (h_1[\ell \leftarrow v] \uplus h_2, B[i \leftarrow b], K_1 \uplus K_2) \\
&\in (h_1[\ell \leftarrow v], B_1\left[i \leftarrow b_1'\right], K_1) \mathbin{\widetilde{*}} (h_2, B_2, K_2) \\
&= \mu_1' \mathbin{\widetilde{*}} \mu_2.
\end{aligned}$$

The second case is symmetric. □

Lemma 6. *If $\mu = \mu_1 \mathbin{\widetilde{\lhd}} \mu_2$ and $\mu \underset{\tau}{\to} \mu'$, then either there exists μ_1' such that $\mu' = \mu_1' \mathbin{\widetilde{\lhd}} \mu_2$, or there exists μ_2' such that $\mu' = \mu_1 \mathbin{\widetilde{\lhd}} \mu_2'$.*

Proof. Without loss of generality we may assume that $\mu = (h, B[i \leftarrow (\ell, v) :: b], K)$ for some $i \notin K$, and thus $\mu' = (h[\ell \leftarrow v], B[i \leftarrow b], K)$. Furthermore, we have

$h = h_1 \backslash\!\!\backslash h_2$, $B(i) = (\ell, v) :: b = B_1(i) +\!\!+ B_2(i)$ and $K = K_1 \uplus K_2$, assuming $\mu_1 = (h_1, B_1, K_1)$ and $\mu_2 = (h_2, B_2, K_2)$. The least-recent write of $B(i)$, (ℓ, v), is either the least-recent write of $B_1(i)$ or $B_2(i)$.

In the first case, $B_1(i) = (\ell, v) :: b'_1$, and $b = b'_1 +\!\!+ B_2(i)$. Let

$$\mu'_1 = (h_1[\ell \leftarrow v], B_1[i \leftarrow b'_1], K_1).$$

Because $i \notin K = K_1 \uplus K_2$, it is also the case that $i \notin K_1$, and so $\mu_1 \underset{\tau}{\rightarrow} \mu'_1$. By $\mu_1 \smile_{\lhd} \mu_2$, we know that if $\ell \in \mathrm{dom}(h_2) \cap \mathrm{dom}(B_1(i))$ then $i \in K$. Hence $\ell \notin \mathrm{dom}(h_2)$, which means that $(h_1 \backslash\!\!\backslash h_2)[\ell \leftarrow v] = h_1[\ell \leftarrow v] \backslash\!\!\backslash h_2$. It follows that

$$\begin{aligned} \mu'_1 \mathrel{\widetilde{\lhd}} \mu_2 &= (h_1[\ell \leftarrow v] \backslash\!\!\backslash h_2, B_1[i \leftarrow b'_1] +\!\!+ B_2, K_1 \uplus K_2) \\ &= ((h_1 \backslash\!\!\backslash h_2)[\ell \leftarrow v], B[i \leftarrow b], K) \\ &= \mu'. \end{aligned}$$

In the second case, $B_2(i) = (\ell, v) :: b'_2$, $B_1(i) = \varepsilon$ and $b = b'_2$. Let

$$\mu'_2 = (h_2[\ell \leftarrow v], B_2[i \leftarrow b'_2], K_2).$$

Because $i \notin K = K_1 \uplus K_2$, it is also the case that $i \notin K_2$, and so $\mu_2 \underset{\tau}{\rightarrow} \mu'_2$. By definition of function overriding, $(h_1 \backslash\!\!\backslash h_2)[\ell \leftarrow v] = h_1 \backslash\!\!\backslash h_2[\ell \leftarrow v]$. It follows that

$$\begin{aligned} \mu_1 \mathrel{\widetilde{\lhd}} \mu'_2 &= (h_1 \backslash\!\!\backslash h_2[\ell \leftarrow v], B_1 +\!\!+ B_2[i \leftarrow b'_2], K_1 \uplus K_2) \\ &= ((h_1 \backslash\!\!\backslash h_2)[\ell \leftarrow v], B[i \leftarrow b], K) \\ &= \mu'. \end{aligned}$$

□

Lemma 7. *If $\mu \in \mu_1 \mathbin{\widetilde{\bullet}} \mu_2$, for $\bullet \in \{(\#), (*), (\lhd), (\blacktriangleleft)\}$, and $\mu' \preceq \mu$ then there exists*

μ_1', μ_2' *such that* $\mu_1' \preceq \mu_1$, $\mu_2' \preceq \mu_2$ *and* $\mu' \in \mu_1' \mathbin{\widetilde{\bullet}} \mu_2'$.

Proof. By induction on the number of τ steps from μ to μ'. The base case is trivial. Otherwise, assume that $\mu'' \preceq \mu$ and $\mu'' \xrightarrow[\tau]{} \mu'$, and by the induction hypothesis that there exists μ_1'', μ_2'' such that $\mu_1'' \preceq \mu_1$, $\mu_2'' \preceq \mu_2$ and $\mu'' \in \mu_1'' \mathbin{\widetilde{\bullet}} \mu_2''$. In which case the result follows from one of Lemmas 4, 5 or 6, and by transitivity of $\preceq$. □

A.2 Soundness Proofs

The following lemma describes the relationship between the "safety" lemmas and "soundness" lemmas in subsequent sections.

Lemma 8. *Let c be a command and J, P, and Q be assertions. Suppose, for all $n \in \mathbb{N}$, $s \in$ **Stack** and $\mu \in$ **Mem**, that whenever $s, \mu, \mathbb{P} \models P$ it is also the case that $safe_n(c, s, \mu, J, Q)$ holds. Then $J \models \{P\}\ c\ \{Q\}$ holds as well.*

Proof. Immediate from the definition of $J \models \{P\}\ c\ \{Q\}$. □

A.2.1 Soundness of the Axioms

This section presents soundness lemmas for each of the axioms described in Section 4.5.1. That is, for each axiom $J \vdash \{P\}\ c\ \{Q\}$, there is a lemma below which asserts $J \models \{P\}\ c\ \{Q\}$.

Lemma 9. *For all $n \in \mathbb{N}$, if $(s, \mu, \mathbb{P}) \models P$ then $safe_n(\mathsf{skip}, s, \mu, J, P)$.*

Proof. By induction on n. The base case is trivial. For the induction step, we show $safe_{n+1}(\mathsf{skip}, s, \mu, J, P)$ under the assumption that $safe_n(\mathsf{skip}, s, \mu, J, P)$.

