INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Separation Logic for Small-step Cminor (extended version)

Andrew W. Appel — Sandrine Blazy

N° 6138 — version 4

version initiale Mars 2007 — version révisée Aout 2007

Theme SYM

ISSN 0249-6399 ISRN INRIA/RR--6138--FR+ENG

Separation Logic for Small-step Cminor (extended version)

Andrew W. Appel*†, Sandrine Blazy‡†

Thème SYM — Systèmes symboliques
Projets Gallium et Moscova

Rapport de recherche n° 6138 — version 4§ — version initiale Mars 2007 — version révisée Aout 2007 — 31 pages

Abstract: Cminor is a mid-level imperative programming language (just below C), and there exist proved-correct optimizing compilers from C to Cminor and from Cminor to machine language. We have redesigned Cminor so that it is suitable for Hoare Logic reasoning, we have designed a Separation Logic for Cminor, we have given a small-step operational semantics so that extensions to concurrent Cminor will be possible, and we have a machine-checked proof of soundness of our Separation Logic. This is the first large-scale machine-checked proof of a Separation Logic w.r.t. a small-step semantics, or for a language with nontrivial reducible control-flow constructs. Our sequential soundness proof of the sequential Separation Logic for the sequential language features will be reusable change within a soundness proof of Concurrent Separation Logic w.r.t. Concurrent Cminor. In addition, we have a machine-checked proof of the relation between our small-step semantics and Leroy's original big-step semantics; thus sequential programs can be compiled by Leroy's compiler with formal end-to-end correctness guarantees.

Key-words: program proof, formal semantics, imperative language, separation logic

* Princeton University; supported in part by NSF Grant CCF-0540914
† This work was done, in part, while both authors were on sabbatical at INRIA
‡ ENSIIE
§ Propagated the changes resulting from the reviewers of the companion paper

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Une logique de séparation pour le langage Cminor (version étendue)

Résumé : Cminor est un langage de bas niveau proche de C, utilisé comme langage intermédiaire dans un compilateur optimisant du langage C certifié en Coq. Ce rapport présente le développement en Coq d'une sémantique axiomatique pour Cminor ainsi que les preuves de correction associées. Cette sémantique constitue un premier lien entre la preuve de programmes et certification de compilateurs. Elle utilise la logique de séparation pour raisonner sur la mémoire et les pointeurs. Elle est définie selon un style à petits pas afin de pouvoir enrichir dans le futur le langage Cminor par des traits concurrents. Cette sémantique est également une sémantique à continuations, ce qui a facilité les preuves de correction.

Mots-clés : sémantique formelle, langage impératif, preuve de programmes, logique de séparation

Cminor is a mid-level imperative programming language (just below C), and there exist proved-correct optimizing compilers from C to Cminor and from Cminor to machine language. We have redesigned Cminor so that it is suitable for Hoare Logic reasoning, we have designed a Separation Logic for Cminor, we have given a small-step operational semantics so that extensions to concurrent Cminor will be possible, and we have a machine-checked proof of soundness of our Separation Logic. This is the first large-scale machine-checked proof of a Hoare Logic w.r.t. a small-step semantics. Our sequential soundness proof of the sequential Separation Logic for the sequential language features will be usable without change within a soundness proof of Concurrent Separation Logic w.r.t. Concurrent Cminor. In addition, we have a machine-checked proof of the relation between our small-step semantics and Leroy's original big-step semantics; thus sequential programs can be compiled by Leroy's compiler with formal end-to-end correctness guarantees.

1 Introduction

The future of program verification is to connect machine-verified source programs to machine-verified compilers, and run the object code on machine-verified hardware. To connect the verifications end to end, the source language should be specified as a structural operational semantics (SOS) represented in a logical framework; the target architecture can also be specified that way. Proofs of source code can be done in the logical framework, or by other tools whose soundness is proved w.r.t. the SOS specification; these may be in safety proofs via type-checking, correctness proofs via Hoare Logic, or (in source languages designed for the purpose) correctness proofs by a more expressive proof theory. The compiler—if it is an optimizing compiler—will be a stack of phases, each with a well specified SOS of its own. There will be proofs of (partial) correctness of each compiler phase, or witness-driven recognizers for correct compilations, w.r.t. the SOS's that are inputs and outputs to the phases.

Machine-verified hardware/compiler/application stacks have been built before. Moore described a verified compiler for a "high-level assembly language" [1]. Leinenbach *et al.* have built a compiler for *C0*, a small C-like language, and have demonstrated correctness (machine-checked proofs largely complete) of source programs, Hoare Logic, compiler, microkernel, and RISC processor [2]. These are both simple one- or two-pass nonoptimizing compilers.

Leroy [3] has built and proved correct in Coq [4] a compiler called *CompCert* from a high-level intermediate language *Cminor* to assembly language for the Power PC architecture. This compiler has 5 phases (with 4 intermediate languages), allowing for optimizations at several natural levels of abstraction. Blazy *et al.* have built and proved correct a translator from a subset of C to Cminor [5]. Another compiler phase on top (not yet implemented) will then yield a proved-correct compiler from C to machine language. We should therefore reevaluate the conventional wisdom that an entire practical optimizing compiler cannot be proved correct.

A software system is often a mixture of components written in different languages—the run-time system of an ML system is often written in C—and we would like end-to-end correctness proofs of the whole system. For this, we propose a new variant of Cminor as a machine-independent intermediate language that can serve as a common denominator between several high-level languages. In this report, we will show that Cminor has a usable Hoare Logic, so that correctness proofs for some components can be done directly at the level of Cminor.

Cminor has a "calculus-like" view of local variables and procedures (*i.e.* local variables are bound in an environment), while C0 has a "storage-allocation" view (*i.e.* local variables are stored in the stack frame). The calculus-like view will lead to easier reasoning about program transformations and easier use of Cminor as a target language, and fits naturally with a multi-pass optimizing compiler such as Leroy's; the storage-allocation view suits the one-pass nonoptimizing C0 compiler and can accommodate in-line assembly code.

Therefore we consider Cminor a promising candidate as a common intermediate language for end-to-end correctness proofs. But we have many demands on our new variant of Cminor, only the first three of which are satisfied by Leroy's Cminor.

- Cminor has an operational semantics represented in a logical framework.
- There is a proved-correct compiler from Cminor to machine language.
- Cminor is usable as the high-level target language of a C compiler.
  - Our semantics is a *small-step* semantics (SOS), to support reasoning about input/output, concurrency, and nontermination.
  - Cminor is machine-independent over machines in the "standard model" (*i.e.* 32- or 64-bit single-address-space byte-addressable multiprocessors).
  - Cminor can be used as a mid-level target language of an ML compiler [6], or of an OO-language compiler, so that we can integrate correctness proofs of ML or OO programs with the proofs of their run-time systems and low-level libraries.
  - As we show in this report, Cminor supports an axiomatic Hoare Logic (in fact, Separation Logic), proved sound with respect to the SOS, for reasoning about low-level (C-like) programs so we can prove correctness of the run-time systems and low-level libraries.
  - In future work, we plan to extend Cminor to be concurrent in the "standard model" of thread-based preemptive lock-synchronized weakly consistent shared-memory programming. We aim at reusing our soundness proofs for the sequential part of the language in a concurrent setting.

Leroy's original Cminor had several Power-PC dependencies, is slightly clumsy to use as the target of an ML compiler, and is a bit clumsy to use in Hoare-style reasoning. But most

important, because Leroy's SOS is a big-step semantics, it can be used only to reason about terminating sequential programs.[1]

We have redesigned Cminor's syntax and semantics to achieve all of these goals. That part of the redesign to achieve target-machine portability was done by Leroy himself. Our redesign to ease its use as an ML back end and for Hoare Logic reasoning was fairly simple. In the sequel of this report, Cminor will refer to the new version of the Cminor language. The main contributions of this report are:

- A sequential small-step SOS suitable for compilation and for Hoare Logic.
- A machine-checked proof of soundness of our sequential Hoare Logic of Separation (Separation Logic) w.r.t. our small-step semantics. Schirmer [8] has a machine-checked *big-step* Hoare-Logic soundness proof for a control flow much like ours, extended by Klein *et al.* [9] to a C-like memory model. Ni and Shao [10] have a machine-checked proof of soundness of a Hoare-like logic w.r.t. a small-step semantics, but for an assembly language and for much simpler assertions than ours.
- A machine-checked big-step/small-step equivalence proof that allows us to use Leroy's existing proved-correct Cminor compiler for (terminating sequential) programs proved correct in Separation Logic.

2 Big-step Expression Semantics

The C standard [11] describes a memory model that is byte- and word-addressable (yet portable to big-endian and little-endian machines) with a nontrivial semantics for uninitialized variables. Blazy and Leroy formalized this model [12] for the semantics of Cminor. In C, pointer arithmetic within any malloc'ed block is defined, but pointer arithmetic between different blocks is undefined; Cminor therefore has non-null pointer values comprising an abstract block-number and an int offset. A `NULL` pointer is represented by the integer value 0. Pointer arithmetic between blocks, and reading uninitialized variables, are undefined but not illegal: expressions in Cminor can evaluate to *undefined* (Vundef) without getting stuck.

Each memory load or store is to a non-null pointer value (*i.e.* a block and an offset) with a "chunk" descriptor *ch* specifying number of bytes, signed or unsigned, int or float. Storing as 32-bit-int then loading as 8-bit-signed-byte leads to an undefined value. Load and store operations on memory, $m \vdash v_1 \overset{ch}{\mapsto} v_2$ and $m' = m[v_1 \overset{ch}{:=} v_2]$, are partial functions that yield results only if reading (resp., writing) a chunk of type *ch* at address v_1 is legal. We write $m \vdash v_1 \overset{ch}{\mapsto} v$ to mean that the result of loading from memory m at address v_1 a chunk-type *ch* is the value v.

[1] Leroy has experimented in Coq coinductive big-step semantics for simple languages, that describe non-terminating executions of programs in addition to terminating executions [7]. But proving semantic preservation properties such as those required in a certified compiler is difficult, especially for a language such as Cminor. Thus, for the time being, it has been decided to avoid the use of coinductive big-step semantics in CompCert. Furthermore, coinductive big-step semantics are not adapted to concurrency.

The *values* of Cminor are *undefined* (Vundef), integers, pointers, and floats. The int type is an abstract data-type of 32-bit modular arithmetic. The expressions of Cminor are literals, variables, primitive operators applied to arguments, and memory loads.

$$\begin{aligned} i &: \mathsf{int} ::= [0, 2^{32}) \\ v &: \mathsf{val} ::= \mathsf{Vundef} \mid \mathsf{Vint}\ (i) \mid \mathsf{Vptr}\ (b, i) \mid \mathsf{Vfloat}\ (f) \\ e &: \mathsf{expr} ::= \mathsf{Eval}\ (v) \mid \mathsf{Evar}\ (id) \mid \mathsf{Eop}\ (op, el) \mid \mathsf{Eload}\ (ch, e) \\ el &: \mathsf{exprlist} ::= \mathsf{Enil} \mid \mathsf{Econs}\ (e, el) \end{aligned}$$

There are 33 primitive operation symbols *op*; two of these are for accessing global names and local stack-blocks, and the rest is for integer and floating-point arithmetic and comparisons. For instance, the boolean negation operator Oneg [2] is used in expressions such as Eop (Oneg, Econs (e, Enil)). In the sequel of this report, we will use the C notation !e for such an expression.

