Multimodal Separation Logic for Reasoning About Operational Semantics

Robert Dockins Andrew W. Appel Aquinas Hobor

Princeton University

Abstract

We show how to reason, in the proof assistant Coq, about realistic programming languages using a combination of separation logic and heterogeneous multimodal logic. A *heterogeneous* multimodal logic is a logic with several modal operators that are not required to satisfy the same frame conditions. The result is a powerful and elegant system for reasoning about programming languages and their semantics. The techniques are quite general and can be adopted to a wide variety of settings.

Keywords: Modal logic, separation logic, operational semantics, mechanical verification

1 Introduction

Recent years have seen major advances in the field of machine-verified proofs for software correctness. In the process, techniques have been developed to aid the process of reasoning about software systems.

Separation logic has emerged as a powerful way to reason about programming languages with mutable stores [25,21,18]. Separation logic contains the substructural *separated conjunction* in addition to the ordinary conjunction of propositional logic. The separated conjunction of two predicates, $P * Q$, captures the notion that the heap can be split into two disjoint pieces, one of which satisfies P and the other of which satisfies Q. Separating conjunction succinctly represents nonaliasing conditions that would otherwise have to be handled manually.

Modal logic is another useful tool for reasoning about programming languages [23,24,20]. Recent work demonstrates how modal operators can succinctly express the construction of semantics for mutable references, recursive functions and recursive predicates [4,15]. Modal logics have also been used in the KeY project to verify JavaCard programs [7].

We show how to productively combine separation logic with heterogeneous multimodal logic. The result is a powerful and elegant system for reasoning about programming languages and their semantics.

©2008 Published by Elsevier Science B. V.

By design, our logic embeds well in the proof assistant Coq. The logic and the (shallow) embedding are quite general and can be adopted to a variety of settings, including interesting, industrial-strength programming languages.

In the next section, we will describe the general form of programming language semantics we treat. The following several sections will present examples of different programming languages of increasing complexity. The needs of each particular language will be used to motivate various features of our model. We will adopt a fairly informal tone and will focus on practical applications of the techniques.

2 Generic operational semantics

Our goal is to reason about a variety of programming languages. We must therefore develop a framework into which we can fit the syntax and semantics of the languages of interest.

We have chosen to reason about deterministic small-step operational semantics. Operational semantics are fairly easy to construct both for low-level programming languages such as assembly language as well as higher level languages. Operational semantics, as a specification language, may be more likely to correspond to the intended meaning of low-level languages than (for example) axiomatic semantics. We chose to reason about small-step semantics because we are specifically interested in languages which feature shared-memory concurrency; using a big-step operational semantics makes reasoning about possible interleavings difficult. Finally, we stipulate that the semantics be deterministic for convenience in reasoning: we can then do simple induction over the number of steps taken rather than doing induction over all possible paths. We fit nondeterminism into this framework by using oracles to determinize; see section 7 for an example.

The general form of our small-step semantics is as follows:

$$\Psi \vdash w,\, \kappa \longrightarrow w',\, \kappa'$$

Here, Ψ stands generically for some description of the program being run and is assumed to be constant throughout the execution. The informal meaning of the judgment is that the pair (w, κ) reduces in one step to (w', κ') in the program described by Ψ. The terms w and w' are of a type $\mathbf{W}$ called "worlds" and represent the state of the running process. This state will typically include things like the contents of memory, a current value to evaluate, *etc.* The exact details of the worlds will vary from setting to setting. The values κ and κ' of type $\mathbf{K}$ represent the "control continuation." [1] The types $\mathbf{K}$ and $\mathbf{W}$ are separated because our specification logic is defined using predicates on worlds. Because these are predicates only over w, and not on κ or Ψ, we can embed statements of the specification logic in programs and continuations, which is sometimes convenient.

A program state typically has multiple components which may be altered during execution. For example, in a von Neumann machine, the memory or the register bank may be modified on each instruction. For this reason, it makes sense to consider a world w as being a labeled tuple containing some number of individual

[1] In C-like languages, the control continuation is similar to the program stack augmented with the program counter. In some languages there is no interesting notion of a control continuation, and one might instantiate $\mathbf{K}$ as the unit type.

components which may be modified independently. In addition, a world will contain other components that do not affect program execution, but are used during proofs—in the same way that type annotations in an untyped lambda calculus may aid proofs but do not affect operational semantics. We will provide examples of these in the following sections.

For the remainder of this section we will hold the exact definition of programs, worlds and control continuations abstract; later on we will see how to instantiate these components to discuss particular programming languages.

