From Mechanized Semantics to Verified Compilation: the Clight Semantics of CompCert

Sandrine Blazy

From Mechanized Semantics to Verified Compilation: the Clight Semantics of CompCert

Sandrine Blazy[0000−0002−0189−0223]

Inria, Univ Rennes, CNRS, IRISA

Abstract. CompCert is a formally verified compiler for C that is specified, programmed and proved correct with the Coq proof assistant. CompCert was used in industry to compile critical embedded software. Its correctness proof states that the compiler does not introduce bugs. This semantic preservation property involves the formal semantics of the source and target languages of the compiler.
Reasoning on C semantics to prove compiler correctness is challenging, as C is a real language that was not designed with semantics in mind. This paper presents the operational style that was designed for the C semantics of CompCert in order to facilitate the mechanized reasoning on terminating and diverging programs, and details the semantics of the Clight source language of CompCert.

Keywords: operational semantics of programming languages · verified compilation · machine-checked proofs

1 Introduction

Deductive verification provides very strong mathematical guarantees that a piece of software is correct with respect to its specification, written in a logical language to avoid ambiguities. A proof is conducted to provide these guarantees. The outcome of deductive verification is a verified software, consisting of an implementation and a proof that can be replayed or given to a certification authority for scrutiny. This proof requires reasoning on properties related to the involved programming language; they become mathematically precise as soon as this language has formal semantics. Defining and reasoning on realistic languages requires mechanized semantics and machine-checked proofs, ensuring that the proof is complete and that no semantic rule has been forgotten.

There are mainly two families of deductive proof tools (also known as program provers), each with its pros and cons: automatic tools (such as Dafny [22], F* [30] or Why3 [15]) where formulas (expressing pre- and post-conditions and invariants) are discharged to logic solvers, and interactive proof assistants (*e.g.*, Coq [17], Isabelle [2] or Agda [1]) where the user decides how to reason and conducts the proof interactively with the tool, that automates part of the reasoning, ensures that the proof is complete and follows the laws of mathematical logic. Automatic program provers are easier to use when the discharged formulas are proved without requiring extra work (namely adding assertions to help the logic solvers).

However, when program provers fail to prove some formulas, interactive proof assistants are better adapted to conduct more advanced proofs. A prototypical example is a proof requiring reasoning on a data structure that is not used by the software under scrutiny, but only defined for the sole purpose of the proof (see for instance the proof of correctness of the famous majority algorithm [12]).

One of the first programs whose proof was mechanized in LCF is a rudimentary compiler for arithmetic expressions [31]. In 1972, when this paper was published, a compiler was a representative example of a particularly complex program. The specification of a compiler is rather simple: the generated code must behave as prescribed by the semantics of the source program. This correctness property is a semantic preservation property from the source language to the target language of the compiler. It becomes mathematically precise as soon as these languages are defined by formal semantics.

Nowadays, the compiler remains a particularly complex piece of software (due to the numerous optimizations it performs to generate efficient code). Moreover, it is the mandatory point of passage in the software production chain. Verifying the compiler provides a means of ensuring that no errors are introduced during compilation, and of preserving at target level the guarantees obtained at source level. The idea of having a single theorem demonstrated once and for all, along with a readable proof, was already present in 1972, but it took several decades for verified compilation to develop and scale up.

CompCert is the first optimizing compiler for the C language targeting different assembly languages and used in safety-critical industries (to compile mission-critical embedded software used in avionics and nuclear power), with a mechanized proof of correctness [23, 27, 19]. In industry, the interest for CompCert arose from a need to improve the performances of the generated code, while guaranteeing the traceability requirements required by the certification authorities in force in these critical fields, which CompCert has indeed provided.

Developing a verified compiler requires both programming the compiler using the programming language of the proof assistant (so that it runs efficiently on real programs), and defining a semantic model and abstractions to reason about, in order to conduct the correctness proof. Mechanized reasoning on C-like languages is tricky; it requires a semantic style that is adapted to inductive reasoning and some associated reasoning principles. In CompCert, the chosen proof technique is the use of simulation diagrams between program executions, which required to define a new semantic model that is detailed in this paper. The semantic model and proof technique scale to realistic languages like C. They are general enough to be applied to all the intermediate languages of the compiler. The proof technique was extended and successfully reused in order to ensure other properties than CompCert correctness [5–7].

This paper is about mechanized operational semantics for compiler verification and their application to the CompCert compiler, with a focus on the Clight semantics, that significantly evolved since its first published version [9]. The Clight language is the preferred language to get guarantees from C programs and then compile them with CompCert (*e.g.*, [18, 13, 11, 8, 21, 16, 33]).

This paper aims at providing the prerequisites needed to design new program transformations or analyses operating over Clight.

All results presented in this paper have been mechanically verified using the Coq proof assistant [25, 32, 3]. This paper is organized as follows. First, Section 2 recalls the early days of compiler verification. Then, Section 3 introduces a small-step semantics for terminating programs written using a toy imperative language, together with the associated proof technique based on simulation diagrams. Section 4 extends this language and its semantics to observe diverging program executions; it defines an alternate semantics that facilitates the mechanized proofs. Section 5 defines the semantics of Clight. Related work is discussed in Section 6, followed by conclusions.

Notations. For functions returning "option" types, $\lfloor x \rfloor$ (read: "some x") corresponds to success with return value x, and ϵ (read: "none") corresponds to failure. In grammars and rules, a^* denotes 0, 1 or several occurrences of syntactic category a, and $a^?$ denotes an optional occurrence of syntactic category a. ϵ denotes the empty list, $[x]$ denotes a list made of a single element x and $h :: t$ denotes the list with head h and tail t. Th lisy $l + +l'$ denotes the concatenation of two lists l and l'. Given a binary relation R, R^* denotes its reflexive transitive closure and R^+ its transitive closure.

2 Historical Example: a First Verified Compiler

The idea of verifying a compiler and stating a theorem for compiler correctness dates back to 1967 [29]. The proof of this theorem was mechanized in 1972 using LCF [31]. This compiler translates in a single pass any simple arithmetic expression a to a code $\mathtt{p}$, namely a list of instructions of a simple stack machine (see Fig. 1); this is the familiar translation to reverse Polish notation used by old HP pocket calculators.

For instance, the expression 1+2 is compiled to the code $\mathtt{iconst}\,1 :: \mathtt{iconst}\,2 :: \mathtt{iplus} :: \epsilon$. The stack contains numbers and the machine instructions pop their arguments off the stack and push their results back. This machine is close to a subset of the Java virtual machine. The machine code for an expression a executes in sequence, and deposits the value of a at the top of the stack π. An instruction either pushes an integer, or pushes the current value of a variable, or pops two integers then pushes their sum.

The source and target languages are defined in Fig. 1 by their semantics. In [29], these are functions interpreting expressions or instructions. In this paper, we rather use inference rules to abstract away the definitions of all our semantics. The semantic judgments for evaluating expression a and executing code $\mathtt{p}$ are respectively $\sigma \vdash a \Rightarrow v$ and $\sigma, \pi \vdash \mathtt{p} \rightarrow \pi'$, where a semantic element, the store σ is injected to assign integer values to variables, and the evaluation stack π contains temporary integer values.

