Weak-Memory Local Reasoning

Ian Wehrman

Abstract

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

Existing program logics make the strong assumption that all threads agree on the value of shared memory at all times. This assumption is unsound, though, for shared-memory concurrent programs with race conditions, like concurrent data structures. Verification of these difficult programs must take into account the weaker models of memory provided by the architectures on which they execute.

In my dissertation project, I explicate a local reasoning principle for a weak memory model based on a formal specification of the x86 multiprocessor memory model. I demonstrate this principle with a new program logic for fine-grained concurrent C-like programs that incorporates ideas from separation logic and rely/guarantee. Notably, the logic may be applied soundly to programs with races, for which no general high-level verification techniques exist.

Last updated: Thursday 16th September, 2010 at 11:20.

1 Overview

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 racey 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 literature offers little insight into the problem of verifying concurrent data structures and other inherently racey 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.[1] Fortunately, the latter problem has been alleviated with recent safety specifications for the x86, Power and ARM memory models [26, 28]. 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.

In my dissertation project, I consider the concurrent verification problem for shared-memory programs with semantics based on the x86 multiprocessor memory model. My focus is on verification via proof with a new Hoare-style logic designed for C-like programs with load, store and fence instructions, pointers and pointer arithmetic, and dynamic memory management.

My main contribution is 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.

For my dissertation, I explore local reasoning in the context of an x86-like memory model by developing a logic which embodies such a principle. The sequential fragment of this logic is 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 will be based on the rely/guarantee calculus for fine-grained concurrency. Finally, I will use the logic to prove the specifications of a selection of concurrent programs, including some non-trivial concurrent data structures.

2 Motivation

To motivate study of a local reasoning princple 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 pseu-

[1] Or, perhaps, privately.

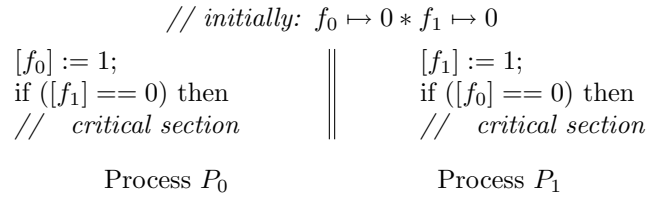

Figure 1: Dekker's algorithm.

docode program in Figure 1.[2]

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 the result of loading the address at other flag f_{1-i} is 0. (I assume for now that load and store are atomic.)

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 [20] 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 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

[2]Throughout, I write $[x]$ for the dereferencing of pointer variable x.

[3]This program does not, of course, preclude deadlock.

more 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) [28, 8]. 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 [26]. 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 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 would add fence instructions directly after the processes' store operations. I claim that 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.

3 Related Work

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 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 [19], Isabelle/HOL [22], or Coq [3]), is within the realm of feasibility for some programs and some experts. But, due to the generality of the logic and complexity of the semantics of programs under study, one expects such proofs to be exceptionally complex. And although experts would certainly 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 a general-purpose logic, both human reasoning (e.g., Unity [7]) and automation (e.g., model checking [9]) were facilitated by specialized logics.

The following sections discuss relevant work on program logics (Section 3.1) and on alternative techniques for weak-memory reasoning (Section 3.2).

3.1 Solutions to Related Problems

Hoare introduced the first program logic for an Algol-like language in his seminal 1969 paper [16]. Programs are specified with a pair of assertions, written in first-order logic, that describe pre- and post-execution system states. The logic has two flavors of specification: partial correctness, $\{P\}\, c\, \{Q\}$, in which nonterminating executions satisfy any specification; and total correctness, $\langle P\rangle\, c\, \langle Q\rangle$, in which termination is required to satisfy any specification.

The axioms and inference rules are mostly directed by the program syntax, making proof construction partially mechanical. Notable exceptions include the inference rule for loops, which requires invention of a suitable loop invariant, and the rule of consequence:

$$\frac{P' \Rightarrow P \quad \{P\}\, c\, \{Q\} \quad Q \Rightarrow Q'}{\{P'\}\, c\, \{Q'\}} \qquad \text{(CONSEQUENCE)}$$

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.