We wish to use separation logic and multimodal logic to reason about programming languages in the proof assistant Coq. The metatheory of Coq does not include modalities or separation logic, so we must encode the object logic we wish to use. There are two basic approaches one can follow when embedding a logic. In a *deep embedding*, also called a *syntactic embedding*, one first defines the *syntax* of the object logic, usually by an inductive definition. One also gives the proof theory of the logic, typically by giving an inductive definition of the allowable rules of inference. This approach is therefore also sometimes called the "proof theoretic" approach. In a *shallow embedding*, also called a *semantic embedding*, statements in the object logic are defined directly by writing down expressions that encode their semantics in the metalogic.

A shallow embedding is not always possible; for example, it is not possible to shallowly embed second-order logic in first-order logic because there is no first-order construct that can directly encode second-order quantification. However, when possible, there are some distinct advantages to shallow embeddings. For one, the syntax of the object logic is not fixed *a priori*, but can be extended at any time by simply providing a new metalogic definition. In a similar way, inference rules can be added at any time by proving (in the metalogic) that the rule is admissible according to the semantics. The second major advantage of a shallow embedding is that one is able to directly reuse the variable binding structure of the metalogic. As we shall see below, this allows us to define impredicative quantification essentially "for free," avoiding the POPLmark quagmire [5]. When doing machine-verified proofs, this feature is particularly salient.

To perform our shallow embedding, we must first decide how to define the semantics of statements in our object logic. It turns out that a very simple definition suffices. In the syntax of Coq:

```
Definition pred := world → Prop.
```

Here the type `pred` is the type (in the metalogic) of statements in the object logic. Notice that this definition is not inductive, and we make no mention of the logic connectives we intend to use; they will all be introduced later by directly defining their semantics. The name `pred` refers to the fact that object-logic statements are predicates on worlds.

Our choice of terminology is intended to evoke a connection to Kripke semantics (possible-worlds semantics) for modal logics. In a Kripke semantics one gives a set of worlds and a forcing relation $\models$ that relates worlds and formulae. For any formula p and world w the relation $w \models p$ is read "w satisfies p." In our system, we have folded the meaning of the forcing relation directly into the definition of object

logic formulae; in other words, the forcing relation is just provability in the Coq metalogic. We can define the forcing relation for our object logic in Coq as follows:

```
Definition forces (w:world) (p:pred) : Prop := p w.
```

The final component of a traditional Kripke model is a binary relation on worlds, R, called the accessibility relation. The accessibility relation is used to give semantics to the modal operators of the logic. Accessibility relations in multimodal logics are quite interesting, as we will discuss in section 4.

These simple definitions already give us enough to define some basic logical operators. In Coq syntax we have:

```
Definition bot := fun _ => False.
Definition top := fun _ => True.
Definition conj (t1 t2:pred) := fun w => t1 w ∧ t2 w.
Definition disj (t1 t2:pred) := fun w => t1 w ∨ t2 w.
Definition implies (t1 t2:pred) := fun w => t1 w → t2 w.
Definition all (A:Type) (F:A → pred) := fun w => ∀x:A, F x w.
Definition ex  (A:Type) (F:A → pred) := fun w => ∃x:A, F x w.
```

The main thing to note here is that the basic connectives of higher-order logic can be easily "lifted" into the object logic. The presence of impredicative universal and existential quantification is particularly notable; our system can effortlessly quantify over any type representable in the metalogic, including statements of the object logic. For notational convenience we write object logic conjunction using `&&`, disjunction using `||` and implication using $\Rightarrow$. This usage is made formal in Coq via syntax directives.

Finally, we define a judgment on predicates that we call `derives`. It expresses (in the metalogic) that one statement entails another in all possible worlds.

```
Definition derives (t1 t2:pred) := ∀w, t1 w → t2 w.
```

We use a turnstile notation $\vdash$ for the derives relation.

Using these definitions and notations, we can now prove lemmas corresponding to the rules of inference for intuitionistic logic. For example:

```
Lemma top_intro : ∀p, p ⊢ top.
Lemma bot_elim  : ∀p, bot ⊢ p.
Lemma conj_elim1: ∀p q, p && q ⊢ p.
Lemma conj_intro: ∀p q x, (x ⊢ p) → (x ⊢ q) → (x ⊢ p && q).
```

3 A simple list machine

Much of the preceding discussion has been quite abstract. In this section, we demonstrate how the multimodal logic can be used to reason about a toy language, the *list machine* proposed by Appel and Leroy [2] as a benchmark for evaluating machine-checked proofs about type systems. The list machine is a minimal Typed Assembly Language (TAL) for list processing; it has few instructions and a simple model of program state. The state consists of an unbounded "register bank," where each register can contain a value. A value is either the nil value or a cons cell containing two other values. Control flow is provided by a conditional jump and an unconditional

jump. Data instructions exist to create a new cons cell and to read the first and second values from a preexisting cons cell. Programs in this language are partial mappings from label values to blocks of instructions.

