Separation Logic for Small-step C Minor

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

[1] Princeton University and INRIA Rocquencourt
appel@princeton.edu
[2] ENSIIE and INRIA Rocquencourt
sandrine.blazy@inria.fr

October 2006

Abstract. C minor is a mid-level imperative programming language (just below C), and there exist proved-correct optimizing compilers from C to C minor and from C minor to machine language. We have redesigned C minor so that it is suitable for Hoare-logic reasoning, we have designed a Separation Logic for C minor, we have given a small-step operational semantics so that extensions to concurrent C minor 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 C minor. 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.

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. We will do this by finding a programming language L that can serve as a common denominator between several high-level languages.

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, micro-kernel, and RISC processor [1].

Leroy [2] has built and proved correct in Coq [3] a compiler called *CompCert* from an high-level intermediate language *C minor* to assembly language for the Power PC architecture. Blazy *et al.* have built and proved correct a translator from a subset of C to C minor [4]. 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.

C minor has a more "calculus-like" view of local variables and procedures, and C0 has a "storage-allocation" view. The calculus-like view will lead to easier reasoning about program transformations and easier use of C minor 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 non-optimizing C0 compiler and can accommodate in-line assembly code.

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

- L should have an operational semantics represented in a logical framework.
- There should be a proved-correct compiler from L to machine language.
- L should be usable as the high-level target language of a C compiler.
  - The semantics should be a *small-step* semantics (SOS), to support reasoning about input/output, concurrency, and non-termination.
  - L should be machine-independent over machines in the "standard model" of byte-addressable multiprocessors.
  - L should be a mid-level target language of an ML compiler, 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.
  - L should support an axiomatic Hoare 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.
  - L should be concurrent in at least the "standard model" of thread-based preemptive lock-synchronized weakly consistent shared-memory programming.

Leroy's original C minor 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.

We have redesigned C minor'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. But the main contributions of this paper 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. We believe this

is the first machine-checked Hoare-logic soundness proof for any full-sized language with a small-step semantics.
- A machine-checked big-step/small-step equivalence proof that allows us to use Leroy's existing proved-correct C minor compiler for (terminating sequential) programs proved correct in Separation Logic.
- Our sequential small-step soundness theorem will be usable (with minimal modification) in an extension of C minor to shared-memory concurrency with Concurrent Separation Logic.

2 Expression Semantics

The C standard [5] describes memory model that is byte- and word-addressible (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 C minor. In C, pointer arithmetic within any malloc'ed block is defined, but pointer arithmetic between different blocks is undefined; C minor therefore has pointer values comprising an abstract block-number and an int offset. Pointer arithmetic between blocks, and reading uninitialized variables, are undefined but not illegal: expressions in C minor can evaluate to *undefined* (Vundef) without getting stuck.

Each memory load or store is to a pointer value (block+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.

The *values* of C minor are *undefined*, integers, pointers, and floats. The int type is an abstract data-type of 32-bit modular arithmetic. The expressions of C minor are literals, variables, primitive operators applied to arguments, and memory loads. We will write $\mathsf{Econs}\, e_1\, (\mathsf{Econs}\, e_2\, \mathsf{Enil})$ as $[e_1, e_2]$.

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

There are 33 primitive $\mathsf{operation}$ symbols op; two of these are for accessing global names and local stack-blocks, and the rest are for integer and floating-point arithmetic and comparisons. C minor 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 $\mathsf{Evar}\, id$ or in a global scope using $(\mathsf{Eop}\, (\mathsf{Oaddrsymbol}\ id\ 0)\, \mathsf{Enil})$.

Expression evaluation in Leroy's C minor 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. The next release (8.2) of Coq is expected to have much better support for functional induction.

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 ($\mathsf{ident} \to \mathsf{option\,block}$) 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 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 : $\mathsf{footprint}$. A mapping from memory addresses to permissions.
- m : mem. The memory, a finite mapping from block to $\mathsf{block\_contents}$ [6]. Each block represents the result of a C `malloc` (or a stack frame, a global static variable, or a function code-pointer); in the "real world" it is a word-aligned contiguous sequence of bytes within which pointer arithmetic is defined. 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.
- e : expr. The expression being evaluated.

