Separation Logic for Small-step Cminor

Andrew W. Appel[1,⋆] and Sandrine Blazy[2,⋆]

[1] Princeton University
[2] ENSIIE

Abstract. Cminor is a mid-level imperative programming language; there are 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 and we have designed a Separation Logic for Cminor. In this paper, we give a small-step semantics (instead of the big-step of the proved-correct compiler) that is motivated by the need to support future concurrent extensions. We detail 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. The work presented in this paper has been carried out in the Coq proof assistant. It is a first step towards an environment in which concurrent Cminor programs can be verified using Separation Logic and also compiled by a proved-correct compiler with formal end-to-end correctness guarantees.

1 Introduction

To appear in TPHOLs 2007 (20th International Conference on Theorem Proving in Higher-Order Logics), September 2007, *Lecture Notes in Computer Science*, Springer-Verlag.

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" [13]. Leinenbach *et al.* [11] have built and proved a compiler for *C0*, a small C-like language, as part of a project to build machine-checked correctness proofs of source programs, Hoare Logic, compiler, micro-kernel, and RISC processor. These are both simple one- or two-pass nonoptimizing compilers.

⋆ Appel supported in part by NSF Grants CCF-0540914 and CNS-0627650. This work was done, in part, while both authors were on sabbatical at INRIA.

Leroy [12] has built and proved correct in Coq [1] a compiler called *CompCert* from a high-level intermediate language *Cminor* to assembly language for the Power PC architecture. This compiler has 4 intermediate languages, allowing 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 can have components written in different languages, 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 to serve as a common denominator between high-level languages. Our new 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 Leinenbach's 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 CompCert; the storage-allocation view suits the one-pass nonoptimizing C0 compiler and can accommodate in-line assembly code.

Cminor is 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, 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 [8], 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 libraries.
  - As we show in this paper, Cminor supports an axiomatic Hoare Logic (in fact, Separation Logic), proved sound with respect to the small-step semantics, for reasoning about low-level (C-like) programs.
  - 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. The sequential soundness proofs we present here should be reusable in a concurrent setting, as we will explain.

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, Leroy's semantics is a big-step semantics that

can be used only to reason about terminating sequential programs. 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. Henceforth in this paper, Cminor will refer to the new version of the Cminor language.

The main contributions of this paper are a small-step semantics suitable for compilation and for Hoare Logic; and the first machine-checked proof of soundness of a sequential Hoare Logic (Separation Logic) w.r.t. a small-step semantics. Schirmer [17] has a machine-checked *big-step* Hoare-Logic soundness proof for a control flow much like ours, extended by Klein *et al.* [10] to a C-like memory model. Ni and Shao [14] 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.

2 Big-step Expression Semantics

The C standard [2] 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 [6] 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 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.

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.

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. Among these operation symbols are casts. Cminor casts correspond to all portable C casts. 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 Eop with the operation symbol for accessing global names).

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

Expression Evaluation. In original Cminor, expression evaluation is expressed by an inductive big-step relation. Big-step statement execution is problematic for concurrency, but big-step *expression* evaluation is fine even for concurrent programs, since we will use the separation logic to prove noninterference.

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 $f\,x\,y \Rightarrow f\,x\,z \Rightarrow y = z$. Before Coq's new functional induction tactic was available, we developed special-purpose tactics to enable these proofs. Although we specify expression evaluation as a function in Coq, we present evaluation as a judgment relation in Fig. 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 ($\mathsf{ident} \to \mathsf{option\,block}$) mapping identifiers to function-pointers and other global constants, and a global mapping ($\mathsf{block} \to \mathsf{option\,function}$) that maps certain ("text-segment") addresses to function definitions.
- sp : block. The "stack pointer" giving the address and size 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.
- ϕ : $\mathsf{footprint}$. It represents the memory used by the evaluation of an expression (or a statement). It is a mapping from memory addresses to permissions. Leroy's Cminor has no footprints.
- m : mem. The memory, a finite mapping from blocks to block contents [6]. 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.

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 $\phi_0 \vdash \mathsf{store}_{ch}\,v \Rightarrow \phi_1 \nvdash \mathsf{load}_{ch}\,v$, $\phi_0 \vdash \mathsf{load}_{ch}\,v \Rightarrow \phi \vdash \mathsf{load}_{ch}\,v$ and $\phi_0 \vdash \mathsf{store}_{ch}\,v \Rightarrow$

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

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

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

Fig. 1. Expression evaluation rules

$\phi \vdash \mathsf{store}_{ch}\, v$. One can think of ϕ as a set of fractional permissions [7], 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. Instead of fractions, we use a more general and powerful model of sharable permissions similar to one described by Parkinson [16, Ch. 5].