Appel and Leroy examine a simple typechecker for this language by using proof-theoretic methods in both Coq and Twelf. Here we examine the same language using semantic methods. [2]

For the list machine, Ψ represents the program to execute. The κ are sequences of instructions to execute. We lack the space to enumerate the instruction set here; we refer the interested reader to the benchmark paper [2]. Worlds are tuples, $w = (n, \rho, v)$ where n is the "age" of the world, ρ is the register bank (a mapping from register numbers to values), and v is a value. Only the ρ component of the world affects program execution; the other components are proof artifacts. v exists so that we can make statements about particular values, and n is important for the definition of recursive predicates. We will discuss world ages in some detail below.

In concrete Coq syntax, we can summarize the definitions required as follows: [3]

```
Inductive instr : Set := ...
Definition program := map instr.

Inductive value : Set :=
  | value_nil   : value
  | value_cons : value → value → value.

Record world : Type :=
  { world_n: nat; world_r: map value; world_v: value }.
```

With these definitions made, we can start to write interesting statements in our object logic. For example, we can write a very simple statement which is true in any world where the value component is nil.

```
Definition isnil := fun w => world_v w = value_nil.
```

We can also write a formula that specifies that some component of the register bank satisfies a property of interest.

```
Definition slot (n:nat) (p:pred) := fun w =>
    ∃x:value, lookup n (world_r w) = Some x ∧ p (set_v x w).
```

Informally, `slot n p` means that if you look up the value at register `n` in the current world, and then set the value component of the world to that value, the predicate `p` will be satisfied. In a similar way, we can define a predicate `pair p q` which is true on a cons cell where the first component satisfies `p` and the second component satisfies `q`. To wit:

```
Definition pair (p q:pred) := fun w => ∃v1:value, ∃v2:value,
    world_v w = value_cons v1 v2 ∧
    p (set_v v1 w) ∧ q (set_v v2 w).
```

[2] We have a tutorial Coq development that compares the syntactic and semantic methods for the list-machine benchmark, which we expect to have polished and released by the camera-ready date for this conference.

[3] We use a finite map implementation from a standard library. Its details are not particularly interesting, and we will simply assume and use the usual operations.

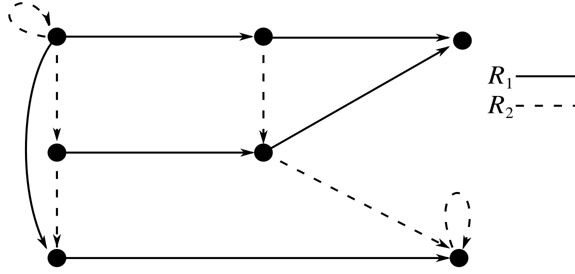

Fig. 1. An example of a set of worlds with two different relations

Now that we have the `isnil` predicate and the `pair` operator, we can make an attempt at defining the predicate `list p`, which defines a list of cells where each member of the list satisfies `p`. Here is a first attempt:

```
Definition listF1 (t:pred) := fun X => isnil || pair t X.
Definition list1 (t:pred) := rec (listF1 t).
```

Informally a list is either the nil value or a cons cell where the second component is again a list. However, we haven't yet defined the crucial fixpoint operator `rec`. Before we can successfully do this, we need to embark on a short digression concerning multimodal logics.

4 Heterogeneous multimodal Kripke semantics

As we mentioned in section 2, a traditional Kripke semantics of modal logic consists of three parts: a set of worlds, a forcing relation, and an accessibility relation. There is a long history in the literature of presenting Kripke semantics this way [19,17]. However, such a presentation is biased toward logics with a *single* modality (and perhaps its negation dual). Experience has shown us that this is too restrictive. We instead wish to have a variety of modalities for different purposes.

Consider instead a model where we fix the set of worlds and a particular forcing relation. Now, given any binary relation on worlds R, we say that R induces a modal operator $\Box_R$. In concrete Coq syntax, the induced modal operator has the following semantics:

```
Definition box (R:world → world → Prop) (p:pred) :=
   fun w => ∀w′, R w w′ → p w′.
```

Essentially, the predicate `box R p` means that `p` is true on any world accessible via `R` from the current world.

Note that in this model there are as many modal operators as there are relations R. Following our recurring theme, we do not have to decide in advance which modalities we wish to use; we can construct a modality from any relation of interest. See figure 2 for some examples of modal operators. This differs from more traditional presentations of multimodal logics where one chooses in advance a particular indexed set of relations R_i; furthermore these R_i are often assumed to have similar properties (e.g., all R_i must be reflexive and transitive) [14]. In contrast, our system makes no assumption about the number of relations or their properties. We call this system a *heterogenous* multimodal logic to contrast it with systems where all the modalities have similar properties. As a consequence, the interactions between the

various modal operators become quite interesting. For example, the commutativity of certain modal operators becomes nontrivial.