The correctness theorem of the compiler is Theorem 2: it states that for any expression a, its value v computed by the semantics of the source language

Arithmetic expressions:

$a ::= x \mid c \mid a + a$ source language (variable, integer constant, addition)

$$\text{CONSTANT}\quad \sigma \vdash c \Rightarrow c \qquad \text{VARIABLE}\quad \sigma \vdash x \Rightarrow \sigma(x) \qquad \text{ADDITION}\quad \frac{\sigma \vdash a_1 \Rightarrow v_1 \qquad \sigma \vdash a_2 \Rightarrow v_2}{\sigma \vdash a_1 + a_2 \Rightarrow v_1 + v_2}$$

VM instructions:

$i ::= \texttt{ivar}\, x \mid \texttt{iconst}\, c \mid \texttt{iplus}$ target language

$$\text{EMPTY STACK}\quad \sigma, \epsilon \vdash c \to c \qquad \text{CONSTANT}\quad \frac{\sigma, c :: \pi \vdash \mathsf{p} \to \pi'}{\sigma, \pi \vdash \texttt{iconst}\, c :: \mathsf{p} \to \pi'} \qquad \text{VARIABLE}\quad \frac{\sigma, \sigma(x) :: \pi \vdash \mathsf{p} \to \pi'}{\sigma, \pi \vdash \texttt{ivar}\, x :: \mathsf{p} \to \pi'}$$

$$\text{ADDITION}\quad \frac{\sigma, (m+n) :: \pi \vdash \mathsf{p} \to \pi'}{\sigma, n :: m :: \pi \vdash \texttt{iplus} :: \mathsf{p} \to \pi'} \qquad \text{OTHER}\quad \frac{\sigma, \pi \vdash \mathsf{p} \to \pi'}{\sigma, \pi \vdash i :: \mathsf{p} \to \pi'}$$

Translation from arithmetic expressions to machine code (compile function):

$$x \mapsto \texttt{ivar}\, x \qquad c \mapsto \texttt{iconst}\, c \qquad \frac{a_1 \mapsto i_1 \quad a_2 \mapsto i_2}{a_1 + a_2 \mapsto i_1 + +i_2 + +[\texttt{iplus}]}$$

Theorem 1 (first correctness). $\forall a\, \sigma\, \pi, \sigma \vdash a \Rightarrow v \;\to\; \sigma, \pi \vdash \mathsf{compile}(a) \to v :: \pi$

Proof. By induction on the structure of arithmetic expressions.

Theorem 2 (compiler correct). $\forall a\, \sigma, \sigma \vdash a \Rightarrow v \;\to\; \sigma, \epsilon \vdash \mathsf{compile}(a) \to [v]$

Proof. By theorem 1.

Fig. 1: Historical example: a first verified compiler.

is exactly the value returned by executing the compiled code $\mathsf{compile}(a)$. This theorem is proved only once, for any expression given as input to the compiler. The verification of this tiny compiler is now taught as an exercise in masters courses (*e.g.*, [25, 32]). It is an illustrative example of the need to generalize a theorem, so that it can be proved by induction (here on expressions). This explains why Theorem 1 is proved by induction on expressions and used to prove Theorem 2, the main theorem for compiler correctness.

3 A First Semantics for a Toy Imperative Language

The previous section defines a big-step semantics for a rudimentary language for arithmetic expressions. In this section, we first extend this language (into a toy imperative language called IMP), and then introduce simulation diagrams, a convenient proof technique for reasoning on IMP programs.

Boolean expressions:

$$b ::= \mathit{true} \mid \mathit{false} \mid a = a \mid a \leq a \mid\sim b \mid b \wedge b \quad \text{source language}$$

IMP commands:

$$\begin{array}{ll} c ::= \mathsf{skip} \mid x := a \mid c; c & \text{skip, assignment, sequence} \\ \quad \mid \mathtt{if}\,(b)\,c\,\mathtt{else}\,c \mid \mathtt{while}\,(b)\;c & \text{conditional, while loop} \end{array}$$

EQUALITY TEST

$$\frac{\sigma \vdash a_1 \Rightarrow v_1 \;\; \sigma \vdash a_2 \Rightarrow v_2}{\sigma \vdash a_1 + a_2 \Rightarrow v_1 + v_2}$$

NEGATION

$$\frac{\sigma \vdash b \Rightarrow v}{\sigma \vdash\sim b \Rightarrow\sim v}$$

AND

$$\frac{\sigma \vdash b_1 \Rightarrow v_1 \;\; \sigma \vdash b_2 \Rightarrow v_2}{\sigma \vdash b_1 + b_2 \Rightarrow v_1 + v_2}$$

ASSIGN

$$\frac{\sigma \vdash a \Rightarrow v}{(x := a, \sigma) \rightarrow (\mathsf{skip}, \sigma[x \rightarrow v])}$$

IF TRUE

$$\frac{\sigma \vdash b \Rightarrow \mathit{true}}{(\mathtt{if}\,(b)\,c_1\,\mathtt{else}\,c_2, \sigma) \rightarrow (c_1, \sigma)}$$

IF FALSE

$$\frac{\sigma \vdash b \Rightarrow \mathit{false}}{(\mathtt{if}\,(b)\,c_1\,\mathtt{else}\,c_2, \sigma) \rightarrow (c_2, \sigma)}$$

SEQUENCE DONE

$$(\mathsf{skip}; c, \sigma) \rightarrow (c, \sigma)$$

SEQUENCE

$$\frac{(c_1, \sigma_1) \rightarrow (c_2, \sigma_2)}{(c_1; c, \sigma_1) \rightarrow (c_2; c, \sigma_2)}$$

WHILE DONE

$$\frac{\sigma \vdash b \Rightarrow \mathit{false}}{(\mathtt{while}\,(b)\;c, \sigma) \rightarrow (\mathsf{skip}, \sigma)}$$

WHILE LOOP

$$\frac{\sigma \vdash b \Rightarrow \mathit{true}}{(\mathtt{while}\,(b)\;c, \sigma) \rightarrow (c; (\mathtt{while}\,(b)\;c), \sigma)}$$

Fig. 2: IMP operational semantics: big-step semantics for expressions, and small-step semantics for commands.

3.1 Small-step Semantics

IMP is made of arithmetic expressions (reused from Section 2), boolean expressions and commands (skip, assignment, sequence, conditional and loop). Boolean expressions are used in conditionals and loops. IMP is defined in Fig. 2, where the semantics of arithmetic expressions defined in Fig. 1 is reused.

Semantics observe the possible behaviors of programs and are defined using an operational style, that is the preferred style for machine-checked reasoning about semantics. Operational semantics consist of big-step semantics and small-step semantics, and both styles are equivalent. Moreover, proving this equivalence is a valuable way of getting confidence in the semantics and supporting both styles may be interesting, as it offers the possibility of choosing the most appropriate one for different needs.

Choosing a style may be a matter of taste. However, big-step semantics are not adapted to define in a natural way some semantic features such as unstructured control, diverging and concurrent executions, whereas small-step semantics are more suitable. Because of while loops (*e.g.*, $\mathtt{while}\,(\mathit{true})\;\mathsf{skip}$), the execution of IMP programs may diverge, contrary to the evaluation of IMP expressions.

So, we rather choose small-step semantics to define IMP commands, and big-step semantics to define IMP expressions.

The small-step semantics is a reduction semantics between semantic states. A semantic state is a pair (c, σ) made of a command and a store. The semantics takes the form of a relation $(c, \sigma) \to (c', \sigma')$, where a command c is reduced into a command c' in an execution step. The c' command represents all the remaining steps and σ' is the store resulting from this computation step. The execution of a sequence of commands $c_1; c_2$ first iterates the reduction of c_1 until the final reduction to skip. Then, c_2 is reduced. The execution of a while loop unfolds the loop when its body is executed at least once. So, this rule generates a sequence of commands that will be further reduced.