1. Because $c = \mathsf{skip}$ we must show that $s, \mu, \mathbb{P} \models P$. This is true by hypothesis.

2. Let μ_0, μ_J, μ_F such that $\mu_0 \in \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} \mu)$, *lock-complete*$(\sigma_0)$ and either $s, \mu_J, \mathbb{P} \models J$ or *locked*(μ_0). We must show that $\mathsf{skip}, s, \mu_0 \not\twoheadrightarrow \lightning$. But the only

evaluation step possible from configuration skip, s, μ_0 is by C-TAU, which never aborts.

3. Let $\mu_0, \mu_J, \mu_F, c_1, \mu_1$ such that $\mu_0 \in \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} \mu)$, *lock-complete*$(\mu_0)$, either $s, \mu_J, \mathbb{P} \models J$ or *locked*(μ_0), and $\mathsf{skip}, s, \mu_0 \rightarrow c_1, s, \mu_1$. We must show μ'_J, μ'_F, μ' such that $\mu_1 \in \mu'_J \mathbin{\widetilde{*}} (\mu'_F \mathbin{\widetilde{\#}} \mu')$, $\mu'_F \preceq \mu_F$, either $s, \mu'_J, \mathbb{P} \models J$ or *locked*(μ_1), and $\mathit{safe}_n(c_1, s, \mu', J, P)$.

   The only evaluation step possible from skip, s, μ_0 is by C-TAU, hence $\mu_1 \preceq \mu_0$. By Lemma 7, there exists μ'_J, μ'_F, μ' such that $\mu_1 \in \mu'_J \mathbin{\widetilde{*}} (\mu'_F \mathbin{\widetilde{\#}} \mu')$, $\mu'_J \preceq \mu_J$, $\mu'_F \preceq \mu_F$, $\mu' \preceq \mu$. By Proposition 2 and $\mu'_J \preceq \mu_J$, if $s, \mu_J, \mathbb{P} \models J$ then also $s, \mu'_J, \mathbb{P} \models J$. Similarly, $s, \mu', \mathbb{P} \models P$ because $s, \mu, \mathbb{P} \models P$ and $\mu' \preceq \mu$. Hence, by the inductive hypothesis we have that $\mathit{safe}_n(\mathsf{skip}, s, \mu', J, P)$.

□

Lemma 10. $J \models \{P\}$ skip $\{P\}$.

Proof. Immediate from Lemmas 8 and 9. □

In the sequel, we write $\mathrm{dom}(\mu)$, for $\mu = (h, B, k)$, as shorthand for $\mathrm{dom}(h) \cup \cup_{i \in \mathbb{P}} \mathrm{dom}(B(i))$, and $\mathrm{dom}(\mu|_i)$ as shorthand for $\mathrm{dom}(h) \cup \mathrm{dom}(B(i))$.

Lemma 11. *For all $n \in \mathbb{N}$ and s, μ such that $s, \mu, \mathbb{P} \models e \leadsto_{e'} e'' \blacktriangleleft P$, if $x \notin \mathrm{fv}(e, e', e'', P)$ then $\mathit{safe}_n(x := [e]_{e'}, s, \mu, J, ((e \leadsto_{e'} e'' \blacktriangleleft P) \wedge x = e''))$.*

Proof. By induction on n. The base case is trivial. For the induction step, we assume the lemma holds for n and show that it holds for $n+1$.

1. The command is not equal to skip, so this part holds vacuously.

2. Let μ_0, μ_J, μ_F such that $\mu_0 \in \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} \mu)$, *lock-complete*$(\mu_0)$, and either *locked*$(\mu_0)$ or $s, \mu_J, \mathbb{P} \models J$. We must show that $c, s, \mu_0 \not\twoheadrightarrow \lightning$. The only aborting step possible is via C-PRIM-A by way of P-LOAD-A. This requires

that $\hat{s}(e) \notin \mathrm{dom}(h \setminus\!\!\setminus \overline{B(\hat{s}(e'))})$. But $s, \mu, \mathbb{P} \models (e \rightsquigarrow_{e'} e'') \blacktriangleleft P$ by hypothesis, which means there exists μ_w such that $s, \mu_w, \mathbb{P} \models e \rightsquigarrow_{e'} e''$ and $\mathrm{dom}(\mu_w|_{\hat{s}(e')}) \subseteq \mathrm{dom}(\mu|_{\hat{s}(e')}) \subseteq \mathrm{dom}(\mu_0|_{\hat{s}(e')})$. But $\hat{s}(e) \in \mathrm{dom}(\mu_w|_{\hat{s}(e')})$, and so

$$\hat{s}(e) \in \mathrm{dom}(h \setminus\!\!\setminus \overline{B(\hat{s}(e'))}).$$

Hence, the command cannot abort.

3. Let $\mu_0, \mu_J, \mu_F, c', s', \mu_1$ such that $\mu_0 \in \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} \mu)$, *lock-complete*$(\mu_0)$, either *locked*$(\mu_0)$ or $s, \mu_J, \mathbb{P} \models J$, and $c, s, \mu_0 \to c', s', \mu_1$. We must show μ'_J, μ'_F, μ' such that $\mu_1 \in \mu'_J \mathbin{\widetilde{*}} (\mu'_F \mathbin{\widetilde{\#}} \mu')$, $\mu'_F \preceq \mu_F$, either *locked*(μ_1) or $s', \mu'_J, \mathbb{P} \models J$, and $\mathit{safe}_n(c', s', \mu', J, (e \rightsquigarrow_{e'} e'' \blacktriangleleft P) \wedge x = e'')$. The evaluation step is either by C-TAU or C-PRIM by way of P-LOAD.

   In the case of C-TAU, $c' = (x := [e]_{e'})$, $s' = s$ and $\mu_1 \preceq \mu_0$. By Lemma 7 there exists μ'_J, μ'_F, μ' such that $\mu'_J \preceq \mu_J$, $\mu'_F \preceq \mu_F$ and $\mu' \preceq \mu$. By Lemma 2, $s, \mu', \mathbb{P} \models (e \rightsquigarrow_{e'} e'') \blacktriangleleft P$ and $s, \mu'_J, \mathbb{P} \models J$ if not *locked*(μ_1). It follows from the inductive hypothesis that $\mathit{safe}_n(c', s', \mu', J, (e \rightsquigarrow_{e'} e'' \blacktriangleleft P) \wedge x = e'')$.