Modal operators are uniquely determined by their underlying relations, and it turns out that many simple properties of the relation correspond to interesting rules for reasoning about the operator. The conditions on relations are usually called "frame conditions." Frame conditions are well studied in the modal logic literature and give rise to a variety of useful axioms [16]. For example:

If R is:	then $\Box_R$ satisfies:
any relation	$\Box_R (p \Rightarrow q) \Rightarrow \Box_R p \Rightarrow \Box_R q$
reflexive	$\Box_R p \Rightarrow p$
transitive	$\Box_R p \Rightarrow \Box_R \Box_R p$
reflexive and transitive	$\Box_R p = \Box_R \Box_R p$
transitive and noetherian	$\Box_R (\Box_R p \Rightarrow p) \Rightarrow \Box_R p$

Field update operators form an interesting and important class of operators. These operators allow one to lift into the metalogic the notion of updating fields in the world. For example, we can define an update operator for the "value" field of list-machine worlds, induced by treating update as a Kripke R-relation:

```
Definition Rvupd (v:value) (w w' :world) := set_v v w = w' .
Definition vupd (v:value) := box (Rvupd v).
```

5 Recursive predicates for the list machine

Having introduced the framework of multimodal logics, we now have the necessary vocabulary to meet our goal, the definition of the recursion operator. In section 3, we defined worlds such that they contain an "age" component, but we have not yet discussed its function. The age of a world an be thought of as the number of time units left before the end of the universe. In other words, every time we take a step in the small-step semantics, we will also decrease the age component of the world. After the age of the world ticks down to 0, we stop caring what happens; the machine may afterwards behave in any arbitrary way. It is important to note that the age of a world ticks *downward* toward 0; in fact, it might be better called a time-to-live.

The reason for doing this is, in large part, so we can give semantics to recursion. The method we use was introduced by Appel and McAllester [3], inspired by Scott's D-infinity models: a semantic object is treated as series of increasingly good approximations, indexed by an "age." This method was later organized into a modal logic by Appel *et al.* [4]; here we will provide only the bare essentials.

We will need several modal operators: $\triangleright P$ ("later P") means that P holds on all worlds occurring strictly in the future (more approximate worlds, having a smaller age). $\Box P$ ("necessarily P") means that P holds now and in the future. $\# P$ ("fashionably P") means that P holds in all worlds with the *same* age as the current world. We define these operators by their Kripke R relations:

```
Definition age_w (n:nat) (w:world) : option world := ...

Definition Rage  (w w':world) := ∃i, age_w (i+1) w = Some w'.
Definition Rage' (w w':world) := ∃i, age_w i w = Some w'.
Definition Rsameage (w w':world) := world_n w = world_n w'.

Definition later := box Rage.
Definition necessarily := box Rage'.
Definition fashionably := box Rsameage.
```

The `age_w` function reduces the time-to-live of a world by the specified amount, keeping other fields the same. If the age of the world is smaller than the specified reduction, `age_w` returns `None`. Aging by 0 returns the same world. Note that a sequence of nonzero age reductions on a world will eventually cause `age_w` to return `None`, which means that the `Rage` relation has no infinite descending chain, *i.e.* it is noetherian.

The `Rage` relation is important because it is a well-founded strict partial order; *i.e.*, it is irreflexive, transitive and noetherian. Relations of this sort give rise to operators which obey the Löb rule.

```
Lemma later_loeb_rule : ∀p, (▷p ⊢ p) → (top ⊢ p).
```

Modal operators of this sort were originally studied in the context of logics of provability [8]. However, we find these "Löb operators" useful because the Löb rule succinctly captures noetherian induction. In our context, where the `Rage` relation is defined on world ages, the Löb rule is basically equivalent to complete induction on the natural numbers.

We can use the special properties of the later operator to help us define a fixpoint operator for our logic. Consider the following:

```
Definition sub (p1 p2) := □(#(p1 ⇒ p2)).
Definition equ (p1 p2) := sub p1 p2 && sub p2 p1.
Definition contractive (F:pred → pred) := ∀p1 p2,
   ▷(equ p1 p2) ⊢ equ (F p1) (F p2).

Fixpoint iterate (n:nat) (F:pred → pred) (z:pred) :=
   match n with 0 => z | S n' => F (iterate n' F z) end.
Definition rec (F:pred → pred) :=
   fun w => iterate (1 + world_n w) F bot w.

Theorem fold_unfold : ∀F, contractive F → F (rec F) = rec F.
```

The motivation for these particular definitions and the proof of the `fold_unfold` theorem is the central topic of the "Indexed Model" [3] and we shall not repeat them here. Note, however, that the definition of the recursion operator bears a strong resemblance to the definition found in CPO semantics.