The evaluation of expressions always terminates and the big-step semantics of expressions observe these terminating behaviors. Contrary to big-step semantics, small-step semantics observe in a similar and convenient way terminating executions of commands together with diverging executions. The reflexive transitive closure $\to^*$ of this step relation is used to chain the finite transition sequences. In a similar way, $\to^\infty$ is used to chain infinite execution steps. Given initial and final stores σ_i and σ_f, the termination of a command c is defined as $\mathsf{terminates}(\sigma_i, c, \sigma_f) \triangleq (c, \sigma_i) \to^* (\mathsf{skip}, \sigma_f)$: c terminates when it is reduced to a skip command. Given an initial store σ, the diverging execution of a command c is defined as $\mathsf{diverges}(\sigma_i, c) \triangleq (c, \sigma_i) \to^\infty$: all transition sequences starting from σ_i are infinite.

Moreover, the semantics observe a third kind of behaviors, going wrong behaviors (or abnormal termination), that happen for instance because of a division by zero. Given a command c and a store σ, this behavior is defined as $\mathsf{goeswrong}(\sigma, c) \triangleq \exists c', \exists \sigma'. (c, \sigma) \to^* (c', \sigma') \wedge (c', \sigma') \not\to \wedge\, c' \neq \mathsf{skip}$: after a finite number of execution steps to (c', σ'), this state cannot reduce (written $\not\to$) and it is not a final state as c' differs from the skip command. However, abnormal termination is not preserved by verified compilation, as compiler optimizations may remove instructions leading to going wrong behaviors [24].

3.2 Reasoning on Operational Semantics: Simulation Diagrams

From a proof point of view, with big-step semantics, the proof follows naturally the structure of programs and is conveniently conducted by induction on derivations of big-step executions. With small-step semantics, the standard proof technique is to rely on simulation diagrams between semantic states and involving invariants defining matching states. Proving a simulation requires reasoning by case analysis on each possible step. An interesting property of simulations is that they are compositional: they are chained together to describe complete program executions. Thus, the proof of correctness of a compiler pass mainly amounts to the proof of a simulation, and the tricky part often consists in finding the right invariants to preserve.

The choice between a big-step and a small-step style simply on the basis of the adequacy to describe semantic features sometimes comes at the expense of the choice of the proof technique. As an example, choosing a small-step style to

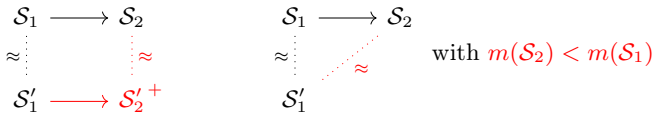

Fig. 3: Forward-simulation diagram with measure. Black lines are hypotheses, red lines are conclusions.

represent in a convenient way diverging executions of IMP prevents the use of standard simulations. Indeed, these simulations also represent the troublesome situation where infinitely many consecutive steps in the source program are simulated by no step at all in the target program. Such situations denote incorrect program transformations, since some diverging behaviors are simulated by some terminating behaviors. In order to handle diverging execution steps and rule out this infinite stuttering problem, a common solution is to strengthen the invariant of the simulation with the definition of a well-founded measure (over the states of the source language) that for instance strictly decreases in cases where stuttering could occur.

An example of a simulation diagram is the forward simulation diagram shown in Fig. 3 and expressed in the following theorem. Given a program P_1 and its transformed program P_2, each transition step in P_1 (from semantic state $\mathcal{S}_1$ to semantic state $\mathcal{S}_2$) must correspond to transitions in P_2 (from semantic state $\mathcal{S}'_1$ to semantic state $\mathcal{S}'_2$) and preserve as an invariant a relation $\approx$ between semantic states of P_1 and P_2. The measure $m(\cdot)$ is defined over the states of P_1 and strictly decreases in cases where stuttering could occur. The diagram ensures that if the source program diverges, it must perform infinitely many non-stuttering steps, so the compiled code executes infinitely many transitions.

4 Continuation-based Small-step Semantics for IMP

Proving simulation diagrams is a general and convenient technique to reason on small-step semantics. This section explains how the simulation diagram defined in Section 3 can be used to reason on a toy imperative language extended with statements. Semantics describe the dynamic of programs, in contrast to compiler passes, which are statically defined, for any source program. A simulation relates the two, by expressing that target execution steps must correspond to source execution steps. One issue with standard small-step semantics is that they describe intermediate steps involving new commands that are subcommands of the source program (*e.g.*, the last rule of Fig. 2).

A consequence of this spontaneous generation of commands is that the reasoning required to prove a simulation becomes difficult and complicates the definition of the anti-stuttering measure. This section first defines an alternative small-step semantics for IMP that is better adapted to mechanized reasoning. Then, it shows that it is equivalent to the first small-step semantics.

4.1 Semantic Rules

The solution adopted in CompCert is to define an original small-step style based on continuations, where the new semantic states become triples, as the command to be executed is explicitly decomposed into a sub-command c under focus, where computation takes place, and a context k that describes the position of the sub-command in the whole command; or, equivalently, a continuation that describes the parts of the whole command that remain to execute once the sub-command terminates. More precisely, the semantic states become of the shape (c, k, σ), and the semantic judgment becomes $(c, k, \sigma) \rightsquigarrow (c', k', \sigma')$. Continuations k are of three kinds, defined in Fig. 4.

- The continuation stop means that nothing remains to be done once the sub-command terminates. In other words, the sub-command under focus is the whole command. This happens either at the beginning or at the end of a program execution.
- A continuation $c; k$ means that when the sub-command terminates, we will then execute the command c, then continue as described by k.
- A continuation $\circlearrowleft(b, c, k)$ means that when the sub-command c terminates, we will then execute the loop $\texttt{while}\,(b)\ c$. When this loop terminates, we will continue as described by k.

Dealing with continuations requires adding new semantic rules to define the execution of commands. The evaluation of expressions remains unchanged. In the end, there are three kinds of semantic rules (see Fig. 4):

- Computation rules evaluate arithmetic and boolean expressions, and modify the triple accordingly. They are close to the rules of the previous semantics.
- Focusing rules describe how to replace the sub-command by a sub-sub-command that must be executed first, enriching the continuation accordingly.
- Resumption rules describe how to extract a continuation in order to execute the next sub-command. More precisely, when the sub-command under focus is skip, and therefore has terminated, resumption rules examine the head of the continuation to find the next sub-command to focus on.

The semantics if IMP rules defines two focusing rules, one for sequences and one for loops. Focusing on a sequence means executing its left part, while pushing the right part to the current continuation. Focusing on a loop means executing its body, while pushing the loop to the current context. The semantics also defines two resumption rules. The resumption rule for a sequence is triggered when its left part is reduced to the skip command; it then steps to the right part of the sequence. The resumption rule for a loop steps to the next execution of the loop body.