Cminor has an infinite supply ident of variable and function identifiers *id*. As in C, there are two namespaces—each *id* can be interpreted in a local scope (using Evar (id)) or in a global scope (using the operation symbol for accessing global names).

Expression evaluation in Leroy's Cminor is expressed by an inductive big-step relation. Big-step statement execution is problematic for concurrency, but big-step *expression* evaluation is fine—as long as we prove noninterference—and has the advantage of simplicity.

Evaluation is deterministic. Leroy chose to represent evaluation as a relation because Coq had better support for proof induction over relations than over function definitions. We have chosen to represent evaluation as a partial function; this makes some proofs easier in some ways: $f(x) = f(x)$ is simpler than $f\,x\,y \Rightarrow f\,x\,z \Rightarrow y = z$. We have developed a tactical technique for proofs over functions with case analysis. [3] Although we specify expression evaluation as a function in Coq, we present evaluation as a judgment relation in Figure 1. Our evaluation function is (proved) equivalent to the inductively defined judgment $\Psi; (sp; \rho; \phi; m) \vdash e \Downarrow v$ where:

Ψ is the "program," consisting of a global environment (ident $\rightarrow$ option block) [4] mapping identifiers to function-pointers and other global constants, and a global mapping (block $\rightarrow$ option function) that maps certain ("text-segment") addresses to function definitions.

sp : block. The "stack pointer" giving the address of the memory block for stack-allocated local data in the current activation record.

ρ : env. The local environment, a finite mapping from identifiers to values.

[2] The Oneg operator doesn't exist in Cminor but it can be defined from more primitive operators.

[3] Our tactic was written in a previous version of Coq. It can be considered as an instantiation of the new tactic called functional induction.

[4] In a global environment, an identifier is mapped either to Some b if there exists an address b for this identifier, or to None.

$$\Psi;(sp;\rho;\phi;m) \vdash \mathsf{Eval}\,(v) \;\Downarrow v \qquad \frac{x \in \mathrm{dom}\,\rho}{\Psi;(sp;\rho;\phi;m) \vdash \mathsf{Evar}\,(x) \;\Downarrow \rho(x)}$$

$$\frac{\Psi;(sp;\rho;\phi;m) \vdash el \Downarrow vl \qquad \Psi;sp \vdash op(vl) \Downarrow_{\mathsf{eval\_operation}} v}{\Psi;(sp;\rho;\phi;m) \vdash \mathsf{Eop}\,(op, el) \;\Downarrow v}$$

$$\frac{\Psi;(sp;\rho;\phi;m) \vdash e_1 \Downarrow v_1 \qquad \phi \vdash \mathsf{load}_{ch}\,v_1 \qquad m \vdash v_1 \overset{ch}{\mapsto} v}{\Psi;(sp;\rho;\phi;m) \vdash \mathsf{Eload}\,(ch, e_1) \;\Downarrow v}$$

Figure 1: Expression evaluation rules

ϕ : footprint. A mapping from memory addresses to permissions. Leroy's Cminor has no footprints, as these are needed only for Separation Logic reasoning.

m : mem. The memory, a finite mapping from block to block_contents [12]. Each block represents the result of a C `malloc` or a stack frame, a global static variable, or a function code-pointer. A block content consists of the dimensions of the block (low and high bounds) plus a mapping from byte offsets to byte-sized memory cells.

e : expr. The expression being evaluated.

v : val. The value of the expression.

The footprint ϕ is a mapping from memory addresses to permissions. Loads outside the footprint will cause expression evaluation to get stuck. Since the footprint may have different permissions for loads than for stores to some addresses, we write $\phi \vdash \mathsf{load}_{ch}\,v$ (or $\phi \vdash \mathsf{store}_{ch}\,v$) to mean that all the addresses from v to $v+|ch|-1$ are readable (or writable).

To model the possibility of exclusive read/write access or shared read-only access, we write $\phi_0 \oplus \phi_1 = \phi$ for the "disjoint" sum of two footprints, where $\oplus$ is an associative and commutative operator with several properties such as $\phi_0 \vdash \mathsf{store}_{ch}\,v \;\Rightarrow\; \phi_1 \not\vdash \mathsf{load}_{ch}\,v$, $\phi_0 \vdash \mathsf{load}_{ch}\,v \;\Rightarrow \phi \vdash \mathsf{load}_{ch}\,v$ and $\phi_0 \vdash \mathsf{store}_{ch}\,v \;\Rightarrow \phi \vdash \mathsf{store}_{ch}\,v$. One can think of ϕ as a set of fractional permissions [13], with 0 meaning no permission, $0 < x < 1$ permitting read, and 1 giving read/write permission. A store permission can be split into two or more load permissions, which can be reconstituted to obtain a store permission. Parkinson [14, Ch. 5] defines a more sophisticated and general model. Either of those models can be used with our Separation Logic, but our initial prototype proof uses a simpler model without fractional permissions.

Most previous models of Separation Logic (*e.g.*, Ishtiaq and O'Hearn [15]) represent heaps as partial functions that can be combined with an operator like $\oplus$. Of course, a partial function can be represented as a pair of a domain set and a total function. Similarly,

we represent heaps as a footprint plus a Cminor memory. We do this for compatibility with Leroy's Cminoroperational semantics; carrying footprints separately from memories does not add any particular difficulty to the soundness proofs for our Separation Logic.

To perform arithmetic and other operations, in the third rule of Figure 1, the judgment $\Psi; sp \vdash op(vl) \Downarrow_{\mathsf{eval\_operation}} v$ takes an operator op applied to a list value vl and (if vl contains appropriate values) produces some value v. The operators for accessing global names and local stack-blocks make use of Ψ and sp respectively to return the global meaning of an identifier or the local stack-block address.

States. We shall bundle together $(sp; \rho; \phi; m)$ and call it the *state*, written as σ. We write $\Psi; \sigma \vdash e \Downarrow v$ to mean $\Psi_\sigma; (sp_\sigma; \rho_\sigma; \phi_\sigma; m_\sigma) \vdash e \Downarrow v$.

Notation. We write $\sigma[:= \rho']$ to mean the state σ with its environment component ρ replaced by ρ', and so on (*e.g.* see Figure 3).

Fact. $\Psi; sp \vdash op(vl) \Downarrow_{\mathsf{eval\_operation}} v$ and $m \vdash v_1 \overset{ch}{\mapsto} v$ are both deterministic relations, *i.e.* functions.

Lemma 1. $\Psi; \sigma \vdash e \Downarrow v$ is a deterministic relation. (Trivial by inspection.)

Lemma 2. For any value v, there is an expression e such that $\forall \sigma.\ (\Psi; \sigma \vdash e \Downarrow v)$.

Proof. Obvious; e is simply $\mathsf{Eval}\, v$. But it is important nonetheless: reasoning about programs by rewriting and by Hoare Logic often requires this property, and it was absent from Leroy's Cminor for Vundef and Vptr values.

An expression may fetch from several different memory locations, or from the same location several times. Because $\Downarrow$ is deterministic, we cannot model a situation where the memory is updated by another thread after the first fetch and before the second. This is deliberate. But on the other hand, we want a semantics that describes real executions on real machines. The solution is to evaluate expressions in a setting where we can guarantee *noninterference*. We will do this (in our extension to Concurrent Cminor) by guaranteeing that the footprints ϕ of different threads are disjoint.

The Cminor compiler (CompCert) is proved correct with respect to an operational semantics that does not use footprints. Any program that successfully evaluates with footprints will also evaluate ignoring footprints. Thus, it is sound to prove properties in a footprint semantics and compile in an erased semantics. We formalize this as follows.

Definition 1 (Erased expression evaluation). Let $\Psi; (sp; \rho; m) \vdash e \dot{\Downarrow} v$ be the relation defined by rules just like those for $e \Downarrow v$ except that ϕ is removed, as is the premise $\phi \vdash \mathsf{load}_{ch}\, v_1$ of the rule for Eload expressions.

Definition 2 (Erased expression state). We define an *erased state* $\dot{\sigma}$ as (Ψ, sp, ρ, κ); that is, just like a sequential state but without ϕ.

Lemma 3. If $\Psi;\sigma \vdash e \Downarrow v$ then $\Psi;\dot{\sigma} \vdash e\dot{\Downarrow}v$.

Proof. Trivial; in the case for Eload it is strictly easier to derive $e\dot{\Downarrow}v$ than $e \Downarrow v$.

Lemma 4. If $\mathsf{pure}\,(e)$ then $\Psi;\dot{\sigma} \vdash e\dot{\Downarrow}v \;\Rightarrow\; \Psi;\sigma \vdash e \Downarrow v$.

Lemma 5. If $\Psi;\sigma \vdash e \Downarrow v$ and $\phi_\sigma \subset \phi'$ then $\Psi;\sigma[:=\phi'] \vdash e \Downarrow v$.

3 Small-step Statement Semantics

The statements of sequential Cminor are:

$$\begin{aligned} s : \mathsf{stmt} \quad ::= {} & x := e \mid [e_1]_{ch} := e_2 \mid \mathsf{loop}\, s \mid \mathsf{block}\, s \mid \mathsf{exit}\, n \\ & \mid \mathsf{call}\, xl\, e\, el \mid \mathsf{return}\, el \mid s_1; s_2 \mid \mathsf{if}\, e\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2 \mid \mathsf{skip}. \end{aligned}$$

The assignment $x := e$ puts the value of e into the variable x. The store $[e_1]_{ch}:=e_2$ puts (the value of) e_2 into the memory-chunk ch at address given by (the value of) e_1. Local variables are not addressable; global variables and heap locations are memory addresses. To model exits from nested loops, $\mathsf{block}\, s$ runs s, which should not terminate normally but which should $\mathsf{exit}\, n$ from the $(n+1)^{th}$ enclosing block, and $\mathsf{loop}\, s$ repeats s infinitely or until it returns or exits. $\mathsf{call}\, xl\, e\, el$ calls function e with parameters (by value) el and results returned back into the variables xl. [5] $\mathsf{return}\, el$ evaluates and returns a sequence of results, $(s_1; s_2)$ executes s_1 followed by s_2 (unless s_1 returns or exits), and the statements if and skip are as the reader might expect.

Combined with infinite loops and if statement, blocks and exits suffice to express efficiently all reducible control-flow graphs, notably those arising from C loops. The C statements break and continue are translated as appropriate exit statements. [5] details the translation of these C statements into Cminor.

Figure 2 shows the translation in Cminor of a general C while loop with continue and break statements in its body: the loop becomes a block consisting of an infinite loop, and the loop body becomes also a block. The statements continue and break are translated as appropriate exit constructs. A C while loop (written as $\mathsf{while}\, e\, s$) without continue and break statement in its body s is thus a Cminor loop written as $\mathsf{block}\,\{\mathsf{loop}\,\{\mathsf{if}\,(!e)\,\mathsf{then}\,\mathsf{exit}\,0\,\mathsf{else}\,s\}\}$. In the sequel of this report, we will use the notation $\mathsf{while}\, e\, s$ for such a loop. In Figure 2 we assume e is a C expression without side-effect, so it translates as itself.