Finally we can define the recursive list type we attempted earlier:

```
Definition listF (t:pred) := fun X => isnil || pair t (▷X).
Definition list (t:pred) := rec (listF t).
Lemma listF_contractive : ∀t:pred, contractive (listF t).
Theorem list_unfold : ∀t:pred,
    list t = isnil || pair t (▷list t).
```

Note there is a subtle difference between this definition and our earlier attempted definition; we have inserted a `later` operator into the definition of `listF`. This extra `later` is responsible for making the definition of `listF` contractive, and thus allows us to use the `fold_unfold` lemma to prove `list_unfold`. In practice, the `later` would normally be incorporated into the semantic definition of `pair`, which is naturally contractive [4].

Although we do not have the space here to discuss further developments, we mention in passing that the logic for the list machine is flexible and powerful enough to define and work with a Hoare triple for this language and to prove the soundness of typechecking algorithms. One method for constructing a Hoare logic in the presence of nontrivial control flow is studied by Tan and Appel [27].

6 Sequential Cminor

Although the list machine is interesting from a didactic and foundational viewpoint, we would like to reason about more realistic languages. For our next example we will focus on the variant of sequential Cminor studied by Appel and Blazy [1]. This language differs from the language of the list machine in an important way: it has a mutable store. Although the list machine has a register bank that can be updated, the individual cons cells are immutable, and there is no pointer construct. In contrast, Cminor has a memory model quite similar to that of C, which has pointers and all the attendant aliasing issues. We use the framework of separation logic to help ease the burden of reasoning about mutable stores [25,18].

We lack sufficient space to explain the language of Cminor in any detail; however, it is quite similar to C, and we shall assume that most readers can draw intuition from this connection. For Cminor, the program description Ψ can be thought of as a partial mapping from memory locations to function bodies. The control continuations κ are "control stacks," abstract representations of the program stack of an ordinary C program. The worlds are tuples of the form $w = (n, \rho, \phi, m, v)$, where n is the world age, ρ is a local environment (mapping identifiers to values), m is the memory, or heap, ϕ is a "resource map" that controls access to the heap, and v is a value. The world values n, ρ and v play very similar roles as they do in the list machine, except that values are now machine words. [4] However, the ϕ and m components were not needed for the list machine; they will be the focus of this section.

The memory m is quite straightforward; it is a partial mapping from addresses to values. The resource map ϕ is a partial mapping from addresses to "shares." A share can be thought of as a rational number between 0 and 1 (inclusive). A nonzero share grants read access to a memory location, and a share equal to 1 grants full read/write permission. [5] The important thing about resource maps is that they allow us to succinctly represent a whole raft of nonaliasing statements by saying that two resource maps are "disjoint." Two resource maps are disjoint if they can be merged so that no location has a share greater than 1. We write

[4] There are a number of subtleties having to do with different word sizes and memory alignment. In this paper we gloss over these details entirely.

[5] The actual definition of shares is somewhat more complicated, but the details are not relevant here.

$\phi_1 \oplus \phi_2 = \phi$ when ϕ_1 and ϕ_2 are disjoint and merge, or *join*, to create ϕ.

Following Calcagno *et al.*, we can build a model of bunched (separation) logic using the joinability relation on resource maps [9]. First we lift the joinability relation on resource maps to a relation on worlds by stipulating that $(n, \rho, \phi_1, m, v) \oplus (n, \rho, \phi_2, m, v) = (n, \rho, \phi, m, v)$ iff $\phi_1 \oplus \phi_2 = \phi$. In other words, two worlds join if their resource maps join and all the other fields are identical. Now we can define the fundamental connectives of separation logic as follows:

```
Definition empty : resource_map := ...
Definition join (w1 w2 w:world) : Prop := ...
Definition emp := fun w => world_phi w = empty.
Definition sep_conj (p q:pred) := fun w =>
   ∃w1, ∃w2, join w1 w2 w ∧ p w1 ∧ q w2.
Definition sep_impl (p q:pred) := fun w =>
   ∀w1, ∀w2, join w w1 w2 → p w1 → q w2.
```

Here `join` represents the joinability relation on worlds and `empty` is the empty resource map, which also serves as a left and right unit for the joinability relation. We use the notation `p ∗ q` for `(sep_conj p q)` and `p −∗ q` for `(sep_impl p q)`. Using these definitions the separating conjunction has a very natural reading; `p ∗ q` means that the heap can be divided into two disjoint parts, where `p` holds on one part and `q` holds on the other. The separating implication `p −∗ q` means, if given a "new" portion of the heap on which `p` holds, then `q` will hold on the joined world. Separating implication is useful for reasoning about situations where new parts of the heap can become available, such as memory allocation.

Critically, this approach to defining separation logic meshes nicely with our approach to defining modal logic. In particular, we can carry out all the same constructions we saw in the section on the list machine, including recursive predicates.