   In the case of C-PRIM and P-LOAD, $c' = \mathsf{skip}$, $s' = s[x \leftarrow v]$ and $\mu_1 = \mu_0$, assuming $\mu_0 = (h_0, B_0, k_0)$ and $(h_0 \setminus\!\!\setminus \overline{B_0(\hat{s}(e'))})(\hat{s}(e)) = v$. Let $\mu'_J = \mu_J$, $\mu'_F = \mu_F$ and $\mu' = \mu$. Then $\mu_1 \in \mu'_J \mathbin{\widetilde{*}} (\mu'_F \mathbin{\widetilde{\#}} \mu')$ by assumption, $\mu'_F \preceq \mu_F$ by reflexivity, $s', \mu'_J, \mathbb{P} \models J$ if not *locked*(μ_1) because $x \notin \mathrm{fv}(J)$ and by Lemmas 2 and 3. Finally, $\mathit{safe}_n(\mathsf{skip}, s', \mu', J, ((e \rightsquigarrow_{e'} e'') \blacktriangleleft P \wedge x = e''))$ follows from Lemma 9 because $s', \mu', \mathbb{P} = s[x \leftarrow v], \mu, \mathbb{P} \models (e \rightsquigarrow_{e'} e'' \blacktriangleleft P) \wedge x = e''$, which itself is because $s, \mu, \mathbb{P} \models e \rightsquigarrow_{e'} e'' \blacktriangleleft P$ with $x \notin \mathrm{fv}(e, e', e'', P)$.

□

Lemma 12. $J \models \{x \rightsquigarrow_{e'} e'' \blacktriangleleft P\}\ \ x := [e]_{e'}\ \ \{(x \rightsquigarrow_{e'} e'' \blacktriangleleft P) \wedge x = e''\}$ *if* $x \notin \mathrm{fv}(e, e', e'', P)$.

Proof. Immediate from Lemmas 8 and 11. □

Lemma 13. *For all $n \in \mathbb{N}$ and s, μ with $s, \mu, \mathbb{P} \models e \rightsquigarrow_{e'} e'' \blacktriangleleft P$, it is the case that $\mathit{safe}_n([e] := f_{e'}, s, \mu, J, (e \rightsquigarrow_{e'} e'' \blacktriangleleft P) \lhd e \rightsquigarrow_{e'} f)$.*

Proof. By induction on n. The base case is trivial. For the induction step, we assume the lemma holds for n and show that it holds for $n+1$.

1. The command is not equal to skip, so this part holds vacuously.

2. Let μ_0, μ_J, μ_F such that $\mu_0 \in \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} \mu)$, $\mathit{lock\text{-}complete}(\mu_0)$, and either $\mathit{locked}(\mu_0)$ or $s, \mu_J, \mathbb{P} \models J$. We must show that $c, s, \mu_0 \nrightarrow \lightning$. The only aborting step possible is via C-PRIM-A by way of P-STORE-A. This requires that $\hat{s}(e) \notin \mathrm{dom}(h \mathbin{\backslash\!\backslash} \overline{B(\hat{s}(e'))})$. But $s, \mu, \mathbb{P} \models (e \rightsquigarrow_{e'} e'') \blacktriangleleft P$, which means there exists μ_w such that $s, \mu_w, \mathbb{P} \models e \rightsquigarrow_{e'} e''$ and $\mathrm{dom}(\mu_w|_{\hat{s}(e')}) \subseteq \mathrm{dom}(\mu|_{\hat{s}(e')}) \subseteq \mathrm{dom}(\mu_0|_{\hat{s}(e')})$. But $\hat{s}(e) \in \mathrm{dom}(\mu_w|_{\hat{s}(e')})$, and so $\hat{s}(e) \in \mathrm{dom}(h \mathbin{\backslash\!\backslash} \overline{B(\hat{s}(e'))})$. Hence, the command cannot abort.

3. Let $\mu_0, \mu_J, \mu_F, c', s', \mu_1$ such that $\mu_0 \in \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} \mu)$, $\mathit{lock\text{-}complete}(\mu_0)$, either $\mathit{locked}(\mu_0)$ or $s, \mu_J, \mathbb{P} \models J$, and $c, s, \mu_0 \to c', s', \mu_1$. We must show μ'_J, μ'_F, μ' such that $\mu_1 \in \mu'_J \mathbin{\widetilde{*}} (\mu'_F \mathbin{\widetilde{\#}} \mu')$, $\mu'_F \preceq \mu_F$, either $\mathit{locked}(\mu_1)$ or $s', \mu'_J, \mathbb{P} \models J$, and $\mathit{safe}_n(c', s', \mu', J, (e \rightsquigarrow_{e'} e'' \lhd P) \lhd e \rightsquigarrow_{e'} f)$. The evaluation step is either by C-TAU or C-PRIM by way of P-STORE.

   In the case of C-TAU, $c' = c$, $s' = s$ and $\mu_1 \preceq \mu_0$. By Lemma 7 there exists μ'_J, μ'_F, μ' such that $\mu'_J \preceq \mu_J$, $\mu'_F \preceq \mu_F$ and $\mu' \preceq \mu$. By Lemma 2, $s, \mu', \mathbb{P} \models e \rightsquigarrow_{e'} e'' \blacktriangleleft P$ and $s, \mu'_J, \mathbb{P} \models J$ if not $\mathit{locked}(\mu_1)$. It follows from the inductive hypothesis that $\mathit{safe}_n(c, s, \mu', J, (e \rightsquigarrow_{e'} e'' \blacktriangleleft P) \lhd e \rightsquigarrow_{e'} f)$.

   In the case of C-PRIM and P-STORE, $c' =$ skip, $s' = s$ and let $\mu_1 = \mu_0 \mathbin{\widetilde{\lhd}} \mu_w$, where μ_w models the new write, defined as follows

$$\mu_w =_{df} (\emptyset, \mathcal{E}\left[\hat{s}(e') \leftarrow [(\hat{s}(e), \hat{s}(f))]\right], \emptyset).$$

From $\mu_0 \in (\mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} \mu))$ we have:

$$\begin{array}{ll} & \mu_1 \\ = & \{\text{definition}\} \\ & \mu_0 \mathbin{\widetilde{\lhd}} \mu_w \\ \in & \{\text{assumption } \mu_0 \in \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} \mu)\} \\ & (\mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} \mu)) \mathbin{\widetilde{\lhd}} \mu_w \\ \subseteq & \{\text{algebra of separation functions}\} \\ & \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} (\mu \mathbin{\widetilde{\lhd}} \mu_w)) \end{array}$$

Finally, we have $\mathit{safe}_n(\mathsf{skip}, s, \mu \mathbin{\widetilde{\lhd}} \mu_w, J, (e \leadsto_{e'} e'' \blacktriangleleft P) \lhd e \leadsto_{e'} f)$ by way of Lemma 9 because $s, \mu \mathbin{\widetilde{\lhd}} \mu_w, \mathbb{P} \models (e \leadsto_{e'} e'' \blacktriangleleft P) \lhd e \leadsto_{e'} f$, which itself is because $s, \mu, \mathbb{P} \models e \leadsto_{e'} e'' \blacktriangleleft P$ and $s, \mu_w, \mathbb{P} \models e \leadsto_{e'} f$

□

Lemma 14. $J \models \{e \leadsto_{e'} e'' \blacktriangleleft P\}\ [e] := f_{e'}\ \{(e \leadsto_{e'} e'' \blacktriangleleft P) \lhd e \leadsto_{e'} f\}$.