[5] Each function in the real Cminor also has a signature that specifies the int/floatness of the parameters; our machine-checked proofs account for this but we shall omit it from this presentation.

```
while (e) {
    ...
    continue;
    ...
    break;
   ...
}
```

```
block {
  loop {
    if (! e) exit 0;
    block {
        ...
        exit 0;
        ...
        exit 1;
       ...
    }
  }
}
```

Figure 2: Translation of a C while loop into a Cminor infinite loop

Function definitions. A Cminor program Ψ is really two mappings: a mapping from function names to memory blocks (*i.e.*, abstract addresses), and a mapping from memory blocks to function definitions. Each function definition $f = (xl, yl, n, s)$, where $\mathsf{params}(f) = xl$ is a list of formal parameters, $\mathsf{locals}(f) = yl$ is a list of local variables, $\mathsf{stackspace}(f) = n$ is the size of the local stack-block to which sp points, and the statement $\mathsf{body}(f) = s$ is the function body.

Operational semantics. Our small-step semantics for statements is based on continuations, mainly to allow a uniform representation of statement execution that facilitates the design of lemmas. Such a semantics also avoids all search rules (congruence rules) and simplifies reasoning in the soundness proof for the Hoare Logic (and for the compiler). In section 7 we will describe the big-step semantics and prove equivalence of the two semantics (for programs that terminate).

Definition 3 (Continuation). A continuation k has a state σ and a control stack κ. There are sequential control operators to handle local control flow (Kseq written as $\cdot$), intraprocedural control flow (Kblock), and function-return (Kcall); this last carries not only a control aspect but an activation record of its own. The control operator Kstop represents the safe termination of the computation.

$$\begin{aligned} \kappa : \mathsf{control} &::= \mathsf{Kstop} \mid s \cdot \kappa \mid \mathsf{Kblock}\,\kappa \mid \mathsf{Kcall}\,xl\,f\,sp\,\rho\,\kappa \\ k : \mathsf{continuation} &::= (\sigma, \kappa) \end{aligned}$$

The sequential small-step function takes the form $\Psi \vdash k \longmapsto k'$ (see Figure 3), and we define as usual its reflexive transitive closure $\longmapsto^*$. As in C, there is no boolean type in Cminor. In Figure 3, the predicate $\mathsf{is\_true}\,v$ (resp. $\mathsf{is\_false}\,v$) interprets a value as true

(resp. false).[6] A store statement $[e_1]_{ch}{:=}e_2$ requires the corresponding store permission $\phi_\sigma \vdash \mathsf{store}_{ch}\, v_1$.

Given a control stack $\mathsf{block}\, s \cdot \kappa$, the small-step execution of the block statement $\mathsf{block}\, s$ enters that block: s becomes the next statement to execute and the control stack becomes $s \cdot \mathsf{Kblock}\, \kappa$.

Exit statements are only allowed from blocks that have been previously entered. For that reason, in the two rules for exit statements, the control stack ends with $(\mathsf{Kblock}\, \kappa)$ control. A statement $(\mathsf{exit}\, n)$ terminates the $(n+1)^{th}$ enclosing block statements. In such a block, the stack of control sequences $s_1 \cdots s_j$ following the exit statement is not executed. Let us note that this stack may be empty if the exit statement is the last statement of the most enclosing block. The small-step execution of a statement $(\mathsf{exit}\, n)$ exits from only one block (the most enclosing one). Thus, the execution of an $(\mathsf{exit}\, 0)$ statement updates the control stack $(\mathsf{exit}\, 0 \cdot s_1 \cdot \cdots \cdot s_j \cdot \mathsf{Kblock}\, \kappa)$ into κ. The execution of an $(\mathsf{exit}\ n+1)$ statement updates the control stack $(\mathsf{exit}\, (n+1) \cdot s_1 \cdot \cdots \cdot s_j \cdot \mathsf{Kblock}\, \kappa)$ into $\mathsf{exit}\, n \cdot \kappa$.

The small-step execution of a call statement allocates the memory required in order to execute the called function. It also initializes formal parameters to their corresponding actual parameters and local variables to the Vundef value. It steps to the body of the callee.

As for exit and block statements, a return statement must be related to a call statement. Thus, in the return rule, the control stack κ must terminate by a Kcall control operator that is compatible with the list el_{args} of the $\mathsf{return}\, el_{\text{args}}$ statement. The control operators that precede this Kcall correspond to statements that are not executed since they follow the return statement in the body of the called function. The allocated memory is freed at the end of the execution of the return statement.

The small-step execution of a program Ψ is of the form $\Psi \vdash (\sigma_0, s_0 \cdot \mathsf{Kstop}) \longmapsto^* (\sigma, \mathsf{Kstop})$ where s_0 is a call statement to the "main" function of Ψ.

Lemma 6. If $\Psi; \sigma \vdash e \Downarrow v$ then $\Psi \vdash (\sigma, (x := e) \cdot \kappa) \longmapsto k'$ iff $\Psi \vdash (\sigma, (x := \mathsf{Eval}\, v) \cdot \kappa)) \longmapsto k'$ (and similarly for other statement forms containing expressions or expression lists).

Proof. Trivial: expressions have no side effects. A convenient property nonetheless, and not true of Leroy's original Cminor.

Definition 4 (Stuck continuation). A continuation $k = (\sigma, \kappa)$ is *stuck* if $\kappa \neq \mathsf{Kstop}$ and there does not exist k' such that $\Psi \vdash k \longmapsto k'$.

Definition 5 (Safe continuation). A continuation k is *safe* (written as $\Psi \vdash \mathsf{safe}(k)$) if it cannot reach a stuck continuation in the sequential small-step relation $\longmapsto^*$.

[6] True values are non null integer values and pointer values. Only the integer 0 is interpreted as a false value.

$$\Psi \vdash (\sigma,\ (s_1; s_2)\cdot\kappa) \longmapsto (\sigma, s_1 \cdot s_2 \cdot \kappa) \qquad \frac{\Psi;\sigma \vdash e \Downarrow v \quad \rho' = \rho_\sigma[x := v]}{\Psi \vdash (\sigma,\ (x := e)\cdot\kappa) \longmapsto (\sigma[:= \rho'], \kappa)}$$

$$\frac{\Psi;\sigma \vdash e_1 \Downarrow v_1 \qquad \Psi;\sigma \vdash e_2 \Downarrow v_2 \qquad \phi_\sigma \vdash \mathsf{store}_{ch}\, v_1 \qquad m' = m_\sigma[v_1 \overset{ch}{:=} v_2]}{\Psi \vdash (\sigma,\ ([e_1]_{ch} := e_2)\cdot\kappa) \longmapsto (\sigma[:= m'], \kappa)}$$

$$\frac{\Psi;\sigma \vdash e \Downarrow v \qquad \mathsf{is\_true}\, v}{\Psi \vdash (\sigma,\ (\mathsf{if}\, e\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2)\cdot\kappa) \longmapsto (\sigma, s_1 \cdot \kappa)}$$

$$\frac{\Psi;\sigma \vdash e \Downarrow v \qquad \mathsf{is\_false}\, v}{\Psi \vdash (\sigma,\ (\mathsf{if}\, e\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2)\cdot\kappa) \longmapsto (\sigma, s_2 \cdot \kappa)} \qquad \Psi \vdash (\sigma, \mathsf{skip}\cdot\kappa) \longmapsto (\sigma,\kappa)$$

$$\Psi \vdash (\sigma,\ (\mathsf{loop}\, s)\cdot\kappa) \longmapsto (\sigma, s \cdot \mathsf{loop}\, s \cdot \kappa) \qquad \Psi \vdash (\sigma, (\mathsf{block}\, s)\cdot\kappa) \longmapsto (\sigma, s \cdot \mathsf{Kblock}\,\kappa)$$

$$\frac{j \geq 1}{\Psi \vdash (\sigma, \mathsf{exit}\, 0 \cdot s_1 \cdot \cdots s_j \cdot \mathsf{Kblock}\,\kappa) \longmapsto (\sigma,\kappa)}$$

$$\frac{j \geq 1}{\Psi \vdash (\sigma, \mathsf{exit}\, (n+1) \cdot s_1 \cdot \cdots s_j \cdot \mathsf{Kblock}\,\kappa) \longmapsto (\sigma, \mathsf{exit}\, n \cdot \kappa)}$$

$$\frac{\begin{array}{cc}\Psi;\sigma \vdash e_{\mathrm{fun}} \Downarrow v_{\mathrm{fun}} & \Psi;\sigma \vdash el_{\mathrm{args}} \Downarrow vl_{\mathrm{args}} \\ \Psi_\sigma(v_{\mathrm{fun}}) = f & \mathsf{alloc}(m_\sigma, \mathsf{stackspace}(f)) = (m', sp') \\ \rho' = [\mathsf{params}(f) \mapsto vl_{\mathrm{args}}][\mathsf{locals}(f) \mapsto \mathsf{Vundef}] & \phi' = \phi_\sigma \oplus [sp', sp' + \mathsf{stackspace}(f))\end{array}}{\Psi \vdash (\sigma,\ \mathsf{call}\, xl\, e_{\mathrm{fun}}\, el_{\mathrm{args}}\cdot\kappa) \longmapsto (\sigma[:= sp', \rho', \phi', m'], \mathsf{body}(f)\cdot \mathsf{Kcall}\, xl\, f\, sp_\sigma\, \rho_\sigma\, \kappa)}$$

$$\frac{\begin{array}{cc}\kappa = \textit{any sequence of}\ \cdot\ \textit{and}\ \mathsf{Kblock}\ \textit{operators terminating in}\ \mathsf{Kcall}\, xl\, f\, sp'\, \rho'\, \kappa' & \\ \Psi;\sigma \vdash el_{\mathrm{args}} \Downarrow vl_{\mathrm{args}} & |\mathsf{params}(f)| = |vl_{\mathrm{args}}| \\ \rho'' = \rho'[xl := vl_{\mathrm{args}}] \qquad \phi' = \phi_\sigma \backslash [sp_\sigma, sp_\sigma + \mathsf{stackspace}(f)) & \mathsf{free}(m_\sigma, sp_\sigma) = m'\end{array}}{\Psi \vdash (\sigma,\ \mathsf{return}\, el_{\mathrm{args}}\cdot\kappa) \longmapsto (\sigma[:= sp', \rho'', \phi', m'], \kappa')}$$

Figure 3: Sequential small-step relation

Erasure. We define an *erased continuation* as a pair $(\dot{\sigma}, \kappa)$. We define the *erased small-step relation* $\dot{\longmapsto}$ derived by erasing from the $\longmapsto$ any mention of ϕ and any premises depending on ϕ.

Lemma 7 (Erasure). If $\Psi \vdash k \longmapsto k'$ then $\Psi \vdash k \dot{\longmapsto} k'$.