3.1.1 Separation Logic and Local Reasoning

A serious drawback of Hoare logic is its inability to soundly cope with pointer variables, thus severely complicating its application to low-level systems pro-

grams. After more than forty years of research, a major breakthrough finally came with Reynolds' invention of *separation logic* [30]. A Hoare-style program logic (insofar as the axioms and inference rules are similar), the crucial difference between it and Hoare logic is the assertion language. Instead of the first-order logic assertions used by Hoare logic specifications, separation logic makes use of a new logic[4] for describing system states with a notion of a heap, into which pointers point, and disjointness of said states. The salient formulas that capture these notions are the *points-to assertion*, $x \mapsto y$, and the *separating conjunction*, $P * Q$. Models of the first formula are heaps with exactly one address allocated, given by the value of x, and with the value of y stored at that address. Models of the second formula are heaps that can be partitioned by address into two subheaps, one of which models the formula P and the other 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*—i.e., just the part of the system state referenced by the program during its execution—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 [25]:

> 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[5] for the store command requires with its precondition only that the relevant location be allocated with some value,[6] and the resulting postcondition 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*:[7]

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

[4] More precisely, a theory of the logic of bunched implications pioneered by O'Hearn, Pym and others [24, 29].

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

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

[7] Some formulations of the frame rule additionally require the syntactic side condition that none of the variables modified by the command c occur freely in R.

No similar conjunctive frame rule is sound in Hoare logic, limiting its ability to scale to large programs.

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 [36]), useful fragments of separation logic have been automated as part of program verifiers and static analysis implementations, which have been successfully applied to multi-thousand line programs [2, 37].

3.1.2 Concurrent Logics

Research into logics for concurrent programs has progressed independently. Early attempts culminated in an extension of Hoare logic by Owicki and Gries [27], 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.

Some relief from these problems was provided by Jones in his rely/guarantee logic [18], 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.

3.1.3 A Marriage of Techniques

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 [34] ("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.[8] 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.

RG-Sep embodies the most advanced ideas about high-level reasoning techniques for fine-grained concurrent shared-memory programs, including local reasoning. Vafeiadis' 2008 dissertation [33] includes correctness proofs for a variety of complex, racey 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 I provide here is that, as with all the other concurrent logics discussed, RG-Sep is not sound w.r.t. weak memory models for racey programs.

3.2 Related Solutions to This Problem

Any of the concurrent logics mentioned above can be used to reason about race-free programs soundly w.r.t. weak memory models because of DRF guarantees. An instance of this was recently formalized [10] for a concurrent variant of separation logic based on Owicki-Gries that treats race conditions as runtime errors [23, 4]. But I am aware of little existing research on the general problem of verifying concurrent software that may include race conditions. Most of what does exist is ad hoc.

The most general solution proposed is from Ridge, who syntactically transforms ML programs to explicitly include the behavior of the memory model by, e.g., rewriting heap update commands to instead enqueue the write to a FIFO list that models a store buffer [31]. A general purpose higher-order logic is then used to reason about the transformed programs. This is perhaps simpler than reasoning directly about a weak memory operational semantics, though one gains none of the benefits of reasoning with a high-level logic. (Previously, I applied a similar technique, using RG-Sep to reason about transfomed C programs. This proved cumbersome at best, but suggested some useful abstractions and frame rules.)

[8] An interesting consequence is that $\boxed{P} * \boxed{Q}$ logically implies $\boxed{P \wedge Q}$.

Additionally, there is work by Burkhardt, Musuvathi and others on verifying compiler transformations on weak memory models [5], and work that shows that the finite-state weak memory verification problem is more computationally complex than the corresponding strong memory problem [1]. Finally there has been work on verifying software transactional memory implementations [13].

Both compiler transformations/optimizations and software transactional memory implementations are excellent examples of classes of software, besides concurrent data structures, that are essentially racey by design and so cannot be soundly handled by existing program logics.

4 Proposed Project

For my dissertation, I propose to explicate a principle of local reasoning for a weak memory model. To that end, I will develop a Hoare-style program logic for fine-grained concurrent shared-memory programs, which may include races, that is sound w.r.t. an x86-like TSO memory model. I have chosen an x86-like memory model for the following reasons:

1. the memory model has been formally specified, thoroughly tested, and is believed to be sound with respect to existing instantiations of the architecture;
2. the semantics is robust, given by equivalent axiomatic and operational definitions;
3. the operational definition is comparatively simple and intuitive;
4. the architecture is extremely widespread, yielding maximum impact to advances in reasoning.