Thanks to continuations, semantic rules become genuine reduction rules. For instance, an if command is now rewritten into a sub-command, namely one of its branches. Moreover, as in the previous small-step semantics, termination and divergence are defined using transition sequences. Initial semantic states are of

Continuations:

$$k ::= \mathsf{stop} \mid c; k \mid \circlearrowleft(b, c, k) \quad \text{stop, sequence, while}$$

$$\textsc{assign (computation)}\ \frac{\sigma \vdash a \Rightarrow v}{(x := a, k, \sigma) \rightsquigarrow (\mathsf{skip}, k, \sigma[x \to v])} \qquad \textsc{sequence (focusing)}\ ((c_1; c_2), k, \sigma) \rightsquigarrow (c_1, c_2; k, \sigma)$$

$$\textsc{if true (computation)}\ \frac{\sigma \vdash b \Rightarrow \mathit{true}}{(\mathtt{if}\,(b)\ c_1\ \mathtt{else}\ c_2, k, \sigma) \rightsquigarrow (c_1, k, \sigma)} \qquad \textsc{if false (computation)}\ \frac{\sigma \vdash b \Rightarrow \mathit{false}}{(\mathtt{if}\,(b)\ c_1\ \mathtt{else}\ c_2, k, \sigma) \rightsquigarrow (c_2, k, \sigma)}$$

$$\textsc{while done (computation)}\ \frac{\sigma \vdash b \Rightarrow \mathit{false}}{(\mathtt{while}\,(b)\ c, k, \sigma) \rightsquigarrow (\mathsf{skip}, k, \sigma)} \qquad \textsc{while loop (computation + focusing)}\ \frac{\sigma \vdash b \Rightarrow \mathit{true}}{(\mathtt{while}\,(b)\ c, k, \sigma) \rightsquigarrow (c, \circlearrowleft(b, c, k), \sigma)}$$

$$\textsc{skip sequence (resumption)}\ (\mathsf{skip}, c; k, \sigma) \rightsquigarrow (c, k, \sigma) \qquad \textsc{skip while (resumption)}\ (\mathsf{skip}, \circlearrowleft(b, c, k), \sigma) \rightsquigarrow (\mathtt{while}\,(b)\ c, k, \sigma)$$

Fig. 4: Continuation-based small-step semantics for IMP

the shape $(c, \mathsf{stop}, \sigma_i)$ and final states are of the shape $(\mathsf{skip}, \mathsf{stop}, \sigma_f)$. Given initial and final stores σ_i and σ_f, the termination of a command c is defined as $\mathsf{kterminates}(\sigma_i, c, \sigma_f) \triangleq (c, \mathsf{stop}, \sigma_i) \rightsquigarrow^* (\mathsf{skip}, \mathsf{stop}, \sigma_f)$. Given an initial store σ_i, the diverging execution of c is defined as $\mathsf{kdiverges}(\sigma_i, c) \triangleq (c, \mathsf{stop}, \sigma_i) \rightsquigarrow^\infty$.

4.2 Equivalence between the Two small-step Semantics

The equivalence between the two small-step semantics states that they agree on which commands terminate and which commands diverge. In other words, it amounts to the two following properties.

Theorem 3 (Equivalence of terminating behaviors).
$\forall c, \sigma_i, \sigma_f.\, \mathsf{terminates}(c, \sigma_i, \sigma_f) \leftrightarrow \mathsf{kterminates}(c, \sigma_i, \sigma_f)$.

Theorem 4 (Equivalence of diverging behaviors).
$\forall c, \sigma_i.\, \mathsf{diverges}(c, \sigma_i) \leftrightarrow \mathsf{kdiverges}(c, \sigma_i)$.

We use a simulation diagram to prove each theorem in a direction. More precisely, we only have to define the matching invariant $\approx$ between semantic states, the anti-stuttering measure between source states. Conducting these proofs is yet another opportunity to validate these semantics.

As an example, we show that every transition of the continuation semantics is simulated by zero, one or several reduction steps. Given a semantic state (c, k, σ) the measure is defined by a recursive function that counts the nesting of sequence operators constructs in c. The invariant $(c, k, \sigma) \approx (c', \sigma')$ is defined in Fig. 5.

$$\textsc{build stop}\quad (\mathsf{stop}, c) \hookrightarrow c \qquad \textsc{build seq}\quad \frac{(k_1, (c; c_1)) \hookrightarrow c'}{((c_1; k_1), c) \hookrightarrow c'} \qquad \textsc{build loop}\quad \frac{(k_1, (c; \texttt{while}\,(b)\ c)) \hookrightarrow c'}{(\circlearrowleft(b_1, c_1, k_1), c) \hookrightarrow c'}$$

$$\textsc{matching invariant}\quad \frac{(k, c) \hookrightarrow c'}{(c, k, \sigma) \approx (c', \sigma)}$$

Fig. 5: Equivalence between the two semantics: matching invariant

The command c' is computed from the command c following the $\hookrightarrow$ function, that takes the sub-command c and the continuation k, and rebuilds the whole command. This is achieved by inserting c to the left of the nested sequence constructors described by k. For instance, the second rule builds a sequence of commands from the left command of a sequence and the sequence continuations related to it. The proof of the simulation proceeds by structural induction on continuations.

5 Clight Semantics

Simulation-based proof techniques scale to realistic languages such as C and continuation-based semantics are the privileged style to facilitate compiler correctness proofs, as shown by their use in the CompCert compiler. There are two C-like languages in CompCert, CompCertC the source language of the compiler and Clight, that is a choice language to reason on C programs. This section introduces some background on CompCert generic semantics. Then, it defines the Clight semantics.

5.1 Form IMP to CompCert

In order to model the execution of programs written in realistic languages such as C, the semantic judgments introduced in Section 4.1 need to be extended in three directions. First, C programs are composed of two kinds of functions, depending whether they are defined in the program (internal) or not (external, that are declared with a name and a signature). So, to ensure some guarantees on external functions, the semantics observe traces of input/output operations performed during execution. These traces belong to program behaviors. Second, because of pointer arithmetic, variables need to be generalized to left values, and the store becomes a memory model storing different kinds of values, with different permissions to prevent memory overflows. Third, because of the presence of global, local and temporary variables and functions, semantic states are more involved. This section gives the background to understand these three extensions that are explained in more detail in [9, 24, 26].

Instrumenting the semantics to collect traces of observables. Traces of input/output operations (*e.g.*, memory accesses to global volatile variables used by hardware devices) are part of the observed behavior. The correctness theorem is strengthened to show preservation of these observable effects (that can not modify memory), and it becomes: if the source program terminates (resp. diverges) and performs observable effects t, then the generated program terminates (resp. diverges) and performs the same effects t, and has no other behavior. Semantic judgments $\mathcal{S} \to \mathcal{S}'$ become $\mathcal{S} \xrightarrow{t} \mathcal{S}'$, where the trace t is a list of (possibly infinite) events. An execution step $\mathcal{S} \xrightarrow{\varepsilon} \mathcal{S}'$ means that no event is triggered during this step.

Memory model. The memory model of CompCert is shared by all the languages of the compiler. It provides an abstract view of memory refined into a concrete memory layout. The memory is a collection of disjoint blocks identified by memory addresses, and with fixed lower and upper bounds. Blocks store values (*i.e.*, byte-sized quantities) that can be either machine integers (stored on 32 and 64 bits), pointers, floating-point numbers, or `undef`. A pointer (or a memory location) is a pair (ℓ, δ) made of a block identifier and an integer offset within that block. The special `undef` value is also used to denote arbitrary bit patterns, such as the value of uninitialized variables.