Proof. The ϕ's can at most cause a computation to get stuck; they never affect the contents of memories or environments.

$$
\begin{aligned}
\mathbf{emp} &=_{\text{def}} \lambda\Psi\sigma.\ \phi_\sigma = \emptyset \\
P * Q &=_{\text{def}} \lambda\Psi\sigma.\ \exists\phi_1.\exists\phi_2.\ \ \phi_\sigma = \phi_1 \oplus \phi_2 \ \wedge\ P(\sigma[:=\phi_1]) \ \wedge\ Q(\sigma[:=\phi_2]) \\
P \vee Q &=_{\text{def}} \lambda\Psi\sigma.\ P\sigma \vee Q\sigma \\
P \wedge Q &=_{\text{def}} \lambda\Psi\sigma.\ P\sigma \wedge Q\sigma \\
P \Rightarrow Q &=_{\text{def}} \lambda\Psi\sigma.\ P\sigma \Rightarrow Q\sigma \\
\neg P &=_{\text{def}} \lambda\Psi\sigma.\ \neg(P\sigma) \\
\exists z.P &=_{\text{def}} \lambda\Psi\sigma.\ \exists z.\, P\sigma \\
\lceil A \rceil &=_{\text{def}} \lambda\Psi\sigma.\ A \qquad \text{where } \sigma \text{ does not appear free in } A \\
\mathbf{true} &=_{\text{def}} \lceil \mathbf{True} \rceil \qquad\qquad \mathbf{false} =_{\text{def}} \lceil \mathbf{False} \rceil \\
e \Downarrow v &=_{\text{def}} \mathbf{emp} \wedge \lceil \mathsf{pure}(e) \rceil \wedge \lambda\Psi\sigma.\, (\Psi; \sigma \vdash e \Downarrow v) \\
\lceil e \rceil_{\mathsf{expr}} &=_{\text{def}} \exists v.\, e \Downarrow v \ast \lceil \mathsf{is\_true}\, v \rceil \\
\mathsf{defined}(e) &=_{\text{def}} \lceil e \overset{\text{int}}{==} e \rceil_{\mathsf{expr}} \vee \lceil e \overset{\text{float}}{==} e \rceil_{\mathsf{expr}} \\
e_1 \overset{ch}{\mapsto} e_2 &=_{\text{def}} \exists v_1.\exists v_2.(e_1 \Downarrow v_1) \wedge (e_2 \Downarrow v_2) \wedge (\lambda\sigma, m_\sigma \vdash v_1 \overset{ch}{\mapsto} v_2 \wedge \phi_\sigma \vdash \mathsf{store}_{ch}\, v_1) \wedge \mathsf{defined}(v_2)
\end{aligned}
$$

Figure 4: Main operators of Separation Logic

4 Separation Logic

Hoare Logic uses triples $\{P\}\, s\, \{Q\}$ where P is a precondition, s is a statement of the programming language, and Q is a postcondition. The assertions P and Q are predicates on the program state. The reasoning on memory is inherently global. Separation Logic is an extension of Hoare Logic for programs that manipulate pointers. In Separation Logic, reasoning is local [16]; assertions such as P and Q describe properties of part of the memory, and $\{P\}\, s\, \{Q\}$ describes changes to part of the memory. We prove the soundness of the Separation Logic via a shallow embedding, that is, we will give each assertion a semantic meaning in Coq. That is, we have $P, Q : \mathsf{assert}$ where $\mathsf{assert} = \mathsf{state} \to \mathsf{Prop}$. So $P\Psi\sigma$ is a proposition of logic and we say that σ satisfies P.

Assertion operators In Figure 4, we define the usual operators of Separation Logic: the empty assertion **emp**, separating conjunction $*$, disjunction $\vee$, conjunction $\wedge$, implication $\Rightarrow$, negation $\neg$, and quantifier $\exists$. A state σ satisfies $P * Q$ if its footprint ϕ_σ can be split into ϕ_1 and ϕ_2 such that $\sigma[:=\phi_1]$ satisfies P and $\sigma[:=\phi_2]$ satisfies Q. We also define some novel operators such as expression evaluation $e \Downarrow v$ and base-logic propositions $\lceil A \rceil$. Imprecise **true** and **false** are defined from the Coq propositions **True** and **False**.

O'Hearn and Reynolds specify Separation Logic for a little language in which expressions evaluate independently of the heap [16]. That is, their expressions access only the program

variables and do not even have *read* side effects on the memory. Memory reads (*i.e.* loads, written as $m_\sigma \vdash v_1 \overset{ch}{\mapsto} v_2$) are done by a command of the language, not within expressions. In Cminor we relax this restriction; expressions can read the heap. But we say that an expression is *pure* if (syntactically) it contains no Eload operators—which guarantees that it does not read the heap.

In Hoare Logic one can use expressions of the programming language as assertions—there is an implicit coercion. We write the assertion $e \Downarrow v$ (defined in Figure 4) to mean that expression e evaluates to value v in the operational semantics. This is an expression of Separation Logic, in contrast to $\Psi; \sigma \vdash e \Downarrow v$ which is a judgment in the underlying logic. In a previous experiment, our Separation Logic permitted impure expressions in $e \Downarrow v$. But, this complicated the proofs unnecessarily. Having $\mathbf{emp} \wedge \lceil \mathsf{pure}(e) \rceil$ in the definition of $e \Downarrow v$ leads to an easier-to-use Separation Logic.

Hoare Logic traditionally allows expressions e of the programming language to be used as expressions of the program logic. We will define explicitly $\lceil e \rceil_{\mathsf{expr}}$ to mean that e evaluates to a true value (*i.e.* a nonzero integer or non-null pointer). Following Hoare's example, we will usually omit the $\lceil \rceil_{\mathsf{expr}}$ braces in our Separation Logic notation.

Cminor's integer equality operator, which we will write as $e_1 \overset{\text{int}}{==} e_2$, applies to integers or pointers, but in several cases it is "stuck" (expression evaluation gives no result): when comparing a nonzero integer to a pointer;[7] when comparing Vundef or $\mathsf{Vfloat}(x)$ to anything. Thus we can write the assertion $\lceil e \overset{\text{int}}{==} e \rceil_{\mathsf{expr}}$ (or just write $e \overset{\text{int}}{==} e$) to test that e is a defined integer or pointer in the current state, and there is a similar operator $e_1 \overset{\text{float}}{==} e_2$. Finally, we have the usual Separation Logic singleton "maps-to", but annotated with a chunk-type ch : $e_1 \overset{ch}{\mapsto} e_2$ means that e_1 evaluates to v_1, e_2 evaluates to v_2 and at address v_1 in memory there is a defined value v_2 of the given chunk-type. Let us note that in this definition, $\mathsf{defined}(v_1)$ is implied by the third conjunct. $\mathsf{defined}(v_2)$ is a design decision. We could leave it out and have a slightly different Separation Logic.

Memory-supported assertions. In Separation Logic, some assertions are *supported* by a nonempty footprint; for example $a \mapsto b$ holds only on states whose footprint ϕ contains just exactly a. (More precisely, $e_1 \overset{ch}{\mapsto} e_2$ holds when ϕ is exactly the chunk ch starting at address a.) Assertion **emp** is supported exactly by the empty footprint. Assertions such as $\lceil x > 2 \rceil$ (where x is a variable of logic, not of Cminor), are supported by any footprint. Thus it makes sense to write $\exists x. \lceil x > 2 \rceil \wedge 10 \mapsto x$ to assert that $m[10] > 2$.

In one set of implementation experiments with semiautomatic tactics for manipulating Separation Logic assertions in Coq [17], we find it convenient to uniformly use $*$ as a conjunctive combinator, instead of arbitrary mixtures of $*$ and $\wedge$. There we might choose a convention in which $\lceil e \rceil$ is *always* written in the context $\mathbf{emp} \wedge \lceil e \rceil$; then we can write, for example, $(\mathbf{emp} \wedge \lceil x > 2 \rceil) * 10 \mapsto x$, which we syntactically sugar as $(x > 2) * 10 \mapsto x$.

[7] Integers may be compared for equality to integers, and pointers to pointers; and the NULL value, which is the integer 0, may be compared to any pointer, yielding false.

The Hoare sextuple. In Cminor there are commands that call functions, and commands that exit (from a block) or return (from a function). Thus, we extend the Hoare triple $\{P\}\, s\, \{Q\}$ with three extra contexts to become $\Gamma; R; B \vdash \{P\} s \{Q\}$ where:

Γ : assert describes context-insensitive properties of the global environment;

R : list val $\rightarrow$ assert is the *return environment*, giving the current function's postcondition as a predicate on the list of returned values; and

B : nat $\rightarrow$ assert is the *block environment* giving the exit conditions of each block statement in which the statement s is nested.

The rules of sequential Separation Logic are given in Figure 5. The rule for $[e]_{ch}{:=}e_1$ requires the same store permission than the small-step rule, but in Fig. 5, the permission is hidden in the definition of $e \overset{ch}{\mapsto} e_2$.

The rules for $[e]_{ch}{:=}e_1$ and if e then s_1 else s_2 require that e be a pure expression. To reason about an such statements where e is impure, one reasons by program transformation using the following rules. It is not necessary to rewrite the actual source program, it is only the local reasoning that is by program transformation.

$$\frac{x, y \text{ not free in } e, e_1, Q \qquad \Gamma; R; B \vdash \{P\}\, x := e;\ y := e_1;\ [x]_{ch}{:=}y\, \{Q\}}{\Gamma; R; B \vdash \{P\}[e]_{ch}{:=}e_1\{Q\}}$$

$$\frac{x \text{ not free in } s_1, s_2, Q \qquad \Gamma; R; B \vdash \{P\}\, x := e; \text{if } x \text{ then } s_1 \text{ else } s_2\, \{Q\}}{\Gamma; R; B \vdash \{P\} \text{ if } e \text{ then } s_1 \text{ else } s_2\, \{Q\}}$$

As a (loop s) statement is an infinite loop, we can write any assertion as postcondition of the (loop s) rule. We have chosen to write the **false** assertion. In the same way, any assertion can be the postcondition of an exit or return statement and we have written the **false** assertion in the corresponding rules.

The statement exit i exits from the $(i+1)^{th}$ enclosing block. A block environment B is a sequence of assertions $B_0, B_1, \ldots, B_{k-1}$ such that (exit i) is safe as long as the precondition B_i is satisfied. We write $\mathsf{nil_B}$ for the empty block environment and $B' = Q \cdot B$ for the environment such that $B'_0 = Q$ and $B'_{i+1} = B_i$. Given a block environment B, a precondition P and a postcondition Q, the axiomatic semantics of a (block s) statement consists in executing some statements of s given the same precondition P and the block environment $Q \cdot B$ (*i.e.* each existing block nesting is incremented). The last statement of s to be executed is an exit statement that yields the **false** postcondition. An (exit n) statement is only allowed from a corresponding enclosing block, *i.e.* the precondition $B(n)$ must exist in the block environment B and it is the precondition of the (exit n) statement.

A function precondition $P(vl_{\mathsf{args}})$ is parameterized by the function argument values, and a function postcondition $Q(vl_{\mathsf{results}})$ is parameterized by the function result values. P and

$$\frac{P \Rightarrow P' \qquad \Gamma; R; B \vdash \{P'\}s\{Q'\} \qquad Q' \Rightarrow Q}{\Gamma; R; B \vdash \{P\}s\{Q\}} \qquad \Gamma; R; B \vdash \{P\}\mathsf{skip}\{P\}$$

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

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

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

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

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

$$\Gamma; R; B \vdash \{B(n)\}\mathsf{exit}\, n\{\mathbf{false}\} \qquad \Gamma; R; B \vdash \{\exists vl.\, el \Downarrow vl \ * R(vl)\}\mathsf{return}\ el\{\mathbf{false}\}$$

$$\frac{P = (e_{\mathrm{fun}} \overset{\mathsf{fun}}{\rightarrow} \{P_f\}\{Q_f\} \ * \exists vl.\, el_{\mathrm{args}} \Downarrow vl \ * P_f(vl))}{\Gamma; R; B \vdash \{P\}\mathsf{call}\, xl\, e_{\mathrm{fun}}\, el_{\mathrm{args}}\{\exists vl'.\, xl \Downarrow vl' \ * Q_f(vl')\}}$$

$$\frac{\Gamma; R; B \vdash \{P\}s\{Q\} \qquad \mathrm{modified\,vars}(s) \ \cap \ \mathrm{free\,vars}(A) \ = \ \emptyset}{\Gamma; (\lambda vl.A * R(vl)); (\lambda n.A * B(n)) \vdash \{A * P\}s\{A * Q\}}$$

Figure 5: Axiomatic Semantics of Separation Logic

Q must be otherwise *closed*, *i.e.* have no free (program) variables—because variable names would have different (local) interpretations for the function caller and callee.