Other architectures either have less well studied or specified memory models, or are less widespread.[9]

The program logic I propose is based on separation logic and, in particular, RG-Sep. Separation logic has proven to be an effective formalism for high-level reasoning about pointer programs, and I believe a solution along the lines of the RG-Sep framework, tailored for the x86-TSO memory model, provides the best chance of success for high-level reasoning about racey programs like concurrent data structures.

However, this logic is far from a minor tweak on RG-Sep. Indeed, nearly every technical aspect of RG-Sep requires research and major adjustment to account for the complexity of concurrent reasoning w.r.t. this memory model. Below, I describe the major issues that must be addressed in the development of a weak-memory concurrent program logic, as well as my work in progress toward this goal.

[9] A unified reasoning framework, parametrized by the memory model, is clearly desirable, but beyond the scope of this dissertation.

4.1 Semantic Model

The operational description of the x86 memory model is based on per-processor *write buffers*: lists of writes (pairs consisting of a memory address and a value). When a processor performs a store instruction, it enqueues a write in its write buffer instead of writing the value directly to the appropriate address in shared memory. When a processor performs a load instruction, it uses the value of the most recent write to the appropriate address in its write buffer if such a write exists, and otherwise uses the value at that address in shared memory. The writes in a processor's write buffer may be flushed to main memory non-deterministically, but in FIFO order; or a processor may flush its write buffer completely by issuing a fence instruction.

Load instructions only reference the write buffer of the processor on which the load was issued, and not other processor's buffers, so different processors may not agree on the value of all memory addresses. But the opposite assumption—that for each address there is exactly one agreed-upon value at all times—is embedded deeply in the semantics of every program logic discussed above. Assertions in Hoare logic and separation logic are formulas whose models abstractly represent system states, but the classes of mathematical objects from which these models are drawn are not well suited to express otherwise. The intended models of Hoare logic assertions are valuations of program variables (called *stores*); models of separation logic assertions additionally include partial functions (called *heaps*) that map memory addresses to values that represents dynamically allocated shared memory. In both cases, the functions map their respective resources (program variables and memory addresses) to a single value.

A program logic for the x86-TSO memory model ought naturally make use of more elaborate mathematical objects to represent system states. The model I have developed includes a store and heap, as well as lists that represent per-processor write buffers. But it also includes other components needed to model fences, locking and dynamic memory management—features which are either unnecessary or implicit in simpler program logics—as well as features that enable local reasoning. Rigorous definition of the semantic model and of the sequential logic described next are available in an unpublished technical note [35].

4.1.1 Locality

For this project, the crucial test of a semantic model is whether it supports a local reasoning principle. At this level, local reasoning is embodied by a technial property, called *locality*, which relates the system model and the semantics of the programming language [6]. Locality is an order-theoretic property which means, informally, that the result of executing a program on a more defined system state results in a more defined outcome. (A precise formulation of locality is beyond the scope of this document.)

The most important contribution of this dissertation project is the definition of a semantic model that reflects the x86 memory model, along with a corresponding proof of locality. As such, the development of the semantic model has

taken place with locality in mind from inception. A model that supports locality is the most important scientific contribution because it is the key component of a formal explication of a weak-memory local-reasoning principle, and a necessary precondition for the development of any separation logic that is to be sound w.r.t. an x86-like memory model. Locality is the foundation upon which the soundness of frame rules in any logic lies.

4.2 Sequential Logic

The next step is to develop a strictly sequential program logic, which generalizes separation logic and uses a simplified version of the semantic model referenced above, with a single write buffer. Though a sequential logic is not itself useful (unless one wishes to prove more precise specifications of sequential programs, which describe the relative order of order of writes), it is an important way-point toward a concurrent logic for two reasons. First, the specification language and corresponding logic constitute a large fragment of an eventual corresponding logic, and soundness proofs for a sequential logic will strongly inform similar proofs for a concurrent logic. Second, I expect the concurrent logic to leverage the sequential logic directly in some ways, making it a genuine component of the concurrent logic.