As previously noted elsewhere, one of the major advantages of using separation logic is that reasoning about updates becomes *local* [18,21]. The separating conjunction takes care of propagating uninteresting non-aliasing facts.

7 Quasi-self-referential predicates

Concurrent Cminor [15] is an extension of sequential Cminor which adds support for concurrency in the C-threads model of shared memory with dynamically allocated locks (semaphores) and dynamically forked threads. Our operational semantics ensures the absence of race conditions by associating a *resource invariant* with each lock, in the sense of Concurrent Separation Logic (CSL) [22]. The operational semantics gets stuck if (1) memory is accessed by a thread that does not "own" the resource corresponding to that address, or (2) an *unlock* operation is attempted without satisfying the corresponding resource invariant. The unlock then transfers the corresponding resource out of the thread; some thread that later acquires the lock will also acquire the corresponding resource, and be able to access the memory controlled by the lock.

A safety or partial-correctness proof in CSL will naturally prove that the program does not get stuck in this operational semantics. The resource invariants are assertions of the object logic, CSL; that is, we model them as predicates on worlds.

Therefore, these predicates can not only say what addresses are controlled by the lock, but what invariants hold on the contents of memory at those addresses *at any time that the lock is unlocked*; while a thread holds the lock, it need not satisfy the invariant.

Unlike in O'Hearn's CSL, we have dynamically creatable locks and threads, which are expected in the C-threads programming model. We generalize the notion of "resource" from just "what share of a value-containing location" to a disjunction between value-containing locations and semaphore locations. That is, the resource-map component of a world tells which locations are locks and which locations are ordinary value cells. The programming model has an explicit `make_lock` operation to convert an address from a value location to a lock; this transition modifies the resource-map component of the world, but not the memory.

In this programming model, a thread can create a new lock l_{new} with resource invariant R_{new}. In order to tell other threads about the existence of the new lock, it must pass a pointer l_{new}, presumably through a shared variable controlled by some other lock l_{old} that is already visible to both threads. The resource invariant R_{old} of the old lock must be able to describe the binding of R_{new} to l_{new}. This means that R_{old} is a predicate that must be able to describe the binding of R_{new} to l_{new}. The argument of a predicate contains another predicate!

There is the danger of a paradox here, which we resolve using a technique similar to the "very modal model" of Appel *et al.* [4]. The new construction is not identical because we need to include separation, which was not needed for the first-class mutable references addressed in the that work.

The construction is workmanlike but not very beautiful. Fortunately, its job is only to serve as the model for a modal logic. In practice (in our Coq development of Concurrent C minor), this model is entirely encapsulated inside an axiomatization of quasi-self-referential resource maps, which in turn is encapsulated by our multimodal substructural logic.

```
Record proto_world (A:Type) : Type :=
  { ... ; world_phi : A ; ... }.
Inductive proto_res (A:Type) : Type :=
  | res_share : share → proto_res A
  | res_lock  : share → (proto_world A → Prop) → proto_res A.
Fixpoint proto_resourcemap (n:nat) : Type :=
  match n with
  | O => unit
  | S n' => prod (proto_resourcemap n')
                 (map (proto_res (proto_resourcemap n')))
  end.
Definition resourcemap := { n:nat & proto_resourcemap n }.
Definition resource :=
  { n:nat & proto_res (proto_resourcemap n) }.
Definition world := proto_world resourcemap.
Definition pred := world → Prop.
```

First we define "proto-worlds" as a record type with all the necessary fields (just as usual). Of course, the set of "necessary fields" depends on the application domain; here we use ellipses to stand for a module parameterization in Coq that permits our construction to be used with any set of "necessary fields."

Name	Symbol	Use
later	$\triangleright$	defining recursive predicates and functions
necessarily	$\Box$	nonstrict version of later, useful for defining "well-behaved" versions of implication
fashionably	#	ensures that a predicate holds on all worlds with the same age
extensionally	$\frown$	forces a predicate to ignore portions of the memory to which the resource map grants no read access

Fig. 2. Some example modal operators from the Concurrent Cminor soundness proof

We define "proto-resources" as either ordinary heap shares or as lock resource invariants, where the resource invariants judge proto-worlds. Next we define the stratified resource maps. A resource map at approximation 0 is just a unit value and contains no information. A resource map at approximation $n + 1$ contains a pair of a *more approximate* resource map (at level n) and a mapping from heap locations to proto-resources such that the contained resource invariants judge more approximate proto-worlds. Resource maps are then defined via a Σ type which existentially closes over the approximation level of an enclosed stratified resource map. Finally we can construct worlds by instantiating proto-worlds with the resource map construction. [6]

Notice that the resource map now has a natural number embedded that records its approximation level. We take this directly to be the world age; aging the world now involves reducing the approximation level of the resource map.