Proof. Immediate from Lemmas 8 and 13. □

Lemma 15. *For all $n \in \mathbb{N}$ and s, μ such that $s, \mu, \mathbb{P} \models$ **emp** it is the case that $\mathit{safe}_n(\mathsf{lock}_e, s, \mu, J, \mathbf{lock}_e)$.*

Proof. By induction on n. The base case is trivial. For the induction step, we assume the lemma holds for n and show that it holds for $n + 1$.

1. The command is not equal to skip, so this part holds vacuously.

2. There are no aborting steps possible from command lock_e.

3. Let $\mu_0, \mu_J, \mu_F, c', s', \mu_1$ such that $\mu_0 \in \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} \mu)$, *lock-complete*$(\mu_0)$, either *locked*$(\mu_0)$ or $s, \mu_J, \mathbb{P} \models J$, and $c, s, \mu_0 \to c', s', \mu_1$. We must show μ'_J, μ'_F, μ' such that $\mu_1 \in \mu'_J \mathbin{\widetilde{*}} (\mu'_F \mathbin{\widetilde{\#}} \mu')$, $\mu'_F \preceq \mu_F$, either *locked*(μ_1) or

$s', \mu'_J, \mathbb{P} \models J$, and $\mathit{safe}_n(c', s', \mu', J, \mathbf{lock}_e)$. The evaluation step is either by C-TAU or C-PRIM by way of P-LOCK.

In the case of C-TAU, $c' = c$, $s' = s$ and $\mu_1 \preceq \mu_0$. By Lemma 7 there exists μ'_J, μ'_F, μ' such that $\mu'_J \preceq \mu_J$, $\mu'_F \preceq \mu_F$ and $\mu' \preceq \mu$. By Lemma 2, $s, \mu', \mathbb{P} \models \mathbf{emp}$ and $s, \mu'_J, \mathbb{P} \models J$ if not $\mathit{locked}(\mu_1)$. It follows from the inductive hypothesis that $\mathit{safe}_n(c', s', \mu', J, \mathbf{lock}_e)$.

In the case of C-PRIM and P-LOCK, $c' = \mathsf{skip}$, $s' = s$ and $\mu_1 = (h_0, B_0, \mathbb{P} \setminus \{\hat{s}(e)\})$, assuming $\mu_0 = (h_0, B_0, K_0)$. By assumption, $K_0 = \emptyset$, and so from $\mu_0 \in \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} \mu)$, we have that $\mu = (h, B, \emptyset)$ for some h, B. Let $\mu' = (h, B, \mathbb{P} \setminus \{\hat{s}(e)\})$. Then from $\mu_0 \in \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} \mu)$ and $K_0 = \emptyset$, we also have $\mu_1 \in \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} \mu')$. Finally, $\mathit{safe}_n(\mathsf{skip}, s, \mu', J, \mathbf{lock}_e)$ follows from Lemma 9 because $s, \mu', \mathbb{P} \models \mathbf{lock}_e$, which itself is because $s, \mu, \mathbb{P} \models \mathbf{emp}$.

□

Lemma 16. $J \models \{\mathbf{emp}\}\ \mathsf{lock}_e\ \{\mathbf{lock}_e\}$.

Proof. Immediate from Lemmas 8 and 15. □

Lemma 17. *For all $n \in \mathbb{N}$ and s, μ such that $s, \mu, \mathbb{P} \models \mathbf{lock}_e$ it is the case that $\mathit{safe}_n(\mathsf{unlock}_e, s, \mu, J, \mathbf{emp})$.*

Proof. By induction on n. The base case is trivial. For the induction step, we assume the lemma holds for n and show that it holds for $n + 1$.

1. The command is not equal to skip, so this part holds vacuously.

2. There are no aborting steps possible from command unlock_e.

3. Let $\mu_0, \mu_J, \mu_F, c', s', \mu_1$ such that $\mu_0 \in \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} \mu)$, $\mathit{lock\text{-}complete}(\mu_0)$, either $\mathit{locked}(\mu_0)$ or $s, \mu_J, \mathbb{P} \models J$, and $c, s, \mu_0 \to c', s', \mu_1$. We must show μ'_J, μ'_F, μ' such that $\mu_1 \in \mu'_J \mathbin{\widetilde{*}} (\mu'_F \mathbin{\widetilde{\#}} \mu')$, $\mu'_F \preceq \mu_F$, either $\mathit{locked}(\mu_1)$ or

$s', \mu'_J, \mathbb{P} \models J$, and $\mathit{safe}_n(c', s', \mu', J, \mathbf{emp})$. The evaluation step is either by C-TAU or C-PRIM by way of P-UNLOCK.

In the case of C-TAU, $c' = c$, $s' = s$ and $\mu_1 \preceq \mu_0$. By Lemma 7 there exists μ'_J, μ'_F, μ' such that $\mu'_J \preceq \mu_J$, $\mu'_F \preceq \mu_F$ and $\mu' \preceq \mu$. By Lemma 2, $s, \mu', \mathbb{P} \models \mathbf{lock}_e$ and $s, \mu'_J, \mathbb{P} \models J$ if not $\mathit{locked}(\mu_1)$. It follows from the inductive hypothesis that $\mathit{safe}_n(c', s', \mu', J, \mathbf{emp})$.

In the case of C-PRIM and P-UNLOCK, $c' = \mathsf{skip}$, $s' = s$ and $\mu_1 = (h_0, B_0, \emptyset)$, assuming $\mu_0 = (h_0, B_0, K_0)$. By assumption, $K_0 = \mathbb{P} \setminus \{\hat{s}(e)\}$, and so from $\mu_0 \in \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} \mu)$, we have that $\mu = (h, B, \mathbb{P} \setminus \{\hat{s}(e)\})$ for some h, B. Let $\mu' = (h, B, \emptyset)$. Then from $\mu_0 \in \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} \mu)$ and $K_0 = \mathbb{P} \setminus \{\hat{s}(e)\}$, we also have $\mu_1 \in \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} \mu')$. Finally, $\mathit{safe}_n(\mathsf{skip}, s, \mu', J, \mathbf{emp})$ follows from Lemma 9 because $s, \mu', \mathbb{P} \models \mathbf{emp}$, which itself is because $s, \mu, \mathbb{P} \models \mathbf{lock}_e$.