A function can be described by its precondition and postcondition (each parameterized as described), so we can write the function specification $\{P\}\{Q\}$ to characterize a function. But sometimes we want to express, in logic, that the result of a function somehow depends on the value of its arguments. For this we want a (logical, not program) variable common to both P and Q; we write $\forall z : \tau.\{P\}\{Q\}$, where τ is any Coq type (that is, "τ : `Set`"). The assertion for functions is $e \stackrel{\mathsf{fun}}{\rightarrow} \Theta$ meaning that the pure expression e evaluates to a function-pointer that is callable with pre and postconditions given by Θ.

$$\Theta : \mathsf{fun\_spec} \ ::= \ \{P\}\{Q\} \quad | \quad \forall z : \tau.\Theta$$

Frame Rules. The most important feature of Separation Logic is the frame rule, usually written

$$\frac{\{P\}\, s\, \{Q\}}{\{A * P\}\, s\, \{A * Q\}}$$

The appropriate generalization of this rule to our language with control flow is the last rule of Figure 5. We can derive from it a *special frame rule* for simple statements s that do not exit or return:

$$\frac{\forall R, B.(\Gamma; R; B \vdash \{P\}\, s\, \{Q\}) \qquad \text{modified vars}(s) \ \cap \ \text{free vars}(A) \ = \ \emptyset}{\Gamma; R; B \vdash \{A * P\}\, s\, \{A * Q\}}$$

If s does not exit or return (even if it has internal call-return and block-exit), then its Hoare triple is independent of B and R as indicated by the quantification $\forall R, B$ in the premise. This rule is easily derivable from the general frame rule shown in Figure 5.

Free Variables. We use a semantic notion of free variables: x is not free in assertion A if, in any two states where only the binding of x differs, A gives the same result. However, we found it necessary to use a syntactic (inductive) definition of the variables modified by a command. One would think that command c "modifies" x if there is some state such that by the time c terminates or exits, x has a different value. However, this definition means that the modified variables of `if false then` B `else` C are *not* a superset of the modified variables of C; this lack of an inversion principle led to difficulty in proofs.

Auxiliary Variables. It is typical in Hoare Logic to use auxiliary variables to relate the pre- and postconditions, e.g., the variable a in $\{x = a\}\, x := x + 1\, \{x = a + 1\}$. In our shallow embedding of Hoare Logic in Coq, the variable a is a Coq variable, not a Cminor variable; formally, the user would prove in Coq the proposition, $\forall a, (\Gamma; R; B \vdash \{P\}s\{Q\})$ where a may appear free in any of Γ, R, B, P, s, Q. The existential assertion $\exists z.Q$ is useful in conjunction with this technique.

Assertions about functions require special handling of these quantified auxiliary variables. The assertion that some value f is a function with precondition P and postcondition Q is

written $f : \underline{\forall} x_1 \underline{\forall} x_2 \ldots \underline{\forall} x_n$, $\{P\}\{Q\}$ where P and Q are functions from value-list to assertion, each $\underline{\forall}$ is an operator of our separation logic that binds a Coq variable x_i using higher-order abstract syntax.

In the next section we show how the Separation Logic (rules of Figure 5) can be used to prove partial-correctness properties of programs; then in the following section we show how to prove soundness of the Separation Logic.

5 An Example.

This section details through the classical in-place list-reversal example [18, 19, 20] how Cminor programs can be proved using Separation Logic.

The following recursive formula defines a singly linked list in Separation Logic. The singly list is represented by a pointer i to its first cell. [8] contents_list relates a singly linked list on the heap to a Coq list. For instance, a Coq list $hd :: tl$ corresponds to a singly linked list on the heap i iff

- i points to hd,
- there exists a pointer j such that $i+1$ points to j, [9]
- the list starting at j corresponds to the Coq list tl,
- in the heap, the cell at address i is separated from the cell at address $i+1$ and both cells are separated from the rest of the list.

Pointer values are stored and loaded as 32-bit integer values and the corresponding chunk is *Mint32*.

$$\mathsf{contents\_list}([], i) =_{\text{def}} \mathbf{emp} \wedge i = 0$$
$$\mathsf{contents\_list}(hd :: tl, i) =_{\text{def}} \exists j. i \overset{Mint32}{\mapsto} hd * i+1 \overset{Mint32}{\mapsto} j * \mathsf{contents\_list}(j, tl)$$

Given a singly linked list v, the following Cminor program $\mathsf{reverse\_list}(v, w, t)$ reverses all the tail-pointers in place, leaving a pointer to the reversed list in variable w. In this program, w denotes the pointer to the previous list cell and t denotes the pointer to the next list cell.

```
w := 0                          (* the previous list cell is initialized to NULL *)
while   (v ≠ 0) do              (* non empty list *)
        t := Eload Mint32 (v+1) (* new value of the next list cell *)
        [v+1]_Mint32 := w       (* new value of the pointer to the next list cell *)
        w := v                  (* previous := current *)
        v := t                  (* current := next*)
```

[8] 0 represents a `NULL` pointer and thus an empty list, see section 2.
[9] The notation $i+1$ of Separation Logic denotes the pointer to the next field of the singly linked list.

Lemma 8 (Correctness of the list-reversal program). If v, w and t are distinct variables, then $\Gamma; R; B \vdash \{\mathsf{contents\_list}(l, v)\}\ \mathsf{reverse\_list}(v, w, t)\ \{\mathsf{contents\_list}\,(\mathsf{rev}(l), w)\}$.

This lemma states that if v points to a Coq list l, then after execution of $\mathsf{reverse\_list}(v, w, t)$, w points to the reversed Coq list $\mathsf{rev}(l)$ [10]. To prove this lemma, we apply the axiomatic semantics rules for each statement of reverse_list program. The invariant of the loop rule is defined as follows. It states that the current list l may be separated in two lists l_1 and l_2 such that firstly, the concatenation of l_1 and l_2 is l, and secondly, v corresponds to l_2 and the reverse of l_1 corresponds to w.

$\mathsf{invariant}\,(l, v, w) =_{def} \exists l_1.\exists l_2.\mathsf{contents\_list}\,(l_2, v) * \mathsf{contents\_list}\,(\mathsf{rev}(l_1), w) * \lceil l = l_1 + + l_2 \rceil$

Tactics for Separation Logic. We have a set of tactics, programmed in the tactic-definition language of Coq, to serve as a proof assistant for Cminor Separation Logic proofs. We have used the tactics to prove small examples such as the list-reversal program [17].

6 Soundness of Separation Logic

Soundness means not only that there is a model for the logic, but that the model is *the* operational semantics for which the compiler guarantees correctness! In principle we could prove soundness by syntactic induction over the Hoare Logic rules, but instead we will give a semantic definition of the Hoare sextuple $\Gamma; R; B \vdash \{P\}\, s\, \{Q\}$, and then prove each of the Hoare rules as a derived lemma from this definition.

A simple example of semantic specification is that the Hoare Logic $P \Rightarrow Q$ is defined, using the underlying logical implication, as $\forall \Psi \sigma.\ P\,\Psi\,\sigma \ \Rightarrow\ Q\,\Psi\,\sigma$. From this one could prove soundness of the Hoare Logic rule,

$$\frac{P \Rightarrow Q \qquad Q \Rightarrow R}{P \Rightarrow R}$$

(where the $\Rightarrow$ is a symbol of Hoare Logic) by expanding the definitions into the lemma,

$$\frac{\forall \sigma.(P\sigma \Rightarrow Q\sigma) \qquad \forall \sigma.(Q\sigma \Rightarrow R\sigma)}{\forall \sigma.(P\sigma \Rightarrow R\sigma)}$$

which is clearly provable in higher-order logic.

Definition 6 (Equivalence between states). (a) Two states σ and σ' are equivalent (written as $\sigma \cong \sigma'$) if they have the same stack pointer, extensionally equivalent environments, identical footprints, and if the footprint-visible portions of their memories are the same. (b) An assertion is a predicate on states that is extensional over equivalent environments (in Coq it is a dependent product of a predicate and a proof of extensionality).

[10] rev is the Coq function that reverses a list.

Definition 7 (Safe assertion). For any control κ, we define the assertion **safe** κ to mean that the combination of κ with the current state is safe:

$$\textbf{safe}\ \kappa\ =_{\text{def}}\ \lambda\Psi\sigma.\,\forall\sigma'.\ (\sigma \cong \sigma'\ \Rightarrow\ \Psi \vdash \mathsf{safe}\,(\sigma', \kappa))$$

Definition 8 (Guard). Let A be a *frame*, that is, a closed assertion (*i.e.* one with no free Cminor variables). An assertion P guards a control κ in the frame A (written as $P \,\square_A\, \kappa$) means that whenever $A * P$ holds, it is safe to execute κ. That is,

$$P \,\square_A\, \kappa\ =_{\text{def}}\ A * P\ \Rightarrow\ \textbf{safe}\ \kappa.$$

We extend this notion to say that a return-assertion R (a function from value-list to assertion) guards a return, and a block-exit assertion B (a function from block-nesting level to assertions) guards an exit:

$$R \,\boxed{\mathbb{R}}_A\, \kappa\ =_{\text{def}}\ \forall vl.R(vl) \,\square_A\, \mathsf{return}\ vl \cdot \kappa \qquad\qquad B \,\boxed{\mathbb{B}}_A\, \kappa\ =_{\text{def}}\ \forall n.B(n) \,\square_A\, \mathsf{exit}\ n \cdot \kappa$$

We extend this notion to say that a return-assertion R (*i.e.* a function from value-list to assertion) guards a return, and a block-exit assertion B (*i.e.* a function from block-nesting level to assertions) guards an exit:

$$R \,\boxed{\mathbb{R}}_A\, \kappa\ =_{\text{def}}\ \forall vl, R(vl) \,\square_A\, \mathsf{return}\ vl \cdot \kappa \qquad\qquad B \,\boxed{\mathbb{B}}_A\, \kappa\ =_{\text{def}}\ \forall n, B(n) \,\square_A\, \mathsf{exit}\ n \cdot \kappa$$

Lemma 9. If $P \,\square_A\, s_1 \cdot s_2 \cdot \kappa$ then $P \,\square_A\, (s_1; s_2) \cdot \kappa$.

Lemma 10.