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

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

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

Fig. 1. Expression evaluation rules

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 $v_1 \in^{ch}_{\mathsf{load}} \phi$ (or $v_1 \in^{ch}_{\mathsf{store}} \phi$) to mean that all the addresses from v_1 to $v_1 + |ch| - 1$ 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, with properties such as $v \in^{ch}_{\mathsf{store}} \phi_0 \Rightarrow v \notin^{ch}_{\mathsf{load}} \phi_1$, $v \in^{ch}_{\mathsf{load}} \phi_0 \Rightarrow v \in^{ch}_{\mathsf{load}} \phi$ and $v \in^{ch}_{\mathsf{store}} \phi_0 \Rightarrow v \in^{ch}_{\mathsf{store}} \phi$. 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 two or more load permissions, which can be reconstituted to obtain a store permission.

Leroy's semantics for C minor did not have footprints, as they are not necessary for the correctness of a compiler (for a sequential language). We add the footprints for use in reasoning about separation, and we show the relation between the language with footprints and the original (see Lemma 6).

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

Sequential states. We shall bundle together $(\Psi; sp; \rho; \phi; m)$ and call it the *sequential state*, written as σ. We write $\sigma \vdash e \Downarrow v$ to mean $\Psi_\sigma; sp_\sigma; \rho_\sigma; \phi_\sigma; m_\sigma \vdash e \Downarrow v$. We write $\sigma[:= \rho']$ to mean the state σ with its environment component replaced by ρ', and so on.

Remark 1. $\Psi; sp \vdash op(vl) \Downarrow_{\mathsf{eval\_operation}} v$ and $m \vdash v_1 \overset{ch}{\mapsto} v$ are both deterministic relations, i.e., functions.

Lemma 2. *$\sigma \vdash e \Downarrow v$ is a deterministic relation. (Trivial by inspection.)*

Lemma 3. *For any value v, there is an expression e such that $\forall \sigma.\ (\sigma \vdash e \Downarrow v)$.*

Proof. Obvious and trivial; 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 C minor for* Vundef *and* Vptr *values.*

Example 4. The expression Eop Oadd [Eload Mint32 (Evar x), Eval (Vint(Int.repr 1))] fetches a 32-bit integer from the address contained in variable x, and adds 1.

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 C minor) by guaranteeing that the footprints ϕ of different threads are disjoint.

Definition 5 (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 $v_1 \in^{ch}_{\mathsf{load}} \phi$ of the rule for* Eload.

Lemma 6. *If $\Psi; sp; \rho; \phi; m \vdash e \Downarrow v$ then $\Psi, sp, \rho, m \vdash e \dot{\Downarrow} v$.*

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

3 Sequential Statement Semantics

The statements of sequential C minor 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. To model exits from nested loops, $\mathsf{block}\, s$ runs s, which should not terminate normally but which should $\mathsf{exit}\, n$ from the nth 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. [3] $\mathsf{return}\, el$ 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.

Function definitions A C minor 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 $\mathsf{f} = (xl, yl, n, s)$, where $\mathsf{params}(\mathsf{f}) = xl$ is a list of formal parameters, $\mathsf{locals}(\mathsf{f}) = yl$ is a list of local variables, $\mathsf{stackspace}(\mathsf{f}) = n$ is the size of the local stack-block to which $\mathsf{body}(\mathsf{f}) = sp$ points, and the statement 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).

A continuation k has a state σ and a control stack κ. There are sequential control operators to handle local control flow (Kseq), 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.

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

The sequential small-step function takes the form $k \longmapsto k'$ (see Figure 2), and we define as usual the reflexive transitive closure $\longmapsto^*$. The small-step execution of a program Ψ is of the form $((\Psi; 0; \{\}; \phi_0; m_0), s_0 \cdot \mathsf{Kstop}) \longmapsto^* (\sigma', \mathsf{Kstop})$ where s_0 calls the "main" function: $s_0 = \mathsf{call}\, [] \, (\mathsf{Eop}(\mathsf{Oaddrsymbol}\ main)[])\, []$.

[3] Each function in the real C minor 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.