□

Lemma 18. *$J \models \{\mathbf{lock}_e\}\ \mathsf{unlock}_e\ \{\mathbf{emp}\}$.*

Proof. Immediate from Lemmas 8 and 17. □

A.2.2 Soundness of the Inference Rules

This section presents soundness lemmas for some, but not all, of the inference rules described in Section 4.5.1. That is, for some inference rules

$$\frac{J' \vdash \{P'\}\ c'\ \{Q'\}}{J \vdash \{P\}\ c\ \{Q\}}$$

we present lemmas below which assert that

$$J' \models \{P'\}\ c'\ \{Q'\} \text{ only if } J \models \{P\}\ c\ \{Q\}.$$

For a set of memory systems S, we write $\mathit{safe}_n(c, s, S, J, Q)$ and $s, S, \Gamma \models Q$ as shorthand for the universal quantifications

$$\forall \mu \in S : \mathit{safe}_n(c, s, \mu, J, Q) \quad \text{and} \quad \forall \mu \in S : s, \mu, \Gamma \models Q.$$

The following two lemmas sketch soundness proofs for a spatiotemporal frame rule. Note, however, that the spatiotemporal conjunction is not actually defined in the assertion language described in this dissertation, nor is there a spatiotemporal frame rule. But the separating conjunctions that are described are refinements of this hypothetical spatiotemporal conjunction, and so it may be interesting to consider its frame rule.

Lemma 19. *If* $\mathit{safe}_n(c, s, \mu, J, Q)$*,* $\mathrm{fv}(R) \cap \mathrm{mod}(c) = \emptyset$*,* $\mathsf{def}(\mu_R \mathbin{\widetilde{\#}} \mu)$*, and* $s, \mu_R, \mathbb{P} \models R$*, then* $\mathit{safe}_n(c, s, \mu_R \mathbin{\widetilde{\#}} \mu, J, R \# Q)$*.*

Sketch. By induction on n. The base case is trivial. For the induction step, we assume the lemma holds for n and show that it holds for $n+1$. That is, we assume, for any $c, s, \mu, \mu_R, J, Q, R$, that whenever $\mathit{safe}_n(c, s, \mu, J, Q)$, $\mathrm{fv}(R) \cap \mathrm{mod}(c) = \emptyset$, $\mathsf{def}(\mu_R \mathbin{\widetilde{\#}} \mu)$, and $s, \mu_R, \mathbb{P} \models R$, it is also the case that $\mathit{safe}_n(c, s, \mu_R \mathbin{\widetilde{\#}} \mu, J, R \# Q)$ holds. We also assume that $\mathit{safe}_{n+1}(c, s, \mu, J, Q)$, $\mathrm{fv}(R) \cap \mathrm{mod}(c) = \emptyset$, $\mathsf{def}(\mu_R \mathbin{\widetilde{\#}} \mu)$, and $s, \mu_R, \mathbb{P} \models R$, and show that $\mathit{safe}_{n+1}(c, s, \mu_R \mathbin{\widetilde{\#}} \mu, J, R \# Q)$. To show this, we must establish three conditions.

1. Suppose $c = \mathsf{skip}$. By assumption, $s, \mu_R, \mathbb{P} \models R$, $\mathsf{def}(\mu_R \mathbin{\widetilde{\#}} \mu)$, and if $c = \mathsf{skip}$ then $s, \mu, \mathbb{P} \models Q$, and so $s, \mu, \mathbb{P} \models Q$. Hence $s, (\mu_R \mathbin{\widetilde{\#}} \mu), \mathbb{P} \models R \# Q$.

2. Let μ_0, μ_J, μ_F such that $\mu_0 \in \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} (\mu_R \mathbin{\widetilde{\#}} \mu))$, and either $\mathit{locked}(\mu_0)$ or $s, \mu_J, \mathbb{P} \models J$. We must show that $c, s, \mu_0 \not\twoheadrightarrow \lightning$. But

$$\mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} (\mu_R \mathbin{\widetilde{\#}} \mu)) \subseteq \mu_J \mathbin{\widetilde{*}} ((\mu_F \mathbin{\widetilde{\#}} \mu_R) \mathbin{\widetilde{\#}} \mu),$$

and so by part 2 of assumption $\mathit{safe}_{n+1}(c, s, \mu, J, Q)$, instantiating with (an arbitrary element of) $\mu_F \mathbin{\widetilde{\#}} \mu_R$, the result holds.

3. Let $\mu_0, \mu_J, \mu_F, \mu_1, c', s'$ such that $\mu_0 \in \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} (\mu_R \mathbin{\widetilde{\#}} \mu))$, either $\mathit{locked}(\mu_0)$ or $s, \mu_J, \mathbb{P} \models J$, and $c, s, \mu_0 \to c', s', \mu_1$. We must exhibit μ'_J, μ'_F, μ' such that:

   (a) $\mu_1 \in \mu'_J \mathbin{\widetilde{*}} (\mu'_F \mathbin{\widetilde{\#}} \mu')$,

   (b) $\mu'_F \preceq \mu_F$,

   (c) either $\mathit{locked}(\mu_1)$ or $s', \mu'_J, \mathbb{P} \models J$, and

   (d) $\mathit{safe}_n(c', s', \mu', J, R \# Q)$.

   We instantiate part 3 of assumption $\mathit{safe}_{n+1}(c, s, \mu, J, Q)$ with μ_J, (an arbitrary element of) $\mu_F \mathbin{\widetilde{\#}} \mu_R$, μ_1, c' and s', which gives us μ'_J, μ'_{FR}, μ' such that:

   (a) $\mu_1 \in \mu'_J \mathbin{\widetilde{*}} (\mu'_{FR} \mathbin{\widetilde{\#}} \mu')$

   (b) $\mu'_{FR} \preceq \mu_F \mathbin{\widetilde{\#}} \mu_R$

   (c) either $\mathit{locked}(\mu_1)$ or $s', \mu'_J, \mathbb{P} \models J$, and

   (d) $\mathit{safe}_n(c', s, \mu', J, Q)$.

   By Lemma 7 and $\mu'_{FR} \preceq \mu_F \mathbin{\widetilde{\#}} \mu_R$, there exists μ'_F, μ'_R such that $\mu'_F \preceq \mu_F$, $\mu'_R \preceq \mu_R$ and $\mu'_{FR} \in \mu'_F \mathbin{\widetilde{\#}} \mu'_R$.