Relative to the concurrent logic, only commands for atomic sections and concurrent composition are omitted from the target programming language in this logic. But, in particular, the sequential logic handles dynamic memory management (i.e., "new" and "free" commands) and fences. Syntactically, the logic includes the same formulas as in traditional sequential separation logic—including the points-to assertion $e \mapsto e'$ and the separating conjunction $P * Q$—and admits the same inference rules—including the frame rule for the separating conjunction. The logic has different small axioms because the semantics of commands in a TSO memory model are so different from the stronger model, and includes additional atomic formulas and conjunctions to precisely describe TSO system states. In particular, the sequential assertion logic adds the following formulas:

- the *leads-to assertion*, $e \rightsquigarrow e'$, which describes a pending write in the write buffer, instead of a value already in the heap with the traditional points-to assertion;
- the *weak sequential conjunction*, $(P\,;Q)$, which describes a composition of system states in which write buffers are concatenated and heaps are unioned (if consistent); and
- the *strong sequential conjunction*, $(P\,,Q)$, which has the same meaning as the weak sequential conjunction, but which requires that the heaps described by P and Q have disjoint domains.

Of course, the meanings of the traditional formulas must change as well. In particular, $P * Q$ now describes a composition of system states in which the

write buffer of P and write buffer of Q are interleaved. And, as before, it is only defined when the domains of the respective heaps are disjoint.

For example, consider the formula $(x \mapsto 0 * y \mapsto 0)\,;(x \rightsquigarrow 1\,, y \rightsquigarrow 2)$. The left-hand side describes a heap with two distinct heap locations allocated, given by the values of identifiers x and y respectively, at which the value of each is 0. The right-hand side describes a write buffer with two distinct writes, the first to the location x and the second to the location y.[10]

The new conjunctions also lead to new frame rules.[11] Because it is safe to add additional elements to a write buffer *behind* those needed for a command's execution, there is a left-side weak sequential frame rule:

$$\frac{\{P\}\,C\,\{Q\}}{\{R\,;P\}\,C\,\{R\,;Q\}} \qquad \text{(L-SEQ-FRAME)}$$

Furthermore, it is safe to add additional elements to a write buffer in front of those needed for a command's execution when the writes are to disjoint addresses, there is a right-side strong sequential frame rule:

$$\frac{\{P\}\,C\,\{Q\}}{\{P\,,R\}\,C\,\{Q\,,R\}} \qquad \text{(R-SEQ-FRAME)}$$

For example, consider a load of x in a state described by the assertion $x \rightsquigarrow 1$. The result of the load will of course be 1. The first frame rule indicates that the result of the load is the same in a state described by $(x \rightsquigarrow 2\,; x \rightsquigarrow 1)$. The second frame rule indicates that the result of the load is the same in a state described by $(x \rightsquigarrow 1\,, y \rightsquigarrow 2)$. The first frame rule is sound for load because that command only uses the top-most write; and the second because that command ignores intervening writes to different locations.

Status The syntax and semantics of the sequential assertion language have been rigorously defined, as well as the syntax, semantics and inference rules for the sequential program logic. Some important properties (e.g., associativity of the new conjunctions) have been mechanically proved. It remains to prove the soundness of the program logic, and to develop an inference system for proving validity of implications in the assertion language, which are needed to apply the rule of consequence in the program logic. As there is no complete proof system for the standard model of separation logic assertions, I do not expect to develop a complete system for this assertion logic. Hence, this problem is essentially to gather a set of equivalence and implication schemas, at worst proven semantically, to make the program logic generally useful.

[10] Here, the right-most end of the list indicates the head of the write buffer, where new writes are enqueued.

[11] Again, possibly with syntactic side conditions on free variables.

4.3 Concurrent Logic

After proving the soundness of the sequential logic, I will move on to the concurrent program logic, which will generalize both the sequential logic and RG-Sep. The semantic model will support multiple processors and concomitant state components, like write buffers. States will also be split into private and shared parts, as in RG-Sep.

The assertion logic can already characterize interleaving concurrency with the separating conjunction $P * Q$, but I anticipate the need for an additional conjunction $P \parallel Q$ for representing processor separation. (In a single-processor model of concurrency, these are identified.) The logic will also require adjustment for distinguishing the private and shared system states; in particular, boxed assertions $\boxed{P}$ must be defined, and the meaning of the various separating conjunctions must be adapted to this new class of assertions. Additional rules must be added to the assertion logic's proof system so that the validity of implications can be mechanically determined for use with the rule of consequence.

The programming language contains additional commands for atomic sections as well as interleaving parallel composition and multi-processor parallel composition. The rules of the program logic must be updated in the style of rely/guarantee, and must additionally contain rules for reasoning about atomicity and concurrency. As in RG-Sep, the rule for atomic sections will leverage the sequential logic directly, which further justifies its initial development.