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

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

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

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

$$\frac{}{\vdash (\sigma, (\mathsf{block}\, s)\cdot\kappa)\ \longmapsto\ (\sigma, s\cdot\mathsf{Kblock}\,\kappa)} \qquad \frac{j\geq 0}{\vdash (\sigma, \mathsf{exit}\, 0\cdot s_0\cdot\cdots s_j\cdot\mathsf{Kblock}\,\kappa)\ \longmapsto\ (\sigma,\kappa)}$$

$$\frac{j\geq 0}{\vdash (\sigma, \mathsf{exit}\,(n+1)\cdot s_0\cdot\cdots s_j\cdot\mathsf{Kblock}\,\kappa)\ \longmapsto\ (\sigma, \mathsf{exit}\, n\cdot\kappa)}$$

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

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

Fig. 2. Sequential small-step relation

Lemma 7 (substitution). *If $\sigma\vdash e\Downarrow v$ then $(\sigma,(x := e)\cdot\kappa)\longmapsto k'$ iff $(\sigma,(x := \mathsf{Eval}\, v)\cdot\kappa))\longmapsto k'$ (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 C minor.

Definition 8. *A continuation $k = (\sigma,\kappa)$ is* stuck *if $\kappa\neq\mathsf{Kstop}$ and there does not exist k' such that $k\longmapsto^* k'$. A sequential state is* safe *if it cannot reach a stuck state in the sequential small-step relation $\longmapsto^*$.*

4 Sequential Reasoning about Sequential Features

Concurrent C minor, 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 [8] 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 to 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 C minor that assumes noninterference (and gets "stuck" on interference) [9].
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 paper. In fact, for *all* of the sequential operators defined in this paper,
$$\frac{(\sigma, \kappa) \longmapsto (\sigma', \kappa')}{(\omega, \sigma, \kappa) \longmapsto (\omega, \sigma', \kappa')}$$
that is, the oracle is not consulted. For the concurrent operators [9] 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 C).
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 C minor as an extension of the Sequential Separation Logic. Its soundness proof uses the sequential soundness proof as a lemma [9].
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 C minor, we will use a sequential C minor compiler equipped with a proof that it compiles non-interfering source threads into equivalent non-interfering 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 paper.
7. We will demonstrate, with respect to a formal model of weak-memory-consistency microprocessor, that non-interfering 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 paper. Previous machine-verified soundness proofs for Hoare Logic [10] and Separation Logic [11] 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 non-local-exit loop.

5 Sequential 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 post-condition. The assertions P and Q are predicates on the program state. We will 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 $P\sigma$ is a proposition of logic.

Assertion operators In Figure 3 we define the usual operators of Separation Logic: separating conjunction $*$, the empty assertion **emp**, **false**, imprecise **true**, disjunction $\vee$, and conjunction $\wedge$. Then we define some novel operators such as expression-evaluation $e \Downarrow v$ and base-logic propositions $\lceil A \rceil$.

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

Fig. 3. Operators of Separation Logic

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

Memory reads are done by a command of the language, not within expressions. In C minor we relax this restriction; expressions can read the heap. But we say 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 $e \Downarrow v$ to mean that expression e evaluates to value v in the operational semantics. What we mean by $\lceil e \rceil_{\mathsf{expr}}$ is that e evaluates to a true value (a nonzero integer or non-null pointer). We will usually omit the $\lceil\ \rceil_{\mathsf{expr}}$ braces in our notation, following Hoare's example.

C minor's integer equality operator $\mathsf{Eop}\,(\mathsf{Ocmp}\,\mathsf{Ceq})\,[e_1, e_2]$, 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;[4] 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* (short int, long int, float, etc.): $e_1 \overset{ch}{\mapsto} e_2$ means that at address e_1 in memory there is a defined value e_2 of the given chunk-type.

The Hoare Sextuple. In C minor there are commands that call functions, and commands that exit (from a block) or return (from a function). Thus we extend the Hoare triple 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 → assert is the *return environment*, giving the current function's post-condition as a predicate on the list of returned values; and
B : nat → assert is the *block environment* giving the exit conditions of each block statement in which the statement s is nested.

We write $P \Rightarrow Q$ when, for all σ we have $P\,\sigma \;\Rightarrow\; Q\,\sigma$. The rules of sequential Separation Logic are given in Figure 4.

The statement exit i exits from the ith 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 $B' = P \cdot B$ for the environment such that $B'_0 = P$ and $B'_{i+1} = B_i$.

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

[4] 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.