1. If $R \,\boxed{\mathbb{R}}_A\, \kappa$ then $\forall s, R \,\boxed{\mathbb{R}}_A\, s \cdot \kappa$.
2. If $B \,\boxed{\mathbb{B}}_A\, \kappa$ then $\forall s, B \,\boxed{\mathbb{B}}_A\, s \cdot \kappa$.

Definition 9 (Hoare sextuple). The Hoare sextuples are defined in "continuation style", in terms of implications between continuations, as follows:

$$\begin{array}{l}\Gamma; R; B \vdash \{P\}\, s\, \{Q\}\ =_{\text{def}}\ \forall A, \kappa. \\ \quad R \,\boxed{\mathbb{R}}_{\mathsf{frame}(\Gamma,A,s)}\, \kappa\ \wedge\ B \,\boxed{\mathbb{B}}_{\mathsf{frame}(\Gamma,A,s)}\, \kappa\ \wedge\ Q \,\square_{\mathsf{frame}(\Gamma,A,s)}\, \kappa\ \Rightarrow\ P \,\square_{\mathsf{frame}(\Gamma,A,s)}\, s \cdot \kappa\end{array}$$

From this definition we prove the rules of Fig. 5 as derived lemmas.

Remark. It should be clear from the definition—after one gets over the backward nature of the continuation transform—that the Hoare judgment specifies partial correctness, not total correctness. For example, if the statement s infinitely loops, then the continuation $(\sigma, s \cdot \kappa)$ is automatically safe, and therefore $P \,\square_A\, s \cdot \kappa$ always holds. Therefore the Hoare tuple $\Gamma; R; B \vdash \{P\}s\{Q\}$ will hold for that s, regardless of Γ, R, B, P, Q.

Sequence. The soundness of the sequence statement is the proof that if the hypotheses $\mathsf{H}_1 : \Gamma; R; B \vdash \{P\}\, s_1\, \{P'\}$ and $\mathsf{H}_2 : \Gamma; R; B \vdash \{P'\}\, s_2\, \{Q\}$ hold, then we have to prove $\mathsf{Goal} : \Gamma; R; B \vdash \{P\}\, s_1; s_2\, \{Q\}$ (see Fig. 5). If we unfold the definition of the Hoare sextuples, H_1, H_2 and Goal become:

$$(\forall A, \kappa_i)\ \frac{R\ \boxed{\mathbb{R}}_{\mathsf{frame}(\Gamma,A,s_i)}\ \kappa_i \qquad B\ \boxed{\mathbb{B}}_{\mathsf{frame}(\Gamma,A,s_i)}\ \kappa_i \quad P'\ \square_{\mathsf{frame}(\Gamma,A,s_i)}\ \kappa_i}{P\ \square_{\mathsf{frame}(\Gamma,A,s_i)}\ s_i \cdot \kappa_i}\ \mathsf{H}_i, i = 1,2$$

$$(\forall A, \kappa)\ \frac{R\ \boxed{\mathbb{R}}_{\mathsf{frame}(\Gamma,A,(s_1;s_2))}\ \kappa \qquad B\ \boxed{\mathbb{B}}_{\mathsf{frame}(\Gamma,A,(s_1;s_2))}\ \kappa \qquad Q\ \square_{\mathsf{frame}(\Gamma,A,(s_1;s_2))}\ \kappa}{P\ \square_{\mathsf{frame}(\Gamma,A,(s_1;s_2))}\ (s_1; s_2) \cdot \kappa}\ \mathsf{Goal}$$

We prove $P\ \square_{\mathsf{frame}(\Gamma,A,(s_1;s_2))}\ (s_1; s_2) \cdot k$ using Lemma 6:[11]

$$\frac{\dfrac{\dfrac{R\ \boxed{\mathbb{R}}\ k}{R\ \boxed{\mathbb{R}}\ s_2 \cdot k}\ \mathsf{Lm.\ 6} \qquad \dfrac{B\ \boxed{\mathbb{B}}\ k}{B\ \boxed{\mathbb{B}}\ s_2 \cdot k}\ \mathsf{Lm.\ 6} \qquad \dfrac{R\ \boxed{\mathbb{R}}\ k \quad B\ \boxed{\mathbb{B}}\ k \quad Q\ \square\ k}{P'\ \square\ s_2 \cdot k}\ \mathsf{H}_2}{P\ \square\ s_1 \cdot s_2 \cdot k}\ \mathsf{H}_1}{P\ \square\ (s_1; s_2) \cdot k}\ \mathsf{Lm.\ 6}$$

Loop rule. Another proof example is the loop rule. The loop rule turns out to be one of the most difficult ones to prove. A loop continues executing until the loop-body performs an exit or return. If $\mathsf{loop}\, s$ executes n steps, then there will be 0 or more complete iterations of $n_1, n_2, \ldots$ steps, followed by j steps into the last iteration. Then either there is an exit (or return) from the loop, or the loop will keep going. But if the exit is from an inner-nested block, then it does not terminate the loop (or even this iteration). Thus we need a formal notion of when a statement exits.

Consider the statement $s = \mathsf{if}\ b\ \mathsf{then\ exit\ 2\ else}\ (\mathsf{skip}; x := y)$, executing in state σ. Let us execute n steps into s, that is, $\Psi \vdash (\sigma, s \cdot \kappa) \longmapsto^n (\sigma', \kappa')$. If n is small, then the behavior should not depend on κ; only when we "emerge" from s is κ important. In this example, if $\rho_\sigma b$ is a true value, then as long as $n \leq 1$ the statement s can *absorb* n steps independent of κ; if $\rho_\sigma b$ is a false value, then s can absorb up to 3 steps. To reason about absorption, we define the concatenation $\kappa_1 \circ \kappa_2$ of a control prefix κ_1 and a control κ_2 as follows:

$$\begin{array}{lll} \mathsf{Kstop} \circ \kappa & =_{\mathrm{def}} & \kappa \\ (s \cdot \kappa') \circ \kappa & =_{\mathrm{def}} & s \cdot (\kappa' \circ \kappa) \end{array} \qquad \begin{array}{lll} (\mathsf{Kblock}\, \kappa') \circ \kappa & =_{\mathrm{def}} & \mathsf{Kblock}\, (\kappa' \circ \kappa) \\ (\mathsf{Kcall}\, xl\ f\ sp\ \rho\ \kappa') \circ \kappa & =_{\mathrm{def}} & \mathsf{Kcall}\, xl\ f\ sp\ \rho\, (\kappa' \circ \kappa) \end{array}$$

Kstop is the empty prefix; $\mathsf{Kstop} \circ \kappa$ does not mean "stop," it means κ.

Definition 10 (Statement absorption). A statement s in state σ $\mathsf{absorbs}$ n steps (written as $\mathsf{absorb}(n, s, \sigma)$) iff $\forall j \leq n.\, \exists \kappa_{\mathrm{prefix}}. \exists \sigma'.\ \forall \kappa.\ \Psi \vdash (\sigma, s \cdot \kappa) \longmapsto^j (\sigma', \kappa_{\mathrm{prefix}} \circ \kappa)$.

[11] We will elide the frames from proof sketches by writing $\square$ without a subscript; this particular proof relies on a lemma that $\mathsf{closemod}(s_1, \mathsf{closemod}((s_1; s_2), A)) = \mathsf{closemod}((s_1; s_2), A)$.

Example. An exit statement by itself absorbs no steps (it immediately uses its control-tail), but block (exit 0) can absorb the 2 following steps:
$\Psi \ \vdash \ (\sigma, \mathsf{block}\,(\mathsf{exit}\,0) \cdot \kappa) \longmapsto (\sigma, \mathsf{exit}\,0 \cdot \mathsf{Kblock}\,\kappa) \longmapsto (\sigma, \kappa)$

Lemma 11.

1. $\mathsf{absorb}(0, s, \sigma)$.
2. $\mathsf{absorb}(n+1, s, \sigma) \ \Rightarrow \ \mathsf{absorb}(n, s, \sigma)$.
3. If $\neg\mathsf{absorb}(n, s, \sigma)$, then $\exists i < n.\mathsf{absorb}(i, s, \sigma) \wedge \neg\mathsf{absorb}(i+1, s, \sigma)$. We say s absorbs at most i steps in state σ.

Definition 11. We write $(s\cdot)^n s'$ to mean $\underbrace{s \cdot s \cdot \ldots \cdot s \cdot}_{n} s'$.

Lemma 12. $\dfrac{\Gamma; R; B \vdash \{I\}s\{I\}}{\Gamma; R; B \vdash \{I\}(s\cdot)^n\mathsf{loop}\,\mathsf{skip}\{\mathbf{false}\}}$

Proof. For $n = 0$, the infinite-loop (loop skip) satisfies any precondition for partial correctness. For $n+1$, assume κ, $R \Rightarrow\!\!\!\!\!\!\mathbb{R}\ \kappa$, $B \Rightarrow\!\!\!\!\!\!\mathbb{B}\ \kappa$; by the induction hypothesis (with $R \Rightarrow\!\!\!\!\!\!\mathbb{R}\ \kappa$ and $B \Rightarrow\!\!\!\!\!\!\mathbb{B}\ \kappa$) we know $I \,\square\, (s\cdot)^n\mathsf{loop}\,\mathsf{skip}\cdot\kappa$. We have $R \Rightarrow\!\!\!\!\!\!\mathbb{R}\ (s\cdot)^n\mathsf{loop}\,\mathsf{skip}\cdot\kappa$ and $B \Rightarrow\!\!\!\!\!\!\mathbb{B}\ (s\cdot)^n\mathsf{loop}\,\mathsf{skip}\cdot\kappa$ by Lemma 10. We use the hypothesis $\Gamma; R; B \vdash \{I\}s\{I\}$ to augment the result to $I \,\square\, (s; (s;)^n\mathsf{loop}\,\mathsf{skip}) \cdot \kappa$.

Theorem 13. $\dfrac{\Gamma; R; B \vdash \{I\}s\{I\}}{\Gamma; R; B \vdash \{I\}\mathsf{loop}\,s\{\mathbf{false}\}}$

Proof. Assume κ, $R \Rightarrow\!\!\!\!\!\!\mathbb{R}\ \kappa$, $B \Rightarrow\!\!\!\!\!\!\mathbb{B}\ \kappa$. To prove $I \,\square\, \mathsf{loop}\,s \cdot \kappa$, assume σ and $I\sigma$ and prove $\mathsf{safe}\,(\sigma, \mathsf{loop}\,s \cdot \kappa)$. We must prove that for any n, after n steps we are not stuck. We unfold the loop n times, that is, we use Lemma 10 to show $\mathsf{safe}\,(\sigma, (s\cdot)^n\mathsf{loop}\,\mathsf{skip} \cdot \kappa)$. We can show that if this is safe for n steps, so is $\mathsf{loop}\,s \cdot \kappa$ by the principle of absorption. Either s absorbs n steps, in which case we are done; or s absorbs at most $j < n$ steps, leading to a state σ' and a control (respectively) $\kappa_{\text{prefix}} \circ (s\cdot)^{n-1}\mathsf{loop}\,\mathsf{skip} \cdot \kappa$ or $\kappa_{\text{prefix}} \circ \mathsf{loop}\,s \cdot \kappa$. Now, because s cannot absorb $j+1$ steps, we know that either $\kappa_{\text{prefix}} = \mathsf{Kstop}$ (because s has terminated normally) or κ_{prefix} starts with a return or exit, in which case we escape (past the loop skip or the loop s, respectively) into κ. If $\kappa_{\text{prefix}} = \mathsf{Kstop}$ then we apply induction on the case $n - j$; if we escape, then (σ', κ) is safe iff $(\sigma, \mathsf{loop}\,s \cdot \kappa)$ is safe. (For example, if $j = 0$, then it must be that $s = \mathsf{return}$ or $s = \mathsf{exit}$, so in one step we reach $\kappa_{\text{prefix}} \circ \mathsf{loop}\,s \cdot \kappa$ with $\kappa_{\text{prefix}} = \mathsf{return}$ or $\kappa_{\text{prefix}} = \mathsf{exit}$.)