Furthermore, as in RG-Sep, specifications in the concurrent logic will include descriptions of environmental interference: a rely condition that bounds the interference the environment may inflict upon shared state, and a guarantee condition that bounds the interference the command may inflict upon shared state. A syntax for describing this interference is required. (Because interference is semantically described as a binary relation on states—relating system states before and after another process performs some updates—the assertion logic itself, which describes flat sets of states, is not suitable.)

Most inference rules in the concurrent logic require that specific parts of the pre and postconditions be *stable* w.r.t. environmental interference—e.g., a precondition assertion is stable w.r.t. a guarantee condition if, whenever a state that satisfies the precondition undergoes interference according to the guarantee condition, it continues to satisfy the precondition. A crucial issue in a weak-memory generalization of RG-Sep is the development of a suitable semantic definition of stability, a useful syntactic approximation of stability, and a proof system for its mechanization.

A major simplifying assumption implicit in the sequential logic is that writes from the write buffer only commit to shared memory as a result of fence operations. But a more precise model must account for the non-deterministic FIFO flushing described by the full operational model. The semantic definition of stability will account for this nondeterministic flushing. For example, consider a precondition assertion $(x \mapsto 0 \,;\, x \rightsquigarrow 1)$, which describes single-address heaps with value 0, and with a pending write of value 1. According to the x86-TSO

memory model, such states may nondeterministically transform to states described by the assertion $x \mapsto 1$ as a result of buffer flushing. The former state must therefore not be considered stable. On the other hand, if there is no other environmental interference, the disjunction $(x \mapsto 0 \,;x \rightsquigarrow 1) \vee x \mapsto 1$ is stable, as it describes a set of states closed under this transformation.

Status Most of my work on the concurrent logic has been toward the development of a suitably general semantic model. But the concurrent logic has yet to be developed concretely beyond that. I believe I understand many of the semantic issues, including the semantics of the additional commands and assertions, and the semantic notion of stability. But the syntactic issues, including the syntactic approximation and proof system for stability, remain as yet unexplored. A key challenge is to find a syntactic approximation of stability that is precise enough to prove linearizability of interesting programs, yet easily computable.

5 Evaluation

To evaluate a local reasoning principle for a weak memory model, I have proposed to develop a sequential and concurrent program logic that embodies the principle. The program logics themselves should be evaluated with respect to two criteria:

1. technical suitability via metatheoretic properties (like soundness and completeness w.r.t. the semantics of the programming language), the compatibility of the semantics w.r.t. the memory model, and the relation of the logic to previously studied logics.

2. empirical suitability via proof development to determine whether interesting programs can be proved, and to determine the ease or difficulty with which such proofs can be constructed.

I discuss my proposed evaluation of technical suitability (Section 5.1) and empirical suitability (Section 5.2) in turn below.

5.1 Technical Evaluation

Of all possible metatheoretic properties one might investigate, soundness is clearly the most important. Indeed, I have developed the semantic model from the beginning with soundness proofs in mind, especially w.r.t. the frame rules. I consider a soundness proof for a sequential logic an important deliverable, and I believe it to be within reach. A soundness proof for a concurrent logic is also desirable, but may be beyond the scope of this dissertation project.[12]

[12] In support of this, I note that a different concurrent extension of separation logic, invented by O'Hearn and called Concurrent Separation Logic [23], was studied for some time without a semantic model or proof of soundness. These were given after five years by Brooks [4].

I do not expect either logic to be complete in any technical sense (e.g., Cook completeness, as with Hoare logic), and so will not investigate this property.

As progress toward a soundness proof, I have formalized the sequential fragment of the semantic model using Coq [3] and have proved a number of algebraic properties (viz. associativity, commutativity, and identities) of the separating conjunctions.[13]

Agreement between the semantic model used by the logic and programming language with the specification of the memory model is another important property. Ideally, I would be able to prove that the model agrees with the existing HOL formalization of the x86-TSO memory model [26]. Alternatively, I could investigate simpler properties known of the abstract model, but without considering the complete details of the memory model—e.g., the total-store ordering property of writes. Ultimately I believe the abstract model to be sufficiently unique and different from existing models, and so manifestly "weak" that such proofs need not be considered necessary for the success of the project. A proof of agreement with the full model would yield the first *denotational model* of the x86-TSO memory model; but even without such a proof, the logic still constitutes the first weak-memory separation logic, which I believe is a worthy contribution on its own. I will therefore investigate the agreement property only if time allows.