   From $\mu'_R \preceq \mu_R$ and $s, \mu_R, \mathbb{P} \models R$—as well as the assumption $\mathrm{fv}(R) \cap \mathrm{mod}(c) = \emptyset$ and Lemmas 2 and 3—it follows that $s', \mu'_R, \mathbb{P} \models R$.

   Next, we wish to apply the inductive hypothesis to $\mathit{safe}_n(c', s', \mu', J, Q)$ to show that $\mathit{safe}_n(c', s', \mu'_R \# \mu', J, R \# Q)$. This follows from the observations that $\mathrm{fv}(R) \cap \mathrm{mod}(c') = \emptyset$ (because $\mathrm{mod}(c') \subseteq \mathrm{mod}(c)$ from Lemma 2) and $\mathsf{def}(\mu'_R \mathbin{\widetilde{\#}} \mu')$.

Returning to our original task, we exhibit μ'_J, μ'_F and $\mu'_R \mathbin{\widetilde{\#}} \mu'$. By associativity and monotonicity it follows that

$$\mu_1 \in \mu'_J \mathbin{\widetilde{*}} (\mu'_F \mathbin{\widetilde{\#}} (\mu'_R \mathbin{\widetilde{\#}} \mu')).$$

We have already showed that $\mu'_F \preceq \mu_F$. Either $\mathit{locked}(\mu_1)$ or $s', \sigma'_J, \mathbb{P} \models J$ by assumption. Finally $\mathit{safe}_n(c', s', \mu'_R \mathbin{\widetilde{\#}} \mu', J, R \mathbin{\#} Q)$ by the inductive hypothesis above.

□

Lemma 20. *If* $J \models \{P\}\ c\ \{Q\}$ *and it is the case that* $\mathrm{fv}(R) \cap \mathrm{mod}(c) = \emptyset$ *then* $J \models \{R \mathbin{\#} P\}\ c\ \{R \mathbin{\#} Q\}$.

Sketch. Let $s, \mu, \mathbb{P} \models R \mathbin{\#} P$ and $n \in \mathbb{N}$. We must show $\mathit{safe}_n(c, s, \mu, J, R \mathbin{\#} Q)$. From $s, \mu, \mathbb{P} \models R \mathbin{\#} P$, there exists μ_R, μ_P with $\mu \in \mu_R \mathbin{\widetilde{\#}} \mu_P$ such that $s, \mu_R, \mathbb{P} \models R$ and $s, \mu_P, \mathbb{P} \models P$. By assumption, $\mathit{safe}_n(c, s, \mu_P, J, Q)$. The result follows from Lemma 19. □

A proof sketch of the safety of the spatial frame rule relies on the following unproven conjecture.

Conjecture 1. *Let* $\mu_a \in \mu_b \mathbin{\widetilde{*}} \mu_c$*, and suppose* $c, s, \mu_a \to c', s', \mu'_a$*. If* $\mu'_a \in \mu'_b \mathbin{\widetilde{\#}} \mu' c$ *with* $\mu'_b \preceq \mu_b$*, then* $\mu'_b \mathbin{\widetilde{\#}} \mu' c = \mu'_b \mathbin{\widetilde{*}} \mu' c$.

Lemma 21. *If* $\mathit{safe}_n(c, s, \mu, J, Q)$*,* $\mathrm{fv}(R) \cap \mathrm{mod}(c) = \emptyset$*,* $\mathsf{def}(\mu_R \mathbin{\widetilde{*}} \mu)$*, and* $s, \mu_R, \mathbb{P} \models R$*, then* $\mathit{safe}_n(c, s, \mu_R \mathbin{\widetilde{*}} \mu, J, R * Q)$.

Sketch. By induction on n. The base case is trivial. For the induction step, we assume the lemma holds for n and show that it holds for $n+1$. That is, we assume, for any $c, s, \mu, \mu_R, J, Q, R$, that whenever $\mathit{safe}_n(c, s, \mu, J, Q)$, $\mathrm{fv}(R) \cap \mathrm{mod}(c) = \emptyset$, $\mathsf{def}(\mu_R \mathbin{\widetilde{*}} \mu)$, and $s, \mu_R, \mathbb{P} \models R$, it is also the case that $\mathit{safe}_n(c, s, \mu_R \mathbin{\widetilde{*}} \mu, J, R * Q)$

holds as well. We additionally assume the antecedent, which is that $\mathit{safe}_{n+1}(c, s, \mu, J, Q)$, $\mathrm{fv}(R) \cap \mathrm{mod}(c) = \emptyset$, $\mathsf{def}(\mu_R \mathbin{\widetilde{*}} \mu)$, and $s, \mu_R, \mathbb{P} \models R$, and show the consequent, which is that $\mathit{safe}_{n+1}(c, s, \mu_R \mathbin{\widetilde{*}} \mu, J, R * Q)$. To show this, we must establish three conditions.

1. Suppose $c = \mathsf{skip}$. By assumption, $s, \mu_R, \mathbb{P} \models R$, $\mathsf{def}(\mu_R \mathbin{\widetilde{*}} \mu)$, and if $c = \mathsf{skip}$ then $s, \mu, \mathbb{P} \models Q$, and so indeed $s, \mu, \mathbb{P} \models Q$. Hence $s, \mu_R \mathbin{\widetilde{*}} \mu, \mathbb{P} \models R * Q$.

2. Let μ_0, μ_J, μ_F such that $\mu_0 \in \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} (\mu_R \mathbin{\widetilde{*}} \mu))$, and either $\mathit{locked}(\mu_0)$ or $s, \mu_J, \mathbb{P} \models J$. We must show that $c, s, \mu_0 \not\rightarrow \lightning$. But

$$\begin{aligned} \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} (\mu_R \mathbin{\widetilde{*}} \mu)) &\subseteq \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} (\mu_R \mathbin{\widetilde{\#}} \mu)) \\ &= \mu_J \mathbin{\widetilde{*}} ((\mu_F \mathbin{\widetilde{\#}} \mu_R) \mathbin{\widetilde{\#}} \mu) \end{aligned}$$

   and so by part 2 of assumption $\mathit{safe}_{n+1}(c, s, \mu, J, Q)$, instantiating with (an arbitrary element of) $\mu_F \mathbin{\widetilde{\#}} \mu_R$, the result holds.