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

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

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

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

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

$$\frac{P \Rightarrow I \qquad \Gamma; R; B \vdash \{I\}s\{I\}}{\Gamma; R; B \vdash \{P\}\mathsf{loop}\ s\{Q\}} \qquad \frac{}{\Gamma; R; B \vdash \{B(n)\}\mathsf{exit}\ n\{Q\}}$$

$$\frac{\Gamma; R; Q \cdot B \vdash \{P\}s\{\mathbf{false}\}}{\Gamma; R; B \vdash \{P\}\mathsf{block}\ s\{Q\}} \qquad \frac{}{\Gamma; R; B \vdash \{\exists vl.\, el \Downarrow vl\ *\ R(vl)\}\mathsf{return}\ el\{Q\}}$$

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

Fig. 4. Sequential Separation Logic

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 $e \overset{\mathsf{fun}}{\rightarrow} \Theta$ meaning that the e evaluates to a function-pointer that is callable with pre and post-conditions given by Θ.

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

The rule for $\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$ requires that e be a pure expression. To reason about an if-statement where e is impure, one reasons by program transformation using the following rule. It is not necessary to rewrite the actual source program, it is only the local reasoning that is by program transformation.

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

6 Soundness of Sequential 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!

We say an assertion P *guards* a control κ (written as $P \,\square\, \kappa$) means that whenever P holds, it is safe to execute κ. That is, $P \,\square\, \kappa \ = \ \forall\sigma.\, P\sigma \Rightarrow \mathrm{safe}(\sigma, \kappa)$.

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:

$R \,\boxed{\mathbb{R}}\, \kappa \;=\; \forall vl, R(vl) \,\square\, \mathsf{return}\; vl \cdot \kappa \qquad\qquad B \,\boxed{\mathbb{B}}\, \kappa \;=\; \forall n, B(n) \,\square\, \mathsf{exit}\; n \cdot \kappa$

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

$$\Gamma; R; B \vdash \{P\}s\{Q\} \;=\; \forall \kappa.\;\; R \,\boxed{\mathbb{R}}\, \kappa \wedge B \,\boxed{\mathbb{B}}\, \kappa \wedge Q \,\square\, \kappa \;\Rightarrow\; P \,\square\, s \cdot \kappa$$

The rules of Figure 4 are lemmas that we prove by case analysis from this definition. Lemma 10 is used for statements made of expressions. Let us note that its left to right implication is an instance of Lemma 6. Lemma 11 is used in the proof of the if rule. Lemmas 12 to 13 are used in the proof of the sequence rule. This proof is detailed in Appendix A.

Lemma 10. *If* $\mathsf{pure}\,(e)$ *then* $(\sigma \vdash e \Downarrow v) \Leftrightarrow (\dot{\sigma} \vdash e \dot{\Downarrow} v)$.

Lemma 11. *If* $\sigma \vdash e \Downarrow v$ *and* $\phi_\sigma \subset \phi'$ *then* $\sigma[:= \phi'] \vdash e \Downarrow v$.

Lemma 12. *If* $P \,\square\, s_1 \cdot s_2 \cdot \kappa$ *then* $P \,\square\, (s_1; s_2) \cdot \kappa$.

Lemma 13. *1. If* $R \,\boxed{\mathbb{R}}\, \kappa$ *then* $\forall s, R \,\boxed{\mathbb{R}}\, s \cdot \kappa$.
2. If $B \,\boxed{\mathbb{B}}\, \kappa$ *then* $\forall s, B \,\boxed{\mathbb{B}}\, s \cdot \kappa$.

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, $(\sigma, s{\cdot}\kappa) {\longmapsto}^n (\sigma', \kappa')$. 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,

$\mathsf{Kstop} \circ \kappa =_{\text{def}} \kappa \qquad\qquad \mathsf{Kblock}\, \kappa' \circ \kappa =_{\text{def}} \mathsf{Kblock}\, (\kappa' \circ \kappa)$

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

Definition 14 (statement absorption). *A statement s in state σ absorbs n steps if* $\forall j \leq n. \exists \kappa_{\text{prefix}} \exists \sigma'.\; \forall \kappa. (\sigma, s \cdot \kappa) \longmapsto^j (\sigma', \kappa_{\text{prefix}} \circ \kappa)$

Example 15. An exit statement by itself absorbs no steps (it immediately uses its control-tail), but $\mathsf{block}\, (\mathsf{exit}\, 0)$ can absorb 2 steps:
$(\sigma, \mathsf{block}\, (\mathsf{exit}\, 0) \cdot \kappa) \longmapsto (\sigma, \mathsf{exit}\, 0 \cdot \mathsf{Kblock}\, \kappa) \longmapsto (\sigma, \kappa)$

Lemma 16. *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 17. *We write* $(s\cdot)^n s'$ *to mean* $\underbrace{s \cdot s \cdot \ldots \cdot s \cdot}_{n} s'$.