Now that we have a workable definition of resource maps, we can define joinability. First of all, two resource maps must be of the same age to join. Next, it must be the case that the resources at each location join pairwise. Two resources join if they are compatible and their shares join. Two regular value resources are always compatible, and two lock resources are compatible iff their resource invariants are the same. Value and lock resources are incompatible. As before, we lift the join relation to worlds, allowing us to define the critical separation logic connectives.

In section 2 we mentioned that it was convenient to allow statements from our object logic to appear in the program and in control continuations. Concurrent Cminor provides a striking example of this situation. Whenever a `make_lock` command creates a lock, it must specify the resource invariant for the lock. Because we defined worlds, and hence our object logic, independently of the syntax of the programming language, we can in fact embed logical assertions directly into the abstract syntax. Thus we can allow the `make_lock` command to directly contain the *semantic, shallowly embedded* statement of the lock's resource invariant. Control continuations also contain statements of the program syntax, so the ability to contain statements of the object logic is important.

In contrast, Gotsman *et al.*'s semantics [13] for Concurrent Separation Logic

[6] The presentation given here is significantly simplified from the construction actually used in the Concurrent Cminor proof. The actual construction deals with a host of other details that we have not the space to cover here.

does not permit the shallow semantic embedding of predicates in programs, so their `make_lock` operation must use an indirection through an external table of resource invariants.

Since we want a deterministic operational semantics to simplify inductive reasoning, we determinize the small-step operational semantics of Concurrent Cminor by providing the thread interleaving schedule as an oracle. Then in soundness (*etc.*) proofs, we quantify over all oracles [15]. Normally one should be suspicious of interleaving proofs, since modern machines have memory models that do not provide sequential consistency. However, our operational semantics provides a strong no-race-condition guarantee connected directly to lock acquire/release, just in the way that modern machines expect, so we may safely use the interleaving abstraction.

8 Related Work

Modal logics have a long history in the philosophical and symbolic logic communities [12]. Kripke semantics in particular are a standard way to to model about these logics [19,17]. Separation logic and the logic of bunched implications have emerged fairly recently as a powerful way to reason about mutable data in programming languages [25,18]. Calcagno et. al. give a nice formulation of the semantics of separation logic which is directly related to our notion of joinability [9].

Using modal logics as a tool for reasoning about programming languages is not a new idea. A form of modal logic for reasoning about programs, known as *dynamic logic* was invented by Pratt in the late 1970s [23,24]. Pratt was able to express the Hoare triple via a modality meaning "after a program a executes, p will be true." Similar to our system, Pratt encoded statements of the object logic as predicates on (sets of) machine states. However, dynamic logic differs from the present work in that Pratt assumes a semantic function that maps an initial machine state into a (set of) final machine state(s), giving a big-step flavor to the system.

The languages studied by Pratt were also rather simple; for example, in one of his early papers on the topic, Pratt studied the language generated by a basic assignment operation and the regular language operators (union, concatenation and Kleene closure) [23]. In a later paper, other basic operations were added, but Kleene closure remained the only way to generate iterative behavior [24]. Although the set of programs that can be generated by regular grammars are interesting, they do not correspond well to languages programmers are accustomed to using.

More recently, the KeY project has been using the framework of dynamic logic to reason about object-oriented programming languages, specifically Java and its variants [6].

Murphy *et al.* use the modal logic IS5 *directly* as a type system for a distributed programming language, Lambda 5 [20]. In that work, modalities are used to represent types for mobile code and remote addresses.

We, and Gotsman *et al.* [13], independently proposed extensions of O'Hearn's Concurrent Separation Logic [22] to first-class locks and threads. These extensions are very similar overall, which is evidence that both are "right." However, Gotsman's logic is not modal and appears to have somewhat less expressive power (e.g., no impredicative quantification, no embedding of semantic assertions in programs,

no machine-checked soundness proof).

Sadrzadeh has encoded a modal linear logic in Coq using the deep embedding approach [26]. This logic lacks quantification, and although it is also a multimodal logic, all the modalities are assumed to be S4 (reflexive and transitive). Using her embedding, Sadrzadeh demonstrates proofs of classic puzzles in epistemic logic.

Coupet-Grimal has developed a linear temporal logic,[7] which has been used to specify and prove correct a nontrivial garbage collection algorithm [10,11]. This logic is similar to ours in that it is a shallow embedding via predicates, and that it uses modalities to reason about time. However, Coupet-Grimal uses coinductive definitions to reason about an infinite time-line as opposed to our "count-down until the end of the universe" approach. One could easily imagine adding a separation logic component to this logic via the same joinability construction we use. It is not clear, however, that it would be possible to carry out the same type of stratification construction we used to build resource maps in a model with an infinite time-line.

9 Conclusion

In this paper we have presented a number of related techniques that combine to form a new system for reasoning about realistic programming languages in Coq. This system, in contrast to the now-standard syntactic subject-reduction approach popularized by Wright and Felleisen [28], is characterized by a semantically defined specification logic which is used to reason about small-step operational semantics.