3. Let $\mu_0, \mu_J, \mu_F, \mu_1, c', s'$ such that $\mu_0 \in \mu_J \mathbin{\widetilde{*}} (\mu_F \mathbin{\widetilde{\#}} (\mu_R \mathbin{\widetilde{*}} \mu))$, either $\mathit{locked}(\mu_0)$ or $s, \mu_J, \mathbb{P} \models J$, and $c, s, \mu_0 \rightarrow c', s', \mu_1$. We must exhibit μ'_J, μ'_F, μ' such that:

   (a) $\mu_1 \in \mu'_J \mathbin{\widetilde{*}} (\mu'_F \mathbin{\widetilde{\#}} \mu')$,

   (b) $\mu'_F \preceq \mu_F$,

   (c) either $\mathit{locked}(\mu_1)$ or $s', \mu'_J, \mathbb{P} \models J$, and

   (d) $\mathit{safe}_n(c', s', \mu', J, R * Q)$.

   We first instantiate part 3 of assumption $\mathit{safe}_{n+1}(c, s, \mu, J, Q)$ with μ_J, (an arbitrary element of) $\mu_F \mathbin{\widetilde{\#}} \mu_R$, μ_1, c' and s', which gives us μ'_J, μ'_{FR}, μ' such that:

   (a) $\mu_1 \in \mu'_J \mathbin{\widetilde{*}} (\mu'_{FR} \mathbin{\widetilde{\#}} \mu')$

   (b) $\mu'_{FR} \preceq \mu_F \mathbin{\widetilde{\#}} \mu_R$

(c) either $\mathit{locked}(\mu_1)$ or $s', \mu'_J, \mathbb{P} \models J$, and

(d) $\mathit{safe}_n(c', s', \mu', J, Q)$.

By Lemma 7 and $\mu'_{FR} \preceq \mu_F \mathbin{\widetilde{\#}} \mu_R$, there exists μ'_F, μ'_R such that $\mu'_F \preceq \mu_F$, $\mu'_R \preceq \mu_R$ and $\mu'_{FR} \in \mu'_F \mathbin{\widetilde{\#}} \mu'_R$.

From $\mu'_R \preceq \mu_R$ and $s, \mu_R, \mathbb{P} \models R$—as well as the assumption $\mathrm{fv}(R) \cap \mathrm{mod}(c) = \emptyset$ and Lemmas 2 and 3—it follows that $s', \mu'_R, \mathbb{P} \models R$.

Next, we wish to apply the inductive hypothesis to $\mathit{safe}_n(c', s', \mu', J, Q)$ to show that $\mathit{safe}_n(c', s', \mu'_R \mathbin{\widetilde{*}} \mu', J, R * Q)$. This follows from the observations that $\mathrm{fv}(R) \cap \mathrm{mod}(c') = \emptyset$ (because $\mathrm{mod}(c') \subseteq \mathrm{mod}(c)$ from Lemma 2) and $\mathsf{def}(\mu'_R \mathbin{\widetilde{*}} \mu')$.

Returning to our original task, we exhibit μ'_J, μ'_F and $\mu'_R \mathbin{\widetilde{*}} \mu'$. By Conjecture 1 it follows that

$$\mu_1 \in \mu'_J \mathbin{\widetilde{*}} (\mu'_F \mathbin{\widetilde{\#}} (\mu'_R \mathbin{\widetilde{*}} \mu')).$$

We have already showed that $\mu'_F \preceq \mu_F$. Either $\mathit{locked}(\mu_1)$ or $s', \sigma'_J, \mathbb{P} \models J$ by assumption. Finally $\mathit{safe}_n(c', s', \mu'_R \mathbin{\widetilde{*}} \mu', J, R * Q)$ by the inductive hypothesis above.

□

Proof of Theorem 1. By induction on the structure of an arbitrary derivation of $\overline{J \vdash \{P\}\ c\ \{Q\}}$. The base cases follow from Lemmas 10, 12, 14, 16, 18, etc. The inductive cases follow from Lemma 20, etc.

□

A.3 Summary of Notation

Mathematical function and predicate symbols are written in an *italic* face. Programing language primitives are written in a sans-serif face. Atomic formulas are written in a **bold serif** face.

The symbols that are typically used to denote particular objects are listed in Figure A.1.

Object	Symbols	Components
Identifiers (aka variables)	x, y, z, t, u, v	
Processor identifiers	i, j, k	
Memory addresses (aka locations)	ℓ	
Lists (generally)	l, m, n	
Lists (as write buffers)	b	
Buffer arrays	B	
Stacks	s	
Heaps	h	
Blocking sets	K	
Buffer-completeness sets	Γ	
Memory systems	μ	(h, b) or (h, B)
Generalized memory systems	ν	(μ, γ) or (μ, Γ)
States	σ	(s, μ)
Models	$\mathcal{M}$	(s, ν)
Expressions	e, f	
Primitive commands	p	
Commands	c	
Assertions (generally)	P, Q, R	
Assertions (as invariants)	I, J	

Figure A.1: Symbols and the objects they denote

Bibliography

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] J. Alglave, A. C. J. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli. The semantics of Power and ARM multiprocessor machine code. In Petersen and Chakravarty [41], pages 13–24.

[3] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

[4] J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory Safety for Systems-level Code. In G. Gopalakrishnan and S. Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*. Springer, 2011.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[6] R. Bornat. Proving pointer programs in hoare logic. In R. C. Backhouse and J. N. Oliveira, editors, *MPC*, volume 1837 of *Lecture Notes in Computer Science*, pages 102–126. Springer, 2000.

[7] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *POPL*, pages 259–270. ACM, 2005.

[8] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *SAS*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

[9] S. Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.

[10] S. Brookes. A revisionist history of concurrent separation logic. *Electronic Notes in Theoretical Computer Science*, 276(0):5 – 28, 2011. Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII).

[11] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, pages 366–378. IEEE Computer Society, 2007.

[12] K. M. Chandy and J. Misra. *Parallel program design: a foundation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[13] N. Chong and S. Ishtiaq. Reasoning about the ARM weakly consistent memory model. In E. D. Berger and B. Chen, editors, *MSPC*, pages 16–19. ACM, 2008.

[14] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[15] E. Cohen and B. Schirmer. From total store order to sequential consistency: A practical reduction theorem. In M. Kaufmann and L. C. Paulson, editors, *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 403–418. Springer, 2010.

[16] R. Dockins, A. Hobor, and A. W. Appel. A fresh look at separation algebras and share accounting. In Z. Hu, editor, *APLAS*, volume 5904 of *Lecture Notes in Computer Science*, pages 161–177. Springer, 2009.

[17] M. Dodds, X. Feng, M. J. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In G. Castagna, editor, *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2009.

[18] R. Ferreira, X. Feng, and Z. Shao. Parameterized memory models and concurrent separation logic. In A. D. Gordon, editor, *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 267–286. Springer, 2010.