Lemma 18. $$\frac{\Gamma; R; B \vdash \{I\}s\{I\}}{\Gamma; R; B \vdash \{I\}(s\cdot)^n\mathsf{loop}\,\mathsf{skip}\{\mathbf{false}\}}$$

Proof. For $n = 0$, *the infinite-loop* ($\mathsf{loop}\,\mathsf{skip}$) *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* $I \,\square\, (s\cdot)^n\mathsf{loop}\,\mathsf{skip}\cdot\kappa$. *We have* $R \,\mathbb{R}\, (s\cdot)^n\mathsf{loop}\,\mathsf{skip}\cdot\kappa$ *and* $B \,\mathbb{B}\, (s\cdot)^n\mathsf{loop}\,\mathsf{skip}\cdot\kappa$ *by Lemma 13. Therefore by* $\{I\}s\{I\}$ *we have:* $I \,\square\, n\cdot(s\cdot)^n\mathsf{loop}\,\mathsf{skip}\cdot\kappa$.

Theorem 19. $$\frac{\Gamma; R; B \vdash \{I\}s\{I\}}{\Gamma; R; B \vdash \{I\}\mathsf{loop}\,s\{\mathbf{false}\}}$$

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 18 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)* $\kappa_{prefix} \circ (s\cdot)^{n-1}\mathsf{loop}\,\mathsf{skip}\cdot\kappa$ *or* $\kappa_{prefix} \circ \mathsf{loop}\,s\cdot\kappa$. *Now, because* s *cannot absorb* $j+1$ *steps, we know that either* $\kappa_{prefix} = \mathsf{Kstop}$ *(because* s *has terminated normally) or* κ_{prefix} *starts with a* return *or* exit, *in which case we escape (past the* $\mathsf{loop}\,\mathsf{skip}$ *or the* $\mathsf{loop}\,s$, *respectively) into* κ. *If* $\kappa_{prefix} = \mathsf{Kstop}$ *then we apply induction on the case* $n - j$; *if we escape, then* (σ', κ) *is safe iff* (σ', κ) *is safe.*

Corollary 20 (the loop **rule).**

$$\frac{P \Rightarrow I \qquad \dfrac{\Gamma; R; B \vdash \{I\}s\{I\}}{\Gamma; R; B \vdash \{I\}\mathsf{loop}\,s\{\mathbf{false}\}}\ \textsc{Thm. 19} \qquad \mathbf{false} \Rightarrow Q}{\Gamma; R; B \vdash \{P\}\mathsf{loop}\,s\{Q\}}$$

7 Big-step Semantics

Leroy's big-step semantics of sequential C minor is of the form $\vdash (\sigma, s) \Downarrow_{out} \sigma'$, where the outcome *out* expresses how the statement s has terminated: either normally by falling through the next statement or prematurely through either an exit or a return statement [2]. The execution of a program is of the form $\vdash p \Downarrow v$. Executing a program yields the value returned by main.

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, small-step with program-points) to specify the small-step semantics. This proof relies on the lemmas 21 to 24. For lack of space in this paper, lemmas 21 and 22 are simpler than the lemmas that we have proved in Coq and the definition of the functions stmt_of_outcome and outcome_of_statement is omitted (see Appendix B).

Lemma 21 (big-step implies small-step, induction lemma).
If $\vdash (\sigma, s) \Downarrow_{out} \sigma'$, *then* $\forall \kappa, \vdash (\sigma, s \cdot \kappa) \longmapsto^* (\sigma', s' \cdot \kappa) \wedge s' = \mathsf{stmt\_of\_outcome}(out)$.

Proof. By induction on the derivation of $\Downarrow$.

The reverse induction lemma relies on a big-step semantics with continuations that is defined by the following rule. κ represents the statements that need to be executed after s in order to get the outcome out_2. The judgment $\vdash (\sigma_1, \kappa); out_1 \rightsquigarrow \sigma_2; out_2$ allows the execution of the statements that belong to the control κ and can be read as: the outcome out_1 transmitted to the control κ yields an outcome out_2. This judgment is defined in Figure 5 of Appendix B.