Similarly, although I believe that the sequential fragment of the logic generalizes separation logic, and I intend to develop the concurrent logic as a generalization of RG-Sep, such an observation would be of limited practical interest. I therefore intend to investigate this property only if time allows.

5.2 Empirical Evaluation

The local reasoning principle and basic suitability of the sequential and concurrent logics can be tested with proofs of simply specified small programs. If I can make sufficient progress on the concurrent logic, I also wish to produce *linearization* proofs of some non-trivial concurrent data structures. (This could be undertaken even without a soundness proof.) Linearizability [15] is a correctness condition and proof technique for concurrent objects that relates complex implementations to their simpler, abstract counterparts.

The non-blocking concurrent stack by Treiber [32] is a good structure to start with due to its relative simplicity. Treiber's stack is lock-free, both in the informal sense that it makes use of no locks (instead only atomic compare-and-swap (CAS) operations), and formally as a liveness property (meaning that some concurrent operation always makes progress [12]).

The stack is based on linked lists; the `push` operation works as follows. A new list node, which will become the new list head, is allocated and initialized with the value to be pushed, and the `next` pointer is then set to the current head of the list. If the list head has not subsequently been changed by a concurrent operation since the `next` assignment, the list head is set to the new node.

[13] The proofs are online at `http://cs.utexas.edu/~iwehrman/x86-logic/coq`

(The aforementioned load, test and store are performed atomically by the CAS operation.) Otherwise the process loops, re-setting new node's `next` pointer to the now-current list head. The `pop` operation uses a similar tight CAS loop to ensure that it pops only the current list head. From previous investigation, I believe that it is possible to give a linearization proof of Treiber's stack with x86-TSO semantics.

A next step could be the concurrent list-based stack from Hendler, Shavit and Yerushalmi [14], which performs better under high load than Treiber's stack. The HSY stack, as it is known, improves on Treiber's stack by using an *elimination scheme*, a general technique used to resolve conflicting operations without waiting for each to complete in turn. For example, in Treiber's stack, when a `push` and `pop` operation conflict, the implementation of those operations simply waits in a loop for the other to complete before proceeding. The elimination scheme used by the HSY stack improves on this by allowing the conflicting operations to complete immediately, without modifying the main data structure: the `pop` operation returns the value of the conflicting `push` operation, which itself returns normally. From the clients' view, it is as though the `push` operation successfully updated the data structure, with the `pop` operation following after. Naturally, this complicates the linearizability argument.

Another interesting concurrent data structure is the list-based concurrent queue from Michael and Scott [21]. There are two locks: one for the head, and one for the tail. A client must acquire the head lock to enqueue, and the tail lock to dequeue. Enqueue and dequeue operations thus do not generally conflict and may operate concurrently. (The empty queue is an exception, handled specially by both operations.)

The two-lock queue is interesting because it is technically linearizable. Linearization is proved by identifying a point in the implementation of each operation—the *linearization point*—at which the corresponding abstract operation can be thought to have completed. For example, in the `push` operation of Treiber's stack, the linearization point comes outside the loop, after the CAS operation that successfully updates the head pointer. I do not believe the two-lock queue is linearizable because, based on previous investigation, it seems that the linearization points for its operations do not occur within the bodies of the functions that implement the abstract operations. Instead, the linearization points occur after the next fence operation to occur, which happens at the beginning of the next operation, whatever that may be. I do believe the algorithm to be correct (in a sense yet to be formally defined) on the x86-TSO memory model though, and hope to find a generalization of linearizability that can be used to demonstrate this. Such a generalization might also be useful for other weak-memory logics, or show the way to similar generalizations for other memory models.

6 Discussion

What Will Not Be Addressed One interesting question is whether a particular instantiation of a computer architecture implements its specified memory

model—e.g., whether recent iterations of the Intel x86 architecture are faithful to the aforementioned formally model. This is orthogonal to the question of whether a program is correct w.r.t. a memory model's specification, however, and will not be addressed in this dissertation.

Liveness and termination are important issues for concurrent data structures, on which separation logic has recently helped to shed light [12] under strong memory consistency assumptions. And the liveness properties of the memory model itself will surely affect liveness of programs above, making this an interesting and challenging problem. But liveness properties of common architectures's memory models has not yet been thoroughly studied or specified. Consequently, it does not yet seem appropriate to study these properties for programs on weak memory models.