Most previous models of Separation Logic (*e.g.*, Ishtiaq and O'Hearn [9]) 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; this 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 Fig. 1, the judgment $\Psi; sp \vdash op(vl) \Downarrow_{\mathsf{eval\_operation}} v$ takes an operator op applied to a list of values vl and (if vl contains appropriate values) produces some value v. Operators that access global names and local stack-blocks make use of Ψ and sp respectively to return the address of a global name or a 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 $\Psi; (sp_\sigma; \rho_\sigma; \phi_\sigma; m_\sigma) \vdash e \Downarrow v$.

Notation. We write $\sigma[:= \rho']$ to mean the state σ with its environment component ρ replaced by ρ', and so on (*e.g.* see rules 2 and 3 of Fig. 2 in Section 4).

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 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. But 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.

Erased Expression Evaluation. The Cminor compiler (CompCert) is proved correct w.r.t. an operational semantics that does not use footprints. Any program that successfully evaluates with footprints will also evaluate ignoring footprints. Thus, for sequential programs where we do not need noninterference, it is sound to prove properties in a footprint semantics and compile in an erased semantics. We formalize and prove this in the full technical report [4].

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 local 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. $\mathsf{return}\, el$ 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 statements, blocks and exits suffice to express efficiently all reducible control-flow graphs, notably those arising from C loops. The C statements break and $\mathsf{continue}$ are translated as appropriate exit statements. Blazy *et al.* [5] detail the translation of these C statements into Cminor.

Function Definitions. A program Ψ comprises 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 may be written as $f = (xl, yl, n, s)$, 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), which avoids induction over search rules in both the Hoare-Logic soundness proof and the compiler correctness proof.[3]

Definition 1. *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.*

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

The sequential small-step function takes the form $\Psi \vdash k \longmapsto k'$ (see Fig. 2), and we define as usual its reflexive transitive closure $\longmapsto^*$. As in C, there is no boolean type in Cminor. In Fig. 2, the predicate $\mathsf{is\_true}\,v$ holds if v is a pointer or a nonzero integer; is_false holds only on 0. 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 $\mathsf{exit}\,n \cdot \kappa$.

Lemma 3. *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 statements containing expressions).*

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

Definition 2. *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 3. *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* $\longmapsto^*$*.*

[3] We have proved in Coq the equivalence of this small-step semantics with the big-step semantics of CompCert (for programs that terminate).

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

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

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

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

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

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

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

Fig. 2. Sequential small-step relation. We omit here call and return, which are in the full technical report [4].

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 [15]; 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 give each assertion a semantic meaning in Coq. We have $P, Q : \mathsf{assert}$ where $\mathsf{assert} = \mathsf{prog} \to \mathsf{state} \to \mathsf{Prop}$. So $P \Psi \sigma$ is a proposition of logic and we say that σ satisfies P.

Assertion Operators. In Fig. 3, 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 $\lceil A \rceil$.

O'Hearn and Reynolds specify Separation Logic for a little language in which expressions evaluate independently of the heap [15]. That is, their expressions access only the program variables and do not even have *read* side effects on the

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

Fig. 3. Main operators of Separation Logic

memory. Memory reads 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 it contains no Eload operators—so that it cannot 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 $e \Downarrow v$ to mean that expression e is pure and 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; 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. That is, $e_1 \overset{ch}{\mapsto} e_2$ 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.

The Hoare Sextuple. Cminor has commands to call functions, to exit (from a block), and to 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 : $\mathsf{list\ val} \to \mathsf{assert}$ is the *return environment*, giving the current function's post-condition as a predicate on the list of returned values; and

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

Most of the rules of sequential Separation Logic are given in Fig. 4. In this paper, we omit the rules for return and call, which are detailed in the full technical report. Let us note that the Γ context is used to update global function names, none of which is illustrated in this paper.

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

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

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

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

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

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

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

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

Fig. 4. Axiomatic Semantics of Separation Logic (without call and return)

The rule for $[e]_{ch}{:=}e_1$ requires the same store permission than the small-step rule, but in Fig. 4, the permission is hidden in the definition of $e \overset{ch}{\mapsto} e_2$. The rules for $[e]_{ch}{:=}e_1$ and $\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$ 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.

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

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

The statement $\mathsf{exit}\, i$ 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 $(\mathsf{exit}\, i)$ 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 $(\mathsf{block}\, s)$ 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 $(\mathsf{exit}\, n)$ 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 $(\mathsf{exit}\, n)$ statement.

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 Fig. 4. We can derive from it a *special frame rule* for simple statements s that do not exit or return:

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

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.

Application. In the full technical report [4], we show how the Separation Logic (*i.e.* the rules of Fig. 4) can be used to prove partial correctness properties of programs, with the classical in-place list-reversal example. Such proofs rely on a set of tactics, that we have written in the tactic definition language of Coq, to serve as a proof assistant for Cminor Separation Logic proofs [3].