[19] D. Galmiche and D. Larchey-Wendling. Expressivity properties of boolean BI through relational models. In S. Arun-Kumar and N. Garg, editors, *FSTTCS*, volume 4337 of *Lecture Notes in Computer Science*, pages 357–368. Springer, 2006.

[20] A. Gotsman, J. Berdine, and B. Cook. Precision and the conjunction rule in concurrent separation logic. *Electr. Notes Theor. Comput. Sci.*, 276:171–190, 2011.

[21] L. Higham, J. Kawash, and N. Verwaal. Weak memory consistency models part I: Definitions and comparisons. Technical Report 98/612/03, Department of Computer Science, The University of Calgary, 1998.

[22] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[23] C. A. R. Hoare, A. Hussain, B. Möller, P. W. O'Hearn, R. L. Petersen, and G. Struth. On locality and the exchange law for concurrent processes. In J.-P. Katoen and B. König, editors, *CONCUR*, volume 6901 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2011.

[24] C. A. R. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent Kleene algebra. In M. Bravetti and G. Zavattaro, editors, *CONCUR*, volume 5710 of *Lecture Notes in Computer Science*, pages 399–414. Springer, 2009.

[25] C. A. R. Hoare, B. Möller, G. Struth, and I. Wehrman. Foundations of concurrent kleene algebra. In R. Berghammer, A. Jaoua, and B. Möller, editors, *RelMiCS*, volume 5827 of *Lecture Notes in Computer Science*, pages 166–186. Springer, 2009.

[26] C. A. R. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent kleene algebra and its foundations. *J. Log. Algebr. Program.*, 80(6):266–296, 2011.

[27] T. Hoare and J. Wickerson. Unifying models of data flow. In M. Broy, C. Leuxner, and T. Hoare, editors, *Software and Systems Safety - Specification and Verification*, volume 30 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 211–230. IOS Press, 2011.

[28] C. B. Jones. *Development methods for computer programs including a notion of interference.* PhD thesis, Oxford University, 1981.

[29] M. Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Trans. Software Eng.*, 23(4):203–213, 1997.

[30] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[31] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[32] J. S. Moore and G. Porter. An executable formal java virtual machine thread model. In *Java Virtual Machine Research and Technology Symposium*, pages 91–104. USENIX, 2001.

[33] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[34] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[35] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[36] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[37] S. Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In T. D'Hondt, editor, *ECOOP*, volume 6183 of *Lecture Notes in Computer Science*, pages 478–503. Springer, 2010.

[38] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer, 2009.

[39] S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[40] M. J. Parkinson. Local reasoning for Java. Technical report, University of Cambridge, 2005. Technical Report UCAM-CL-TR-64.

[41] L. Petersen and M. M. T. Chakravarty, editors. *Proceedings of the POPL 2009 Workshop on Declarative Aspects of Multicore Programming, DAMP 2009, Savannah, GA, USA, January 20, 2009*. ACM, 2009.

[42] G. L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[43] V. R. Pratt. On the composition of processes. In *POPL*, pages 213–223, 1982.

[44] V. R. Pratt. The pomset model of parallel processes: Unifying the temporal and the spatial. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 180–196. Springer, 1984.

[45] D. J. Pym, P. W. O'Hearn, and H. Yang. Possible worlds and resources: the semantics of BI. *Theor. Comput. Sci.*, 315(1):257–305, 2004.

[46] U. S. Reddy and J. C. Reynolds. Syntactic control of interference for separation logic. In J. Field and M. Hicks, editors, *POPL*, pages 323–336. ACM, 2012.

[47] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[48] T. Ridge. Operational reasoning for concurrent caml programs and weak memory models. In K. Schneider and J. Brandt, editors, *TPHOLs*, volume 4732 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2007.

[49] T. Ridge. A rely-guarantee proof system for x86-TSO. In G. T. Leavens, P. W. O'Hearn, and S. K. Rajamani, editors, *VSTTE*, volume 6217 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 2010.

[50] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding power multiprocessors. In M. W. Hall and D. A. Padua, editors, *PLDI*, pages 175–186. ACM, 2011.

[51] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In Z. Shao and B. C. Pierce, editors, *POPL*, pages 379–391. ACM, 2009.

[52] R. C. Steinke and G. J. Nutt. A unified theory of shared memory consistency. *J. ACM*, 51(5):800–849, 2004.

[53] V. Vafeiadis. Modular fine-grained concurrency verification. Technical report, University of Cambridge, 2008. Technical Report UCAM-CL-TR-726.

[54] V. Vafeiadis. Concurrent separation logic and operational semantics. In J. Ouaknine, editor, *27th Conference on the Mathematical Foundations of Programming Semantics*, 2011.

[55] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In L. Caires and V. T. Vasconcelos, editors, *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2007.

[56] I. Wehrman. Semantics and syntax of a weak-memory concurrent separation logic: Sequential fragment. Technical Report TR-12-08, University of Texas at Austin, Department of Computer Science, 2010.

[57] I. Wehrman and J. Berdine. A proposal for weak-memory local reasoning. In *Syntax and Semantics of Low-Level Languages*, 2011. Available at `http://www.cs.utexas.edu/~iwehrman/pub/lola.pdf`.

[58] I. Wehrman, C. A. R. Hoare, and P. W. O'Hearn. Graphical models of separation logic. *Inf. Process. Lett.*, 109(17):1001–1004, 2009.

[59] G. Winskel. Event structures. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets*, volume 255 of *Lecture Notes in Computer Science*, pages 325–392. Springer, 1986.

[60] H. Yang. *Local Reasoning for Stateful Programs.* PhD thesis, University of Illinois at Urbana-Champaign, 2001. Technical Report UIUCDCS-R-2001-2227.

[61] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In A. Gupta and S. Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 385–398. Springer, 2008.

[62] H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. In M. Nielsen and U. Engberg, editors, *FoSSaCS*, volume 2303 of *Lecture Notes in Computer Science*, pages 402–416. Springer, 2002.

Vita

Ian Wehrman was born in St. Louis, Missouri on June 7, 1980. He received his B.Sc. in Computer Science from Webster University in 2004 and his M.Sc. in Computer Science from Washington University in St. Louis in 2006. His master's thesis describes a fully automated approach to solving the word problem from automated deduction by using termination checkers of term-rewriting systems. Later in 2006 he entered the doctoral program in Computer Science at the University of Texas at Austin where he studied formal methods, concurrency and the theory of programming languages. He received a Ph.D in 2012 for a dissertation that describes a program logic for reasoning locally about the behavior of programs w.r.t. a weak, x86-like memory model.

Permanent Address: `ian@wehrman.org`

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.