7 Semantic Small-step/Big-step Equivalence

Leroy's big-step semantics of Cminor is of the form $\Psi \vdash (\dot{\sigma}, s) \Downarrow_{out} \sigma'$, where the outcome *out* expresses how the statement s has terminated [3]: either normally ($\mathsf{Out}_{\mathsf{normal}}$) by falling through the next statement or prematurely through either an exit ($\mathsf{Out}_{\mathsf{exit}}\, n$) or a return statement ($\mathsf{Out}_{\mathsf{return}}\, vl$).

$$out : \mathsf{outcome} \ ::= \mathsf{Out}_{\mathsf{normal}} \mid \mathsf{Out}_{\mathsf{exit}}\, n \mid \mathsf{Out}_{\mathsf{return}}\, vl$$

Thanks to the erasure lemma 7, we can reason on the big-step semantics extended with footprints (*i.e.* $\Psi \vdash (\sigma, s) \Downarrow_{out} \sigma'$). In previous sections, we showed that Cminor programs can be proved correct in Separation Logic; that Separation Logic is sound with respect to a small-step semantics. Leroy has shown that the CompCert compiler is correct with respect to big-step semantics. Therefore, a proof of semantics equivalence between small-step and big-step semantics allows end-to-end correctness proofs from source programs to machine code.[12]

The semantic equivalence (for programs that terminate) between the small-step and the big-step semantics was helpful to us in debugging our attempts (*e.g.* pure small-step semantics with contexts, reduction semantics, transition semantics) to specify the small-step semantics. Our criteria for choosing a small-step semantics was the ability to reason inductively about nonlocal control constructs (return and exit statements) mixed with structured programming (*e.g.* loops).

The semantic equivalence is defined by two theorems using two functions $\mathsf{stmt\_of\_outcome}$ and $\mathsf{outcome\_of\_statement}$. The two functions express that the statements skip and $\mathsf{exit}\, n$ correspond respectively to the outcomes $\mathsf{Out}_{\mathsf{normal}}$ and $\mathsf{Out}_{\mathsf{exit}}\, n$. We also have:

$$\begin{array}{l}\mathsf{stmt\_of\_outcome}\,(\mathsf{Out}_{\mathsf{return}}\, vl) = \mathsf{return}\,(\mathsf{Eval} vl) \\ \mathsf{outcome\_of\_stmt}\,(\mathsf{return}\, le) = (\mathsf{Out}_{\mathsf{return}}\, vl) \qquad \forall vl\ \textit{such that}\ \Psi; \sigma \vdash e \Downarrow vl.\end{array}$$

Definition 12. Consider a statement s and a control κ. The big-step semantics accounts for some part of the effect $s \cdot \kappa$ in the *outcome* of s, but what remains (not accounted for by the outcome) is a control that we write as $s \# \kappa$. Formally it is defined by these rules:

$$\begin{array}{l}\mathsf{skip} \# \kappa \ =_{\mathrm{def}}\ \kappa \\ \mathsf{exit}\ 0 \# s_1 \cdot \ldots s_j \cdot (\mathsf{Kblock}\, \kappa) \ =_{\mathrm{def}}\ \kappa \\ \mathsf{exit}\,(n+1) \# s_1 \cdot \ldots s_j \cdot (\mathsf{Kblock}\, \kappa) \ =_{\mathrm{def}}\ \mathsf{exit}\ n \cdot \kappa \\ \mathsf{return}\, l \# s_1 \cdot \ldots s_j \cdot (\mathsf{Kcall}\ xl\ f\ sp\, \rho\, \kappa) \ =_{\mathrm{def}}\ \kappa \\ s \# \kappa \ =_{\mathrm{def}}\ s \cdot \kappa \quad \textit{when}\ s \neq \mathsf{skip}, \mathsf{exit}, \mathsf{return}\end{array}$$

Lemma 15 (Big-step implies small-step, induction lemma). If $\Psi \vdash (\sigma, s) \Downarrow_{out} \sigma'$, then $\forall \kappa, \Psi \vdash \ (\sigma, s \cdot \kappa) \longmapsto^* (\sigma', \mathsf{stmt\_of\_outcome}\,(out) \# \kappa)$.

[12] In future work, we hope to extend this to nonterminating or concurrent programs by directly proving a stronger (small-step) theorem about the CompCert compiler.

Proof. By induction on the derivation of $\Downarrow$ and case analysis on s.

$$\Psi \vdash (\sigma, \mathsf{Kstop}); out \rightsquigarrow \sigma; out \qquad \frac{\Psi \vdash (\sigma, s) \Downarrow_{out_1} \sigma_1 \quad \Psi \vdash (\sigma_1, \kappa); out_1 \rightsquigarrow \sigma'; out'}{\Psi \vdash (\sigma, (s \cdot \kappa)); \mathsf{Out}_{\mathsf{normal}} \rightsquigarrow \sigma'; out'}$$

$$\frac{\begin{array}{c}\kappa = s_1 \cdot s_2 \cdot \ldots s_j \cdot \mathsf{Kblock}\, \kappa_1 \\ out_1 = \mathit{if}\ (n = 0) \mathit{then}\, \mathsf{Out}_{\mathsf{normal}} \mathit{else}\, \mathsf{Out}_{\mathsf{exit}}\, (n-1) \\ \Psi \vdash (\sigma, \kappa_1); out_1 \rightsquigarrow \sigma'; out'\end{array}}{\Psi \vdash (\sigma, \kappa); \mathsf{Out}_{\mathsf{exit}}\, (n) \rightsquigarrow \sigma'; out'}$$

$$\frac{\begin{array}{c}\kappa = \mathit{any\ sequence\ of} \cdot \mathit{\ and\ } \mathsf{Kblock}\ \mathit{operators\ terminating\ in\ } \mathsf{Kcall}\, xl\, f\, sp_1\, \rho_1\, \kappa_1 \\ \rho_1 = \rho_1[x := vl] \qquad \phi_1 = \phi_\sigma \backslash [sp_1, sp_1 + \mathsf{stackspace}(f)) \\ \sigma_1 = (sp_1, \rho_1, \phi_\sigma, \mathsf{free}(m_\sigma, sp_\sigma)) \qquad \Psi \vdash (\sigma_1, \kappa); \mathsf{Out}_{\mathsf{normal}} \rightsquigarrow \sigma'; out\end{array}}{\Psi \vdash (\sigma, \kappa); \mathsf{Out}_{\mathsf{return}}\, (vl) \rightsquigarrow \sigma'; out}$$

$$\frac{\Psi \vdash (\sigma, s) \Downarrow_{out_1} \sigma_1 \qquad \Psi \vdash (\sigma_1, \kappa); out_1 \rightsquigarrow \sigma_2; out_2}{\Psi \vdash (\sigma, s \cdot \kappa) \Downarrow_{out_2} \sigma_2}$$

Figure 6: Big-step semantics with control stack

The reverse induction lemma relies on a big-step semantics with control stacks (belonging to continuations) that is defined by the following rule. The control stack κ represents the statements that need to be executed after s in order to get the outcome out_2. The judgment $\Psi \vdash (\sigma_1, \kappa); out_1 \rightsquigarrow \sigma_2; out_2$ allows the execution of the statements that belong to the control stack κ and can be read as: the outcome out_1 transmitted to the control stack κ yields an outcome out_2.

$$\frac{\Psi \vdash (\sigma, s) \Downarrow_{out_1} \sigma_1 \qquad \Psi \vdash (\sigma_1, \kappa); out_1 \rightsquigarrow \sigma_2; out_2}{\Psi \vdash (\sigma, s \cdot \kappa) \Downarrow_{out_2} \sigma_2}$$

Figure 6 defines judgments of the form $\Psi \vdash (\sigma, \kappa); out \rightsquigarrow \sigma'; out'$ that are called from the big-step semantics with control stacks.

Lemma 16 (Small-step implies big-step, main induction lemma). If $\Psi \vdash (\sigma, s \cdot \kappa) \longmapsto^* (\sigma', s' \cdot \kappa)$ then $\Psi \vdash (\sigma, s) \Downarrow_{out} \sigma' \wedge out = \mathsf{outcome\_of\_stmt}(s')$.

Proof. This proof relies on the following lemma and on an induction on the length of the reduction sequence $(\sigma, s \cdot \kappa) \longmapsto^* (\sigma', s' \cdot \kappa)$.

Lemma 17. If $\Psi \vdash (\sigma, s \cdot \kappa) \longmapsto (\sigma_1, s_1 \cdot \kappa_1)$ and $\Psi \vdash (\sigma_1, s_1 \cdot \kappa_1) \Downarrow_{out} \sigma'$ then $\Psi \vdash (\sigma, s \cdot \kappa) \Downarrow_{out} \sigma'$.

Proof. By induction on the derivation of $\longmapsto$, using lemmas such as the following one.

Lemma 18. If $\Psi \vdash (\sigma, \mathsf{exit}\, n \cdot \kappa) \Downarrow_{out} \sigma'$ then $\Psi \vdash (\sigma, \mathsf{exit}\, (n+1) \cdot (\mathsf{Kblock}\, \kappa)) \Downarrow_{out} \sigma'$.

Proof. By case analysis.

Theorem 19 (Semantic preservation). For all safe program P, the big-step execution of P is equivalent to the small-step execution of P.

Theorem 20 (Compiler correctness [Leroy]). For all program P, if the CompCert compiler transforms P into machine code C without reporting errors, and P has well-defined semantics, the C has the same semantics as P.

Proof. Proved in Coq. In addition, the compiler is always observed empirically to produce a result—1000-lines programs have been compiled—so the partial-correctness proof is not vacuous.

Corollary 21. If a program is correct and terminates in our small-step continuation semantics, then CompCert correctly compiles it to equivalent (therefore safe and correct) machine code.

8 The Machine-checked Proof

We have proved in Coq the soundness of Separation Logic for Cminor. Each rule is proved as a lemma; in addition there is a main theorem that if you prove all your function bodies satisfy their pre/postconditions, then the program "call main()" is safe. We have informally tested the adequacy of our result by doing tactical proofs of small programs [17].

Lines	Component
41	Axioms: dependent unique choice, relational choice, extensionality
8792	Memory model, floats, 32-bit integers, values, operators, maps (exactly as in CompCert [3])
4408	Sharable permissions, Cminor language, operational semantics
462	Separation Logic operators and rules
9874	Soundness proof of Separation Logic

These line counts include some repetition of specifications (between Modules and Module Types) in Coq's module system.

Figure 7 shows the architecture of our Coq development and the connection between our Separation Logic with the CompCert certified compiler. The next section describes how we plan to connect a Cminor certified compiler directly to the small-step semantics, instead of going through the big-step semantics.