5 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 on the left (where the $\Rightarrow$ is a symbol of Hoare Logic) by expanding the definitions into the lemma on the right (where the $\Rightarrow$ is in the underlying logic), which is clearly provable in higher-order logic:

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

Definition 4. *(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).*

Definition 5. *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 6. *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:

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

Lemma 4. *If $P\,\Box_A\, s_1 \cdot s_2 \cdot \kappa$ then $P\,\Box_A\, (s_1; s_2) \cdot \kappa$.*

Lemma 5. *If $R\,\boxed{\mathbb{R}}_A\,\kappa$ then $\forall s, R\,\boxed{\mathbb{R}}_A\, s \cdot \kappa$. If $B\,\boxed{\mathbb{B}}_A\,\kappa$ then $\forall s, B\,\boxed{\mathbb{B}}_A\, s \cdot \kappa$.*

Definition 7 (Frame). *A* frame *is constructed from the global environment Γ, an arbitrary frame assertion A, and a statement s, by the conjunction of Γ with the assertion A closed over any variable modified by s:*

$$\mathsf{frame}(\Gamma, A, s) \;=_{\text{def}}\; \Gamma * \mathsf{closemod}(s, A)$$

Definition 8 (Hoare sextuples). *The Hoare sextuples are defined in "continuation style," in terms of implications between continuations, as follows:*

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

From this definition we prove the rules of Fig. 4 as derived lemmas.

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 $P\,\Box_A\, s \cdot \kappa$ 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. 4). If we unfold the definition of the Hoare sextuples, H_1, H_2 and Goal become:

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

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

We prove $P\,\Box_{\mathsf{frame}(\Gamma,A,(s_1;s_2))}\,(s_1; s_2) \cdot k$ using Lemma 4:[4]

$$\dfrac{\dfrac{\dfrac{R\,\boxed{\mathbb{R}}\,k}{R\,\boxed{\mathbb{R}}\,s_2 \cdot k}\,\mathsf{Lm.\ 5} \qquad \dfrac{B\,\boxed{\mathbb{B}}\,k}{B\,\boxed{\mathbb{B}}\,s_2 \cdot k}\,\mathsf{Lm.\ 5} \qquad \dfrac{R\,\boxed{\mathbb{R}}\,k \quad B\,\boxed{\mathbb{B}}\,k \quad Q\,\Box\,k}{P'\,\Box\, s_2 \cdot k}\,\mathsf{H}_2}{P\,\Box\, s_1 \cdot s_2 \cdot k}\,\mathsf{H}_1}{P\,\Box\,(s_1; s_2) \cdot k}\;\mathsf{Lm.\ 4}$$

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

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 $s = \mathsf{if}\, b\, \mathsf{then}\, \mathsf{exit}\, 2\, \mathsf{else}\, (\mathsf{skip}; x := y)$, 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:

$$\mathsf{Kstop} \circ \kappa =_{\text{def}} \kappa \qquad (\mathsf{Kblock}\, \kappa') \circ \kappa =_{\text{def}} \mathsf{Kblock}\, (\kappa' \circ \kappa)$$
$$(s \cdot \kappa') \circ \kappa =_{\text{def}} s \cdot (\kappa' \circ \kappa) \qquad (\mathsf{Kcall}\, xl\, f\, sp\, \rho\, \kappa') \circ \kappa =_{\text{def}} \mathsf{Kcall}\, xl\, f\, sp\, \rho\, (\kappa' \circ \kappa)$$

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

Definition 9 (absorption). *A statement s in state σ* $\mathsf{absorbs}$ *n steps (written as* $\mathsf{absorb}(n, s, \sigma)$*) iff* $\forall j \leq n. \exists \kappa_{\text{prefix}}. \exists \sigma'.\ \forall \kappa.\ \Psi \vdash (\sigma, s \cdot \kappa) \longmapsto^j (\sigma', \kappa_{\text{prefix}} \circ \kappa)$.

Example 1. An exit statement by itself absorbs no steps (it immediately uses its control-tail), but $\mathsf{block}\, (\mathsf{exit}\, 0)$ 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 6. *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 that s absorbs at most i steps in state σ.*

Definition 10. *We write $(s;)^n s'$ to mean* $\underbrace{s; s; \ldots; s;}_{n}\ s'$.