Basic memory operations are load, store, alloc, and free operations. Among the properties of memory operations are good variables properties, that ensure memory safety (*e.g.*, no out-of-bound array access) in terminating and diverging executions of programs. Moreover, memory operations are preserved by generic memory transformations called extensions and injections. They preserve the properties of memory operations. Last, in the C semantics of CompCert, each variable allocation creates a new block, and the number of blocks decreases during compilation.

Semantics states. Three environments are used in the semantic judgments for Clight, in addition to the memory store.

- A global environment G maps global variables to memory blocks, and function pointers to their definitions. It does not change during evaluation and execution.
- A local environment σ maps local variables to pairs made of a memory block and a type.
- A temporary environment σ_l maps local temporaries (namely a special class of local variables that do not reside in memory and whose address cannot be taken) to values.

Semantic states all carry a memory store M, mapping addresses to values, and a continuation k materializing the call stack. These states are of three kinds:

- regular states $\mathcal{S}(f, c, k, \sigma, \sigma_l, M)$, that are execution points within an internal function f at statement c,

Statements

c ::=	skip	empty statement
	$\mid a_1^{\tau_1} = a_2^{\tau_2}$	assignment to a left value
	$\mid id \leftarrow a^{\tau}$	assignment to a temporary variable
	$\mid (a_1^{\tau_1})^{?} = a_2^{\tau_2}((a^{\tau})^*)$	function call
	$\mid (a_1^{\tau_1})^{?} = \mathsf{ef}\ \tau_{ext}^*\ (a^{\tau})^*$	builtin invocation
	$\mid c_1; c_2$	sequence
	$\mid \mathtt{if}\,(a^{\tau})\ c_1\ \mathtt{else}\ c_2$	conditional
	$\mid \mathsf{switch}\,(a^{\tau})\ ls$	multi-way test and branch
	$\mid \mathsf{loop}\,(c_1)\ \ c_2$	infinite loop
	$\mid \mathsf{break}$	exit from the current loop
	$\mid \mathsf{continue}$	next iteration of the current loop
	$\mid \mathsf{return}\ a^{\tau}$	return from current function
	$\mid lbl : c$	labeled statement
	$\mid \mathtt{goto}\ lbl$	jump to a label

Switch cases:

$ls ::= \epsilon \mid (lbl^{?} : c) :: ls$

Fig. 6: Clight syntax

- call states $\mathcal{C}(\mathsf{Fd}, v^*, k, M)$, that are reached each time a function defined by Fd is called; the state carries the parameters passing v^* from the caller,
- return states $\mathcal{R}(v, k, M)$ from a caller to a callee, with resulting value v.

5.2 Clight Syntax

The syntax of Clight is defined in Fig. 6. Clight is a simplified version of the CompCertC source language of CompCert, where expressions are pure, and assignments and function calls are commands instead of expressions. Clight expressions are annotated with their types and written a^{τ}; expressions are not detailed in this paper as they are similar to those defined in [9]. A novelty in expressions is the bitfield access mode for members of struct or unions.

Base statements are skip, assignments, function calls (with optional assignment of the return value to a local variable) and builtin invocations, break, continue and function return. Other statements describe the control flow: sequences, conditionals, loops, switch and goto statements.

An infinite loop written $\mathsf{loop}\,(c_1)\ c_2$ executes c_1 then c_2 repeatedly. It is equivalent to the C loop written for (; ; c1) c2. A continue in c_1 branches to c_2. The three C loops are derived forms; a while loop $\mathtt{while}\,(e)\ c$ is defined as $\mathsf{loop}\,(\{\mathtt{if}\,(e)\,\mathsf{skip}\,\mathtt{else}\,\mathsf{break}\}; c)\,\mathsf{skip}$, and a for loop $\mathsf{for}\,(c_1; a_2; c_3)\ c_4$ is defined as the sequence $c_1; \mathsf{loop}\,(\mathtt{if}\,(a_2)\,\mathsf{skip}\,\mathtt{else}\,\mathsf{break}; c_3)\ c_4$. A switch statement consists of an expression and a list of cases. A case is a labeled statement $\lfloor lbl \rfloor : c$ or the default case $\epsilon : c$.

A program is composed of several definitions of functions, global variables and struct and union types. A function definition Fd is either $\mathsf{internal}(f)$ or $\mathsf{external}(\mathsf{ef}, \mathsf{targs}, \mathsf{tres}, \mathsf{cconv})$. The definition of an internal function f is composed

of a signature, local variables and a body (namely a statement, called $f.\mathsf{body}$). The definition of an external function ef only declares its signature.

The signature of a function f is composed of a return type called $f.\mathsf{return}$, the types of parameters and information cconv related to calling conventions (*e.g.*, the possibility to return struct for functions, or the use of old-style unprototyped functions). External functions model input/output operations; they include system calls and compiler built-in functions (*e.g.*, volatile reads and stores, memory allocation and deallocation, and copy of memory blocks). Function calls and built-in invocations are annotated with their signature.

5.3 Clight Semantics

The semantics of Clight is defined by the following semantic judgments. The terminating (resp. diverging) execution of a whole program is defined using the relation $\rightarrow^*$ (resp. $\rightarrow^\infty$), as in Section 3.

- The big-step evaluation $G, \sigma, \sigma_l, M \vdash a_1^{\tau_1} \Leftarrow (\ell, \delta), b$ of an expression $a_1^{\tau_1}$ in left-value position results in a memory location (ℓ, δ) that contains the value of $a_1^{\tau_1}$ and the bitfield designation b, that is the access mode for members of structs or unions (either a plain field or a bitfield).
- The big-step evaluation $G, \sigma, \sigma_l, M \vdash a^\tau \Rightarrow v$ of an expression a^τ computes its value v.
- The big-step evaluation $G, \sigma, \sigma_l, M \vdash (a^\tau)^* \Rightarrow v^*$ of a list of expressions computes a list of values.
- The small-step execution $G \vdash S \xrightarrow{t} S'$ from a semantic state S steps to state S' and emits trace t.

The semantic rules for statements are defined in Fig. 7, Fig. 8 and Fig. 9. The rules of Fig. 7 and Fig. 8 step within the currently-executing function and do not trigger any external event, hence the empty trace ε in the rules. Fig. 7 defines the continuations for these statements and the semantics of assignments, sequences of statements, loops, break and continue statements. The rule for if statements is not shown as it is similar to the rule of Fig. 4.

As in Fig. 4, a continuation k consists of the remainder of a command c and a control stack that describes the context in which k occurs. The stop and sequence (;) continuations are defined as in Fig. 4. Two continuations are defined for loops: $\circlearrowleft(c_1, c_2, k)$ means after c_1 in $\mathsf{loop}\,(c_1)\;\; c_2$, and $\circlearrowleft\circlearrowleft(c_1, c_2, k)$ means after c_2 in this loop. A continuation $\nearrow(k)$ is defined to catch in k a break statement arising out of a switch statement. To handle a call to a function f, we need a new form $\rightsquigarrow(x^?, f, \sigma, \sigma_l, k)$ of continuation representing pending function calls in k, given the local (resp. temporary) environment σ (resp. σ_l) of the calling function and the optional identifier x where the result is stored.