The benefits of our system include: an effortless embedding of higher-order logic, including impredicative quantification; a versatile fixpoint operator based on a generic notion of world aging; powerful tools for reasoning about programs with general pointers via separation logic; a flexibility which derives from the fact that new logical operators and inference rules can be added at any time without disturbing preexisting lemmas.

A powerful testament to the strength of our techniques is their success in a real-world proof: the soundness proof for the operational semantics of Concurrent Cminor [15]. At approximately 50,000 lines of Coq code, the proof is a sizable software-engineering artifact. More importantly, however, the proof is an interesting result about a realistic programming language. As we increasingly used modal techniques in this proof, it went from infeasible to feasible, then became smaller and more manageable. As we continue to rewrite this proof, we expect that an increased and more consistent use of modal techniques will continue to simplify and modularize the proof, decreasing its size and increasing its elegance.

References

[1] A. W. Appel and S. Blazy. Separation logic for small-step C minor. In *20th International Conference on Theorem Proving in Higher-Order Logics (TPHOLs 2007)*, 2007.

[2] A. W. Appel and X. Leroy. A list-machine benchmark for mechanized metatheory. Technical Report RR-5914, INRIA, May 2006.

[7] Despite its name, Coupet-Grimal's logic is not a linear logic; the "linear" refers to the linear-time temporal model to differentiate it from branching-time models.

[3] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. on Programming Languages and Systems*, 23(5):657–683, Sept. 2001.

[4] A. W. Appel, P.-A. Melliès, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *Proc. 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 109–122, Jan. 2007.

[5] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, August 2005.

[6] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[7] B. Beckert and A. Platzer. Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In U. Furbach and N. Shankar, editors, *Proceedings, International Joint Conference on Automated Reasoning, Seattle, USA*, LNCS 4130, pages 266–280. Springer, 2006.

[8] G. Boolos. *The Logic of Provability*. Cambridge University Press, Cambridge, England, 1993.

[9] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic as modal logic: Completeness and parametric inexpressivity. In *POPL '07*, pages 123–134, 2007.

[10] S. Coupet-Grimal. An axiomatization of linear temporal logic in the calculus of inductive constructions. *Journal of Logic and Computation*, 13(6):801–813, 2003.

[11] S. Coupet-Grimal and C. Nouvet. Formal verification of an incremental garbage collector. *Journal of Logic and Computation*, 13(6):815–833, 2003.

[12] R. Goldblatt. Mathematical modal logic: A view of its evolution. In D. M. Gabby and J. Woods, editors, *Logic and the Modalities in the Twentieth Century*, chapter 1. Elsevier, Amsterdam, 2006.

[13] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *Proceedings 5th Asian Symposium on Programming Languages and Systems (APLAS'07)*, 2007.

[14] J. Y. Halpern and Y. Moses. A guide to completeness and complexity for modal logics of knowledge and belief. *Artifical Intelligence*, 54:311–379, 1992.

[15] A. Hobor, A. W. Appel, and F. Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Proc. European Symp. on Programming (ESOP 2008)*, 2008. to appear.

[16] G. E. Hughes and M. J. Cresswell. *A Companion to Modal Logic*. Routledge, London, UK, 1984.

[17] G. E. Hughes and M. J. Cresswell. *A New Introduction to Modal Logic*. Routledge, London, UK, 1996.

[18] S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *POPL 2001: The 28th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 14–26. ACM Press, Jan. 2001.

[19] S. A. Kripke. Semantical considerations on modal logic. In *Proceedings of a Colloquium: Modal and Many Valued Logics*, volume 16, pages 83–94, 1963.

[20] T. Murphy, K. Crary, R. Harper, and F. Pfenning. A symmetric modal lambda calculus for distributed computing. In *IEEE Symposium on Logic in Computer Science*, pages 286–297, Turku, Finland, July 2004.

[21] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL'01: Annual Conference of the European Association for Computer Science Logic*, pages 1–19, Sept. 2001. LNCS 2142.

[22] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1):271–307, May 2007.

[23] V. R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proceedings of the 17th Annual IEEE Symposium on Foundations of Computer Science*, pages 109–121, 1976.

[24] V. R. Pratt. Application of modal logic to programming. *Studia Logica*, 39:257–274, 1980.

[25] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002: IEEE Symposium on Logic in Computer Science*, pages 55–74, July 2002.

[26] M. Sadrzadeh. Modal linear logic in higher order logic: An experiment with Coq. In *Emerging Trends TPHOLS '03*, pages 75–93, 2003.

[27] G. Tan and A. W. Appel. A compositional logic for control flow. In *7th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'06)*, pages 80–94, Jan. 2006.

[28] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report COMP TR91-160, Rice University, April 1991.