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

Lemma 22 (small-step implies big-step, main induction lemma).
If $(\sigma, s \cdot \kappa) \longmapsto^* (\sigma', s' \cdot \kappa)$ *then* $\vdash (\sigma, s) \Downarrow_{out} \sigma' \wedge out = \mathsf{outcome\_of\_stmt} s'$.

Proof. This proof relies on the following lemma.

Lemma 23. *If* $(\sigma, s \cdot \kappa) \longmapsto (\sigma_1, s_1 \cdot \kappa_1)$ *and* $\vdash (\sigma_1, s_1 \cdot \kappa_1) \Downarrow_{out} \sigma'$ *then* $\vdash (\sigma, s \cdot \kappa) \Downarrow_{out} \sigma'$.

Proof. By induction on the derivation of $\longmapsto$, *using lemmas such as Lemma 24.*

Lemma 24. *If* $\vdash (\sigma, \mathsf{exit}\, n \cdot \kappa) \Downarrow_{out} \sigma'$ *then* $\vdash (\sigma, \mathsf{exit}\, (n+1) \cdot (\mathsf{Kblock}\, \kappa)) \Downarrow_{out} \sigma'$.

Proof. By case analysis.

Theorem 25 (Semantic preservation).
If the program p *is safe, then* $\vdash p \Downarrow v \;\Leftrightarrow\; \vdash p \longmapsto^* v$.

Erasure. We define an *erased state* $\dot{\sigma}$ as (Ψ, sp, ρ, κ); that is, just like a sequential state but without ϕ. We define an *erased continuation* as a pair $(\dot{\sigma}, \kappa)$. We define the *erased sequential small-step relation* $\stackrel{\cdot}{\longmapsto}$ derived by erasing from the $\longmapsto$ any mention of ϕ and any premises depending on ϕ.

Lemma 26. *If* $k \longmapsto k'$ *then* $k \stackrel{\cdot}{\longmapsto} k'$.

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

Theorem 27 (Compiler correctness [Leroy]). *If* $\vdash p \Downarrow v$ *in Leroy's big-step semantics, then the CompCert compiler either fails to compile the program* p *to any machine code at all, or compiles* p *to PowerPC machine code that evaluates to the same result* v.

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 28. *If a sequential 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 Conclusion

Small-step reasoning is useful for sequential programming languages that will be extended with concurrent features; but small-step reasoning about non-local control constructs (return, exit) 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 C) which will allow the theorems in this paper to be used even in a concurrent language. We strongly recommend this approach, as it cleanly separates the difficult problems in non-local control flow from the problems of concurrent access to shared memory.

References

1. 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*, 2005.
2. 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.
3. The Coq proof assistant. http://coq.inria.fr.
4. Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *Symp. on Formal Methods*, pages 460–475, 2006.
5. *American National Standard for Information Systems – Programming Language – C*. American National Standards Institute, 1990.
6. Sandrine Blazy and Xavier Leroy. Formal verification of a memory model for C-like imperative languages. In *Formal Engineering Methods*, 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. Stephen Brookes. A grainless semantics for parallel programs with shared mutable data. In *Mathematical Foundations of Programming Semantics*, May 2006.
9. Andrew W. Appel, Aquinas Hobor, and Francesco Zappa Nardelli. Oracle semantics for Concurrent C minor. in preparation, 2006.
10. Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.
11. 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 A: Soundness of the Sequence Statement

The soundness of the sequence statement is the proof that if $\mathsf{H}_1 : \Gamma; R; B \vdash \{P\}s_1\{P'\}$ and $\mathsf{H}_2 : \Gamma; R; B \vdash \{P'\}s_2\{Q\}$ hold, then we have $\mathsf{Goal} : \Gamma; R; B \vdash \{P\}s_1; s_2\{Q\}$ (see Figure 4). If we unfold the definition of the Hoare sextuples, H_1, H_2 and Goal become:

$$(\forall k_1)\frac{R\,\mathbb{R}\,k_1 \qquad B\,\mathbb{R}\,k_1 \qquad P'\,\square\,k_1}{P\,\square\,s_1\cdot k_1}\ \mathsf{H}_1$$

$$(\forall k_2)\frac{R\,\mathbb{R}\,k_2 \qquad B\,\mathbb{R}\,k_2 \qquad Q\,\square\,k_2}{P'\,\square\,s_2\cdot k_2}\ \mathsf{H}_2$$

$$(\forall k)\frac{R\,\mathbb{R}\,k \qquad B\,\mathbb{R}\,k \qquad Q\,\square\,k}{P\,\square\,(s_1;s_2)\cdot k}\ \mathsf{Goal}$$