A major factor in the success of separation logics is the degree to which they have been successfully automated. Although I am not aware of any aspects of the logics I propose which would prevent similar automation, the focus of this dissertation is foundational and not immediately practical. I will consider the project a success if I can precisely formulate a weak-memory separation logic and use it to construct hand-proofs of small, interesting programs. Automation of the construction of these proofs—e.g., with a program verifier—should be considered future work.

Schedule Below is a timeline of past and proposed future work:

- **Fall 2008–spring 2009:** Background research on memory models and separation logic.
- **Summer 2009:** Problem formulation; exploration of alternate approaches. Initial research on a weak-memory separation logic.
- **Fall 2009–spring 2010:** Development of semantic model. Syntax of assertion logic and sequential proof system.
- **Summer 2010–fall 2010:** Soundness proof for sequential fragment. Small examples in sequential logic.
- **Fall 2010–spring 2011:** Finalize concurrent semantic model. Syntax and proof system for concurrent logic.
- **Summer 2011–summer 2012:** Soundness proof for concurrent fragment. Linearization proofs of concurrent data structures.

References

[1] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. In Manuel V. Hermenegildo and Jens Palsberg, editors, *POPL*, pages 7–18. ACM, 2010.

[2] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[4] Stephen D. Brookes. A semantics for concurrent separation logic. In Gardner and Yoshida [11], pages 16–34.

[5] Sebastian Burckhardt, Madanlal Musuvathi, and Vasu Singh. Verifying local transformations on relaxed memory models. In Rajiv Gupta, editor, *CC*, volume 6011 of *Lecture Notes in Computer Science*, pages 104–123. Springer, 2010.

[6] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS*, pages 366–378. IEEE Computer Society, 2007.

[7] K. Mani Chandy and Jayadev Misra. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[8] Nathan Chong and Samin Ishtiaq. Reasoning about the arm weakly consistent memory model. In Emery D. Berger and Brad Chen, editors, *MSPC*, pages 16–19. ACM, 2008.

[9] Edmund M. Clarke and E. Allen 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.

[10] Rodrigo Ferreira, Xinyu Feng, and Zhong Shao. Parameterized memory models and concurrent separation logic. In Andrew D. Gordon, editor, *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 267–286. Springer, 2010.

[11] Philippa Gardner and Nobuko Yoshida, editors. *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, volume 3170 of *Lecture Notes in Computer Science*. Springer, 2004.

[12] Alexey Gotsman, Byron Cook, Matthew J. Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In Zhong Shao and Benjamin C. Pierce, editors, *POPL*, pages 16–28. ACM, 2009.

[13] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Software transactional memory on relaxed memory models. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 321–336. Springer, 2009.

[14] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In Phillip B. Gibbons and Micah Adler, editors, *SPAA*, pages 206–215. ACM, 2004.

[15] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[17] Samin S. Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. In *POPL*, pages 14–26, 2001.

[18] Cliff B. Jones. *Development methods for computer programs including a notion of interference*. PhD thesis, Oxford University, 1981.

[19] Matt Kaufmann and J. Strother Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Trans. Software Eng.*, 23(4):203–213, 1997.

[20] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[21] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.

[22] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[23] Peter W. O'Hearn. Resources, concurrency and local reasoning. In Gardner and Yoshida [11], pages 49–67.

[24] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[25] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[26] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer, 2009.

[27] Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[28] Leaf Petersen and Manuel 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.

[29] David J. Pym, Peter W. O'Hearn, and Hongseok Yang. Possible worlds and resources: the semantics of BI. *Theor. Comput. Sci.*, 315(1):257–305, 2004.

[30] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[31] Tom Ridge. Operational reasoning for concurrent caml programs and weak memory models. In Klaus Schneider and Jens Brandt, editors, *TPHOLs*, volume 4732 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2007.

[32] R.K.Treiber. Systems programming: Coping with parallelism. rj5118, April 1986.

[33] Viktor Vafeiadis. Modular fine-grained concurrency verification. Technical report, University of Cambridge, 2008. Technical Report UCAM-CL-TR-726.

[34] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2007.

[35] Ian Wehrman. Semantics and syntax of a weak-memory concurrent separation logic. `http://cs.utexas.edu/~iwehrman/x86-logic/semantics.pdf`, 2010.

[36] Hongseok Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 2001. Technical Report UIUCDCS-R-2001-2227.

[37] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 385–398. Springer, 2008.