An assignment $a_1^{\tau_1} := a_2^{\tau_2}$ to a left-value $a_1^{\tau_1}$ evaluates $a_2^{\tau_2}$ to a memory location (ℓ, δ), and expression $a_1^{\tau_1}$ to value v_2, then casts v_2 into v in order to take into account the types of both expressions. The value v is stored at this memory location, which may fail. Last, the memory M' is returned after storing

Continuations:

$$
\begin{array}{lll}
k ::= \mathsf{stop} \mid c; k \mid \circlearrowright(c, c, k) \mid \circlearrowright\circlearrowright(c, c, k) & & \text{stop, sequence, loops} \\
\quad \mid \nearrow(k) \mid \rightsquigarrow(x^?, f, \sigma, \sigma_l, k) & & \text{switch, call}
\end{array}
$$

ASSIGN (COMPUTATION)

$$
\frac{G, \sigma, \sigma_l, M \vdash a_1^{\tau_1} \Leftarrow (\ell, \delta), b \quad G, \sigma, \sigma_l, M \vdash a_2^{\tau_2} \Rightarrow v_2 \quad \mathsf{semCast}(v_2, a_2^{\tau_2}, a_1^{\tau_1}, m) = \lfloor v \rfloor \quad G \vdash \tau_1, m, (\ell, \delta) : b, v, m'}{G \vdash \mathcal{S}(f, (a_1^{\tau_1} := a_2^{\tau_2}), k, \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{S}(f, \mathsf{skip}, k, \sigma, \sigma_l, M')}
$$

SET (COMPUTATION)

$$
\frac{G, \sigma, \sigma_l, M \vdash a^{\tau} \Rightarrow v}{G \vdash \mathcal{S}(f, (id \leftarrow a^{\tau}), k, \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{S}(f, \mathsf{skip}, k, \sigma, \sigma_l[id \to v], M)}
$$

SEQUENCE (FOCUSING)

$$
G \vdash \mathcal{S}(f, (c_1; c_2), k, \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{S}(f, c_1, c_2; k, \sigma, \sigma_l, M)
$$

SKIP SEQUENCE (RESUMPTION)

$$
G \vdash \mathcal{S}(f, \mathsf{skip}, c; k, \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{S}(f, c, k, \sigma, \sigma_l, M)
$$

CONTINUE SEQUENCE (RESUMPTION)

$$
G \vdash \mathcal{S}(f, \mathsf{continue}, c; k, \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{S}(f, \mathsf{continue}, k, \sigma, \sigma_l, M)
$$

BREAK SEQUENCE (RESUMPTION)

$$
G \vdash \mathcal{S}(f, \mathsf{break}, c; k, \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{S}(f, \mathsf{break}, k, \sigma, \sigma_l, M)
$$

LOOP (COMPUTATION + FOCUSING)

$$
G \vdash \mathcal{S}(f, (\mathsf{loop}\,(c_1)\ \ c_2), k, \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{S}(f, c_1, \circlearrowright(c_1, c_2, k), \sigma, \sigma_l, M)
$$

SKIP OR CONTINUE LOOP (RESUMPTION)

$$
\frac{x \in \{\mathsf{skip}; \mathsf{continue}\}}{G \vdash \mathcal{S}(f, x, \circlearrowright(c_1, c_2, k), \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{S}(f, c_2, \circlearrowright\circlearrowright(c_1, c_2, k), \sigma, \sigma_l, M)}
$$

BREAK LOOP1 (RESUMPTION)

$$
G \vdash \mathcal{S}(f, \mathsf{break}, \circlearrowright(c_1, c_2, k), \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{S}(f, \mathsf{skip}, k, \sigma, \sigma_l, M)
$$

BREAK LOOP2 (RESUMPTION)

$$
G \vdash \mathcal{S}(f, \mathsf{break}, \circlearrowright\circlearrowright(c_1, c_2, k), \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{S}(f, \mathsf{skip}, k, \sigma, \sigma_l, M)
$$

SKIP LOOP (RESUMPTION)

$$
G \vdash \mathcal{S}(f, \mathsf{skip}, \circlearrowright\circlearrowright(c_1, c_2, k), \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{S}(f, \mathsf{loop}\,(c_1)\ \ c_2, k, \sigma, \sigma_l, M)
$$

Fig. 7: Clight semantics for statements (first rules)

the value v in the datum of type τ stored at memory location (ℓ, δ), and the

LABEL (COMPUTATION)

$$G \vdash \mathcal{S}(f, (lbl : c), k, \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{S}(f, c, k, \sigma, \sigma_l, M)$$

GOTO (COMPUTATION + FOCUSING)

$$\frac{\mathsf{findLabel}(lbl, f.\mathsf{body}, \mathsf{callCont}(k)) = \lfloor (c', k') \rfloor}{G \vdash \mathcal{S}(f, (\mathtt{goto}\, lbl), k, \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{S}(f, c', k', \sigma, \sigma_l, M)}$$

SWITCH (COMPUTATION + FOCUSING)

$$\frac{G, \sigma, \sigma_l, M \vdash a^\tau \Rightarrow v \quad \mathsf{semSwitchArg}(v, \tau) = \lfloor lbl \rfloor}{G \vdash \mathcal{S}(f, (\mathsf{switch}\,(a^\tau)\ \mathsf{sl}), k, \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{S}(f, \mathsf{seq}(\mathsf{selectSwitch}(lbl) = \mathsf{sl}), \nearrow(k), \sigma, \sigma_l, M)}$$

SKIP BREAK SWITCH (RESUMPTION)

$$\frac{x \in \{\mathsf{skip}; \mathsf{break}\}}{G \vdash \mathcal{S}(f, x, \nearrow(k), \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{S}(f, \mathsf{skip}, k, \sigma, \sigma_l, M)}$$

CONTINUE SWITCH (RESUMPTION)

$$G \vdash \mathcal{S}(f, \mathsf{continue}, \nearrow(k), \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{S}(f, \mathsf{continue}, k, \sigma, \sigma_l, M)$$

Fig. 8: Clight semantics for goto and switch statements

statement is reduced to skip. An assignment $id \leftarrow a^\tau$ to a temporary variable id evaluates a^τ to a value v and updates the local environment accordingly.

The two rules for sequences are similar to the rules given in Fig. 4. The execution of a continue statement in a loop body interrupts the current execution of this loop body and triggers its next iteration. So, when a continue statement is after c_1 in a loop $\mathsf{loop}\,(c_1)\ \ c_2$, then c_2 is the next statement to execute and the continuation is updated accordingly.

The execution of a break statement in a loop body terminates the execution of the current loop body. So, the statements c_1 and c_2 of the loop body are popped from the continuation stack. Moreover, when a continue or a break statement is followed by a statement c, then c is not executed, hence it is popped from the continuation stack. The resumption rule for loops steps to the execution of the next execution of the loop body, when the continuation is a $\circlearrowright\circlearrowright$ continuation.

Fig. 8 defines the semantics of labeled, goto and switch statements. The execution of a labeled statement $lbl : c$ steps to the execution of c. The execution of a $\mathtt{goto}\, lbl$ statement in a function f first pops the continuation stack k until a call or a stop, in order to remove from k its local context part. Then, from this continuation $\mathsf{callCont}(k)$ representing the control flow from the last caller of f, findLabel computes recursively (if any) the control flow in f from its entry point until the statement labeled lbl. A new continuation k' that extends k and represents this control flow is then manufactured, and findLabel returns (if any) the pair (c', k'), where c' is the leftmost sub-statement of c labeled lbl. The rule thus steps to statement c' and continuation k', with no change in environments.