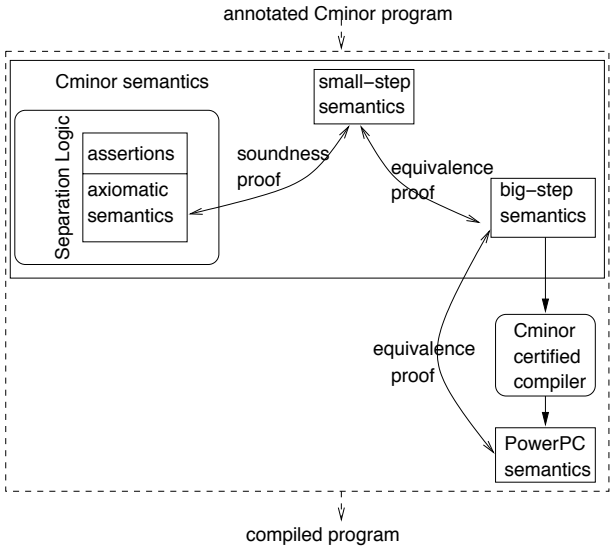

Figure 7: Architecture of the Coq development

9 Sequential Reasoning about Sequential Features

Concurrent Cminor, like most concurrent programming languages used in practice, is a sequential programming language with a few concurrent features (locks and threads) added on. We would like to be able to reason about the sequential features using purely sequential reasoning, then add the concurrent reasoning as an afterthought. If we have to reason about all the many sequential features without being able to assume such things as determinacy and sequential control, then the proofs become much more difficult.

One would expect this approach to run into trouble because critical assumptions underlying the sequential operational semantics would not hold in the concurrent setting. For example, on a shared-memory multiprocessor we cannot assume that (x:=x+1; x:=x+1) has the same effect as (x:=x+2); and on any real multiprocessor we cannot even assume *sequential consistency*—that the semantics of n threads is some interleaving of the steps of the individual threads.

Brookes [21] proposes a solution with his "footprint semantics," in which all the steps from one synchronization to another can be (semantically) collapsed into one step. One key idea in that work, which we adopt in a different way, is that the operational semantics "gets stuck" if there's interference, so we need not reason about racy programs. Brookes writes, "One can use more elementary reasoning techniques to deal with synchronization-free code, such as the familiar Hoare-style inference rules for sequential programs." But this is not enough! Consider a multi-exit loop whose body contains a lock/unlock synchronization.

This is not synchronization-free code, and yet we still want to use sequential reasoning to handle the complicated (sequential) multi-exit feature.

We will solve this problem in several stages.

1. We give a sequentially consistent small-step operational semantics for Concurrent Cminor that assumes noninterference (and gets "stuck" on interference) [22].

2. From this semantics, we calculate a single-thread small-step semantics equipped with an oracle ω that predicts the effects of synchronizations (it is *not* just for running synchronization-free code). The oracular step $(\omega, \sigma, \kappa) \longmapsto (\omega', \sigma', \kappa')$ is almost the same as the $(\sigma, \kappa) \longmapsto (\sigma', \kappa')$ that we define in the present report. In fact, for *all* of the sequential operators defined in this report,

$$\frac{(\sigma, \kappa) \longmapsto (\sigma', \kappa')}{(\omega, \sigma, \kappa) \longmapsto (\omega, \sigma', \kappa')}$$

   that is, the oracle is not consulted. For the concurrent operators [22] it is still the case that $(\omega, \sigma, \kappa) \longmapsto (\omega', \sigma', \kappa')$ is deterministic (because ω records the order in which threads interleave—see Appendix).

3. We define a Sequential Separation Logic and prove it sound w.r.t. the (oracular) sequential small-step.

4. We define a Concurrent Separation Logic for Cminor as an extension of the Sequential Separation Logic. Its soundness proof uses the sequential soundness proof as a lemma [22].

5. (Future work.) We will use Concurrent Separation Logic to guarantee noninterference of source programs. Then (x:=x+1; x:=x+1) *will* have the same effect as (x:=x+2). (Brookes can actually say something slightly stronger: (x:=x+1; x:=x+1) is semantically *equal* to (x:=x+2).)

6. To compile Concurrent Cminor, we will use a sequential Cminor compiler equipped with a proof that it compiles noninterfering source threads into equivalent noninterfering machine-language threads. Leroy's compiler of today is proved correct w.r.t. a big-step semantics, but perhaps the same compiler can be proved correct w.r.t. the deterministic (oracular) sequential small-step semantics given in this report.

7. We will demonstrate, with respect to a formal model of weak-memory-consistency microprocessor, that noninterfering machine-language programs give the same results as they would on a sequentially consistent machine.

Stages 1 and 3 of this plan are the main results of the current report. Previous machine-verified soundness proofs for Hoare Logic [8] and Separation Logic [23] cannot be used in this way, because they are with respect to big-step semantics. Brookes's footstep semantics cannot be used (in its current form) because it does not support the massively sequential reasoning that we need for such features as the nonlocal-exit loop.

10 Conclusion

In this report, we have defined a formal semantics for the language Cminor. It consists of a big-step semantics for expressions and a small-step semantics for statements. The small-step semantics is based on continuations mainly to allow a uniform representation of statement execution. Then, we have defined a Separation Logic for Cminor. It consists of an assertion language and an axiomatic semantics. We have extended classical Hoare triples to sextuples in order to take into account nonlocal control constructs (return, exit). From this definition of sextuples, we have proved the rules of axiomatic semantics, thus proving the soundness of our Separation Logic. We have also proved the semantic equivalence between our small-step semantics and the big-step semantics of the CompCert certified compiler.

Small-step reasoning is useful for sequential programming languages that will be extended with concurrent features; but small-step reasoning about nonlocal control constructs mixed with structured programming (loop) is not trivial. We have relied on the determinacy of the small-step relation so that we can define concepts such as $\mathsf{absorb}(n, s, \sigma)$.

Of course, sequential threads are not deterministic in the presence of concurrency. Therefore, in our extension to concurrency we will rely on determinizing oracles (see Appendix) which will allow the theorems in this report to be used even in a concurrent language. We strongly recommend this approach, as it cleanly separates the difficult problems in nonlocal control flow from the problems of concurrent access to shared memory.

References

[1] J Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989.

[2] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *IEEE Conference on Software Engineering and Formal Methods (SEFM'05)*, 2005.

[3] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL'06*, pages 42–54. ACM Press, 2006.

[4] The Coq proof assistant. `http://coq.inria.fr`.

[5] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *Symp. on Formal Methods (FM'06)*, volume 4805 of *Lecture Notes in Computer Science*, pages 460–475, 2006.

[6] Zaynah Dargaye. Décurryfication certifiée. In *JFLA (Journées Françaises des Langages Applicatifs)*, pages 119–133, 2007.

[7] Xavier Leroy. Coinductive big-step operational semantics. In *European Symposium on Programming (ESOP 2006)*, volume 3924 of *Lecture Notes in Computer Science*, pages 54–68. Springer-Verlag, 2006.

[8] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[9] Gerwin Klein, Harvey Tuch, and Michael Norrish. Types, bytes, and separation logic. In *POPL'07*, pages 97–108. ACM Press, January 2007.

[10] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *POPL'06*, pages 320–333. ACM Press, January 2006.

[11] *American National Standard for Information Systems – Programming Language – C*. American National Standards Institute, 1990.

[12] Sandrine Blazy and Xavier Leroy. Formal verification of a memory model for C-like imperative languages. In *Formal Engineering Methods*, volume 3785 of *Lecture Notes in Computer Science*, pages 280–299, 2005.

[13] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL'05*, pages 259–270, 2005.

[14] Matthew J. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.

[15] Samin Ishtiaq and Peter O'Hearn. BI as an assertion language for mutable data structures. In *POPL'01*, pages 14–26. ACM Press, January 2001.

[16] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL'01*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19, September 2001.

[17] Andrew W. Appel. Tactics for separation logic. `http://www.cs.princeton.edu/~appel/papers/septacs.pdf`, January 2006.

[18] T.Weber. Towards mechanized program verification with separation logic. In *CSL*, number 3210 in Lecture Notes in Computer Science, pages 250–264, 2004.

[19] Richard Bornat. Proving pointer programs in hoare logic. In *MPC*, pages 102–126, 2000.

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

[21] Stephen Brookes. A grainless semantics for parallel programs with shared mutable data. In *Mathematical Foundations of Programming Semantics*, May 2006.

[22] Andrew W. Appel, Aquinas Hobor, and Francesco Zappa Nardelli. Oracle semantics for Concurrent C minor. Submitted for publication, July 2007.

[23] Reynald Affeldt, Nicolas Marti, and Akinori Yonezawa. Towards formal verification of memory properties using separation logic. 22nd Workshop of the Japan Society for Software Science and Technology, 2005.

Appendix: Summary of Concurrent Cminor

Appel, Hobor, and Zappa Nardelli have specified [22] Concurrent Cminor and shown how to use oracles to create the illusion of a sequential, deterministic, constructively computable small-step relation. Alas, their Latex is not ready, so we summarize the result here.

We add five more statements to make *Concurrent Cminor*:

$$s : \mathsf{stmt} \quad ::= \ \ldots \mid \mathsf{fork}\, e(el) \mid \mathsf{make\_lock}\, e\, \mathsf{with}\, R \mid \mathsf{free\_lock}\, e \mid \mathsf{lock}\, e \mid \mathsf{unlock}\, e$$

The fork statement spawns a new thread; the new thread starts with the call of function e with arguments el. No variables are shared between the caller and callee except through the function parameters. There is no special statement for thread-exit; a thread exits by returning from its top-level function call.

The statement $\mathsf{make\_lock}\, e\, \mathsf{with}\, R$ takes a memory address e and declares it to be a lock with resource invariant R, where R is an assertion. The address is turned back into an ordinary location by $\mathsf{free\_lock}\, e$.

The $\mathsf{lock}\,(e)$ statement evaluates e to an address v, then attempts to acquire lock v, waiting if necessary. The $\mathsf{unlock}\,(e)$ statement releases a lock.

The concurrent operational semantics is achieved by using an oracle to determinize the thread interleaving. The soundness property for the Concurrent Separation Logic must then be shown for all oracles.

From the global oracle we can derive a per-thread oracle ω, from which we can compute the small-step relation even across synchronization operations such as lock and unlock. From the continuation $(\omega, \sigma, \mathsf{lock}\, l \cdot \kappa)$ we use the information in ω to run all the other threads that take turns executing until the current thread resumes, yielding a state σ' with the remainder ω'. Then we use information from ω' to find the footprint ϕ_l controlled by the lock l, and make a new state $\sigma'' = \sigma'[:= \phi_{\sigma'} \oplus \phi_l]$. Now the current thread can resume with $(\omega', \sigma'', \kappa)$.

In the present report we have used a style of Coq in which the assertion logic (`Prop`) is classical, but the functional notation (`Fixpoint`) is constructive—we reason classically about executable programs. But in the present report we have argued that it's convenient to use Coq's functional notation for the small-step. The oracular step $(\omega, \sigma, \kappa) \longmapsto (\omega', \sigma', \kappa')$, although it is deterministic, will not be constructively computable. Therefore, all the proofs shown in this report are still correct as explained in English, but in Coq they will need to be rephrased using deterministic (nonconstructive) relations, rather than constructive functions.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399