Lemma 7. $$\frac{\Gamma; R; B \vdash \{I\} s \{I\}}{\Gamma; R; B \vdash \{I\} (s;)^n \mathsf{loop}\, \mathsf{skip} \{\mathbf{false}\}}$$

Proof. For $n = 0$, the infinite-loop $(\mathsf{loop}\, \mathsf{skip})$ *satisfies any precondition for partial correctness. For $n+1$, assume κ, $R \,\mathbb{R}\, \kappa$, $B \,\mathbb{B}\, \kappa$; by the induction hypothesis (with $R \,\mathbb{R}\, \kappa$ and $B \,\mathbb{B}\, \kappa$) we know $I \,\square\, (s;)^n \mathsf{loop}\, \mathsf{skip} \cdot \kappa$. We have $R \,\mathbb{R}\, (s;)^n \mathsf{loop}\, \mathsf{skip} \cdot \kappa$ and $B \,\mathbb{B}\, (s;)^n \mathsf{loop}\, \mathsf{skip} \cdot \kappa$ by Lemma 5. 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 1. $$\frac{\Gamma; R; B \vdash \{I\} s \{I\}}{\Gamma; R; B \vdash \{I\} \mathsf{loop}\, s \{\mathbf{false}\}}$$

Proof. Assume κ, $R \,\mathbb{R}\, \kappa$, $B \,\mathbb{B}\, \kappa$. 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 7 to show*

$\mathsf{safe}\,(\sigma, (s;)^n \mathsf{loop}\,\mathsf{skip} \cdot \kappa)$. *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)* $\kappa_{prefix} \circ (s;)^{n-1} \mathsf{loop}\,\mathsf{skip} \cdot \kappa$ *or* $\kappa_{prefix} \circ \mathsf{loop}\,s \cdot \kappa$. *Now, because s cannot absorb $j+1$ steps, we know that either κ_{prefix} is empty (because s has terminated normally) or κ_{prefix} starts with a* return *or* exit*, in which case we escape (resp. past the* $\mathsf{loop}\,\mathsf{skip}$ *or the* $\mathsf{loop}\,s$*) into κ. If κ_{prefix} is empty then we apply strong induction on the case $n - j$ steps; 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 $s =$* return *or $s =$* exit *, so in one step we reach* $\kappa_{prefix} \circ (\mathsf{loop}\,s \cdot \kappa)$ *with* $\kappa_{prefix} =$ return *or* $\kappa_{prefix} =$ exit *.)* ■

6 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. 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.

We will solve this problem in several stages. Stage 1 of this plan is the current paper. Stages 2, 3, and 4 are work in progress; the remainder is future work.

1. We have made the language, the Separation Logic, and our proof extensible: the set of control-flow statements is fixed (inductive) but the set of straight-line statements is extensible by means of a parameterized module in Coq. We have added to each state σ an *oracle* which predicts the meaning of the extended instruction (but which does nothing on the core language). All the proofs we have described in this paper are on this extensible language.
2. We define spawn, lock, and unlock as extended straight-line statements. We define a concurrent small-step semantics that assumes noninterference (and gets "stuck" on interference).
3. From this semantics, we calculate a single-thread small-step semantics equipped with the oracle that predicts the effects of synchronizations.
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.
5. 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).

6. We will prove that the Cminor compiler (CompCert) compiles each footprint-safe source thread into an equivalent footprint-safe machine-language thread. Thus, noninterfering source programs will produce noninterfering machine-language programs.
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.

7 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 [3].

Lines	Component
41	Axioms: dependent unique choice, relational choice, extensionality
8792	Memory model, floats, 32-bit integers, values, operators, maps (exactly as in CompCert [12])
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.

8 Conclusion

In this paper, 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. The small-step semantics deals with nonlocal control constructs (return, exit) and is designed to extend to the concurrent setting.

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. 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, so the Cminor programs that we prove in Separation Logic can be compiled by the CompCert certified compiler. We plan to connect a Cminor certified compiler directly to the small-step semantics, instead of going through the big-step semantics.

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 $\mathsf{absorb}(n, s, \sigma)$.

References

1. The Coq proof assistant. `http://coq.inria.fr`.
2. *American National Standard for Information Systems – Programming Language – C*. American National Standards Institute, 1990.
3. Andrew W. Appel. Tactics for separation logic. `http://www.cs.princeton.edu/~appel/papers/septacs.pdf`, January 2006.
4. Andrew W. Appel and Sandrine Blazy. Separation logic for small-step Cminor (extended version). Technical Report RR 6138, INRIA, March 2007. `https://hal.inria.fr/inria-00134699`.
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. 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.
7. Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL '05*, pages 259–270, 2005.
8. Zaynah Dargaye. Décurryfication certifiée. In *JFLA (Journées Françaises des Langages Applicatifs)*, pages 119–133, 2007.
9. 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.
10. Gerwin Klein, Harvey Tuch, and Michael Norrish. Types, bytes, and separation logic. In *POPL'07*, pages 97–108. ACM Press, January 2007.
11. 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.
12. 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.
13. J Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989.
14. Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *POPL'06*, pages 320–333. ACM Press, January 2006.
15. 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.
16. Matthew J. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.
17. Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.