The proof that $P\,\square\,(s_1;s_2)\cdot k$ uses the lemmas 12 and 13 (see Section 6) as follows.

$$\dfrac{\dfrac{R\,\mathbb{R}\,k}{R\,\mathbb{R}\,s_2\cdot k}\ \mathsf{Lm.\ 13.1} \qquad \dfrac{B\,\mathbb{R}\,k}{B\,\mathbb{R}\,s_2\cdot k}\ \mathsf{Lm.\ 13.2} \qquad \dfrac{R\,\mathbb{R}\,k \qquad B\,\mathbb{R}\,k \qquad Q\,\square\,k}{P'\,\square\,s_2\cdot k}\ \mathsf{H}_2}{\dfrac{P\,\square\,s_1\cdot s_2\cdot k}{P\,\square\,(s_1;s_2)\cdot k}\ \mathsf{Lm.\ 12}}\ \mathsf{H}_1$$

Appendix B: Semantic Preservation

In a big-step semantics judgment $\vdash (\sigma, s) \Downarrow_{out} \sigma'$, the outcome out expresses how the statement s has terminated: either normally ($\mathsf{Out_{normal}}$) by falling through the next statement or prematurely through either an exit ($\mathsf{Out_{exit}\,n}$) or a return statement ($\mathsf{Out_{return}}\,vl$).

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

The two functions $\mathsf{stmt\_of\_outcome}$ and $\mathsf{outcome\_of\_stmt}$ express that the statements skip and $\mathsf{exit}\,n$ correspond respectively to the outcomes $\mathsf{Out_{normal}}$ and $\mathsf{Out_{exit}\,n}$. We also have:
$\mathsf{stmt\_of\_outcome}(\mathsf{Out_{return}}\,vl) = \mathsf{return}\,(\mathsf{Eval}vl)$
$\mathsf{outcome\_of\_stmt}(\mathsf{return}\,le) = \mathsf{Out_{return}}\,(\mathsf{Out_{return}}\,vl)$ for vl such that $\sigma \vdash e \Downarrow vl$.

Lemma 29 (generalization of lemma 21). *If* $\vdash (\sigma, s) \Downarrow_{out} \sigma'$, *then* $\forall \kappa, \kappa'$, *such that* $\mathsf{cont\_of}(\kappa, \mathsf{stmt\_of\_outcome}(out)) = \kappa'$, *we have* $\vdash\ (\sigma, s\cdot\kappa) \longmapsto^* (\sigma', \kappa')$.

This induction lemma states that for any big-step execution of a statement s and any control κ, there exists a corresponding small-step execution from $s\cdot\kappa$ to κ',

where κ' is defined by the cont_of function as follows. Compared to Lemma 21, κ' is of the form $s' \cdot \kappa$ only when s is neither skip nor exit or return.

$$\begin{array}{l} s = \mathsf{skip} \wedge \kappa' = \kappa \quad \vee \quad s \notin \{\mathsf{skip}, \mathsf{exit}\, n\} \wedge \kappa' = s' \cdot \kappa \quad \vee \\ s = \mathsf{exit}\, n \wedge j \geq 0, \kappa = s_0 \cdot s_1 \cdot \ldots s_j \cdot \mathsf{Kblock}\, \kappa'' \wedge \\ \qquad (n = 0 \wedge \kappa' = \kappa) \vee (n > 0 \wedge \kappa' = \mathsf{exit}\,(n-1) \cdot \kappa'') \\ s = \mathsf{return}\, l \wedge j \geq 0, \kappa = s_0 \cdot s_1 \cdot \ldots s_j \cdot Kcall\, xl\, f\, sp\, \rho\, \kappa'' \wedge \kappa' = \kappa'' \end{array}$$

Figure 5 defines judgments on the form $\vdash (\sigma, \kappa); out \rightsquigarrow \sigma'; out'$ that are called from the big-step semantics with continuations.

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

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

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

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

Fig. 5. Sequential big-step relation with control

Appendix C: Summary of Concurrent C minor

Appel, Hobor, and Zappa Nardelli have specified [9] Concurrent C minor 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 C minor*:

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

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 paper 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 paper 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 paper are still correct as explained in English, but in Coq they will need to be rephrased using deterministic (nonconstructive) relations, rather than constructive functions.