The execution of a statement $\mathsf{switch}\,(a^\tau)\ \mathsf{sl}$ first evaluates a^τ into value v, which is then casted into an unsigned integer when τ is an integer type (and fails otherwise). The rule steps to the appropriate case of the switch, given the value of the selector expression, and the corresponding statements are executed (after being converted into a sequence of statements from a labeled statement). In other words, the rules focus on a case switch and the continuation remembers this control flow. This rule is general enough to model executions of unstructured switch statements such as Duff's device [14].

The execution of a break statement in a switch case terminates the execution of this case. In other words, the execution of break (or a skip) statement in a switch case steps to skip and updates the continuation into k. The execution of a continue statement in a switch case updates the continuation into k as well, while keeping the continue statement as the current statement.

The semantic rules involving call and return states are defined in Fig. 9. First, the rule for a call to an internal function identified by $a_f^{\tau_f}$ evaluates $a_f^{\tau_f}$ into v and each argument a^τ of the function. The value v identifies the block where the function definition Fd is stored in the global environment G, and $\mathsf{funct}(G, v)$ returns this definition if any. The rule requires that the signature of the called function matches the signature τ_f annotating the call, namely $\tau_f \# \mathsf{sigOf}(\mathsf{Fd})$.

The rule for a builtin invocation also evaluates the list of its arguments. A builtin is an external function ef and the rule applies ef to arguments v^*: it mainly checks that the builtin is known, that ef cannot modify the memory state M, that v^* are integers or floats and that they agree in number and types with the function signature (see [24]).

The execution of a return statement frees in memory M all the blocks of the current environment σ, and steps to a return state with the retuned value in any (or undef otherwise), and updated continuation and memory state.

A step from a callstate with an internal function f steps to a regular state to further execute the statements $f.\mathsf{body}$ of f. The semantics for allocation of variables (hence the modified memory M') and binding of parameters is given by $\mathsf{functionEntry}(f, v^*, M, \sigma, \sigma_l, M')$. Two semantics are supported, one where parameters are local variables, reside in memory, and can have their address taken, and the other where parameters are temporary variables and do not reside in memory.

A step from a callstate with an external function ef steps directly to a return state (to further return to its caller) after generating the appropriate event in the trace t. Moreover, the rule applies ef to arguments v^*, to perform similar checks to those performed by the rule for builtin invocation. Last, a step from a return state either ends the program execution (when the call stack becomes empty) or reaches the regular state of the caller that carries a skip statement and the returned value v stored in the local environment.

FUNCTION CALL

$$\frac{G, \sigma, \sigma_l, M \vdash a_f^{\tau_f} \Rightarrow v \quad G, \sigma, \sigma_l, M \vdash (a^\tau)^* \Rightarrow v^* \quad \mathsf{funct}(G, v) = \lfloor \mathsf{Fd} \rfloor \quad \tau_f \# \mathsf{sigOf}(\mathsf{Fd})}{G \vdash \mathcal{S}(f, id^? = a_f^{\tau_f}((a^\tau)^*), k, \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{C}(\mathsf{Fd}, v^*, \rightsquigarrow(id^?, f, \sigma, \sigma_l, k), M)}$$

BUILTIN INVOCATION

$$\frac{G, \sigma, \sigma_l, M \vdash (a^\tau)^* \Rightarrow v^* \quad G \vdash \mathsf{ef}(v^*), M \xrightarrow{t} v, M'}{G \vdash \mathcal{S}(f, id^? = \mathsf{ef}\ \tau^*_{ext}\ (a^\tau)^*, k, \sigma, \sigma_l, M) \xrightarrow{t} \mathcal{S}(f, \mathsf{skip}, k, \sigma, \sigma_l\{id^? \leftarrow v\}, M)}$$

RETURN 1

$$\frac{\mathsf{semCast}(v, \tau, f.\mathsf{return}, m) = \lfloor v' \rfloor \quad \mathsf{freeAll}(M, \sigma) = \lfloor M' \rfloor}{G \vdash \mathcal{S}(f, \mathsf{return}\ \lfloor a^\tau \rfloor, k, \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{R}(v', \mathsf{callCont}(k), M')}$$

RETURN 0

$$\frac{\mathsf{freeAll}(M, \sigma) = \lfloor M' \rfloor}{G \vdash \mathcal{S}(f, \mathsf{return}\ \epsilon, k, \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{R}(\mathsf{undef}, \mathsf{callCont}(k), M')}$$

SKIP CALL

$$\frac{\mathsf{freeAll}(M, \sigma) = \lfloor M' \rfloor}{G \vdash \mathcal{S}(f, \mathsf{skip}, k, \sigma, \sigma_l, M) \xrightarrow{\varepsilon} \mathcal{R}(\mathsf{undef}, k, M')}$$

INTERNAL FUNCTION

$$\frac{\mathsf{functionEntry}(f, v^*, M, \sigma, \sigma_l, M')}{G \vdash \mathcal{C}(\mathsf{internal}(f), v^*, k, M) \xrightarrow{\varepsilon} \mathcal{S}(f, f.\mathsf{body}, k, \sigma, \sigma_l, M')}$$

EXTERNAL FUNCTION

$$\frac{G \vdash \mathsf{ef}(v^*), M \xrightarrow{t} v, M'}{G \vdash \mathcal{C}(\mathsf{external}(\mathsf{ef}, \mathsf{targs}, \mathsf{tres}, \mathsf{cconv}), v^*, k, M) \xrightarrow{t} \mathcal{R}(v, k, m')}$$

RETURNSTATE

$$G \vdash \mathcal{R}(v, \rightsquigarrow(id^?, f, \sigma, \sigma_l, k), M) \xrightarrow{\varepsilon} \mathcal{S}(f, \mathsf{skip}, k, \sigma, \sigma_l\{id^? \leftarrow v\}, M)$$

Fig. 9: Clight semantics for functions

6 Related Work

The semantics of the Clight language were first mechanized using big-step semantics [9] that were targeting a smaller language and only observing terminating behaviors. Then, a co-inductive interpretation of big-step semantics for diverging behaviors was defined [28]. However, this approach did not scale to conduct compiler correctness proofs of CompCert, contrary to the current continuation-based small-step semantics. Indeed, the cost for extending the correctness proof to diverging behaviors was relatively high (and Coq support for coinductive proofs is temperamental). Compared to [9], the Clight language was extended to model assignments of temporary variables, single infinite loops (instead of C lops), labeled and general goto statements and switch statements.

Other mechanized semantics were defined for realistic languages such as Java, the JVM [20] and JavaScript [10]. In [20], the authors define a big-step semantics and a small-step semantics, which are proved equivalent. A correctness proof of a two-stage compiler from Java to a virtual machine is proved correct using the simulation proof technique. These semantics target a simpler compiler than CompCert and only observe terminating behaviors and do not use continuations.

The idea of using continuations to facilitate some mechanized semantic reasoning first appeared in [4], where an axiomatic semantics (a.k.a. program logics) was defined from an operational semantics. The considered language was Cminor, a lower-level language than Clight, that is the target language of the CompCert front-end. Thanks to continuations, the soundness proof of the axiomatic semantics reuses the induction principles generated by Coq, thus avoiding to craft error-prone induction principles. Continuation-based small-step semantics were then used in the backend of the CompCert compiler [24].

7 Conclusion

This paper presented some operational styles for defining mechanized semantics of programming languages, starting from a toy imperative language to the C language. Exploration on toy languages is essential, but the results do not directly scale to big languages. This paper details the Clight semantics of CompCert, a reasonable proposal that works well in the context of compiler verification and a choice language to reason on C programs.

The continuation-based small-step semantics style detailed in this paper is the style chosen for all the languages of the CompCert compiler. It models terminating and diverging executions of programs and facilitates the semantic reasoning using simulation proof techniques.

Mechanized semantics is a need shared by many verification efforts, not just verified compilation. It is still a difficult task, especially for realistic programming languages. Better tooling for defining and maintaining mechanized semantics for realistic languages is needed.

References

1. Agda version 2.6.4 (2023), https://wiki.portal.chalmers.se/agda/Main/HomePage
2. Isabelle2023 (2023), https://isabelle.in.tum.de/
3. Online Coq development for CompCert version 3.13 (2023), https://compcert.org/doc/index.html
4. Appel, A.W., Blazy, S.: Separation logic for small-step cminor. In: Schneider, K., Brandt, J. (eds.) Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4732, pp. 5–21. Springer (2007), https://doi.org/10.1007/978-3-540-74591-4_3
5. Barrière, A., Blazy, S., Flückiger, O., Pichardie, D., Vitek, J.: Formally verified speculation and deoptimization in a JIT compiler. Proc. ACM Program. Lang. (POPL) (2021), https://doi.org/10.1145/3434327

6. Barrière, A., Blazy, S., Pichardie, D.: Formally verified native code generation in an effectful JIT: turning the CompCert backend into a formally verified JIT compiler. Proc. ACM Program. Lang. **7**(POPL), 249–277 (2023), https://doi.org/10.1145/3571202
7. Barthe, G., Blazy, S., Grégoire, B., Hutin, R., Laporte, V., Pichardie, D., Trieu, A.: Formal verification of a constant-time preserving C compiler. Proc. ACM Program. Lang. **4**(POPL), 7:1–7:30 (2020). https://doi.org/10.1145/3371075, https://doi.org/10.1145/3371075
8. Blazy, S., Hutin, R.: Formal verification of a program obfuscation based on mixed boolean-arithmetic expressions. In: Mahboubi, A., Myreen, M.O. (eds.) Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019. pp. 196–208. ACM (2019). https://doi.org/10.1145/3293880.3294103, https://doi.org/10.1145/3293880.3294103
9. Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. Journal of Automated Reasoning **43**(3), 263–288 (2009). https://doi.org/10.1007/s10817-009-9148-3, https://hal.inria.fr/inria-00352524
10. Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., Smith, G.: A trusted mechanised javascript specification. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 87–100. ACM (2014). https://doi.org/10.1145/2535838.2535876, https://doi.org/10.1145/2535838.2535876
11. Bourke, T., Brun, L., Dagand, P., Leroy, X., Pouzet, M., Rieg, L.: A formally verified compiler for lustre. In: Cohen, A., Vechev, M.T. (eds.) Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 586–601. ACM (2017). https://doi.org/10.1145/3062341.3062358, https://doi.org/10.1145/3062341.3062358
12. Boyer, R.S., Moore, J.S.: MJRTY: A fast majority vote algorithm. In: Boyer, R.S. (ed.) Automated Reasoning: Essays in Honor of Woody Bledsoe. pp. 105–118. Kluwer Academic Publishers (1991)
13. Cao, Q., Beringer, L., Gruetter, S., Dodds, J., Appel, A.W.: VST-Floyd: A separation logic tool to verify correctness of C programs. J. Autom. Reason. **61**(1-4), 367–422 (2018). https://doi.org/10.1007/S10817-018-9457-5, https://doi.org/10.1007/s10817-018-9457-5
14. Duff, T.: (1983), www.lysator.liu.se/c/duffs-device.html
15. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Proceedings of the 22nd European Symposium on Programming. LNCS, vol. 7792, pp. 125–128. Springer (Mar 2013). https://doi.org/10.1007/978-3-642-37036-6_8, https://doi.org/10.1007/978-3-642-37036-6_8
16. Herklotz, Y., Pollard, J.D., Ramanathan, N., Wickerson, J.: Formal verification of high-level synthesis. Proc. ACM Program. Lang. **5**(OOPSLA) (oct 2021). https://doi.org/10.1145/3485494, https://doi.org/10.1145/3485494
17. Inria: The Coq proof assistant reference manual (2022), http://coq.inria.fr, version 8.12.1
18. Jourdan, J., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified C static analyzer. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Princi-

ples of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 247–259. ACM (2015). https://doi.org/10.1145/2676726.2676966, https://doi.org/10.1145/2676726.2676966
19. Kästner, D., Barrho, J., Wünsche, U., Schlickling, M., Schommer, B., Schmidt, M., Ferdinand, C., Leroy, X., Blazy, S.: CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler. In: ERTS2 2018 - 9th European Congress Embedded Real-Time Software and Systems. pp. 1–9. 3AF, SEE, SIE (Jan 2018)
20. Klein, G., Nipkow, T.: A machine-checked model for a java-like language, virtual machine, and compiler. ACM Trans. Program. Lang. Syst. **28**(4), 619–695 (jul 2006). https://doi.org/10.1145/1146809.1146811, https://doi.org/10.1145/1146809.1146811
21. Koenig, J., Shao, Z.: CompCertO: Compiling certified open c components. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 1095–1109. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454097, https://doi.org/10.1145/3453483.3454097
22. Leino, R.M.: Program Proofs. The MIT Press (2023), https://mitpress.mit.edu/9780262546232/program-proofs/
23. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM (2009). https://doi.org/10.1145/1538788.1538814
24. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4), 363–446 (2009). https://doi.org/10.1007/s10817-009-9155-4
25. Leroy, X.: Coq development for the course "mechanized semantics" (2019), https://github.com/xavierleroy/cdf-mech-sem/tree/master
26. Leroy, X., Blazy, S.: Formal verification of a c-like memory model and its uses for verifying program transformations. J. Autom. Reason. **41**(1), 1–31 (2008). https://doi.org/10.1007/S10817-008-9099-0, https://doi.org/10.1007/s10817-008-9099-0
27. Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: CompCert - A Formally Verified Optimizing Compiler. In: ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress. SEE (Jan 2016)
28. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Information and Computation **207**(2), 284–304 (2009). https://doi.org/10.1016/j.ic.2007.12.004
29. Mac Carthy, J., Painter, J.: Correctness of a compiler for arithmetic expressions. Mathematical Aspects of Computer Science (1967)
30. Martínez, G., Ahman, D., Dumitrescu, V., Giannarakis, N., Hawblitzel, C., Hriţcu, C., Narasimhamurthy, M., Paraskevopoulou, Z., Pit-Claudel, C., Protzenko, J., Ramananandro, T., Rastogi, A., Swamy, N.: Meta-F*: Proof automation with SMT, tactics, and metaprograms. In: Caires, L. (ed.) Programming Languages and Systems. pp. 30–59. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-17184-1_2, https://doi.org/10.1007/978-3-030-17184-1_2
31. Milner, R.J., Weyrauch, R.: Proving compiler correctness in a mechanized logic. Machine Intelligence **7**, 51–73 (1972)
32. Pierce, Benjamin, e.a.: Software foundations - volume 1: logical foundations (2023), https://softwarefoundations.cis.upenn.edu/lf-current/Imp.html, version 6.6
33. Tassarotti, J., Tristan, J.B.: Verified density compilation for a probabilistic programming language. Proc. ACM Program. Lang. **7**(PLDI) (jun 2023). https://doi.org/10.1145/3591245, https://doi.org/10.1145/3591245