Mechanized semantics for the Clight subset of the C language

Sandrine Blazy, Xavier Leroy

Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Mechanized semantics for the Clight subset of the C language

Sandrine Blazy · **Xavier Leroy**

the date of receipt and acceptance should be inserted later

Abstract This article presents the formal semantics of a large subset of the C language called Clight. Clight includes pointer arithmetic, `struct` and `union` types, C loops and structured `switch` statements. Clight is the source language of the CompCert verified compiler. The formal semantics of Clight is a big-step operational semantics that observes both terminating and diverging executions and produces traces of input/output events. The formal semantics of Clight is mechanized using the Coq proof assistant. In addition to the semantics of Clight, this article describes its integration in the CompCert verified compiler and several ways by which the semantics was validated.

Keywords The C programming language · Operational semantics · Mechanized semantics · Formal proof · The Coq proof assistant

1 Introduction

Formal semantics of programming languages—that is, the mathematical specification of legal programs and their behaviors—play an important role in several areas of computer science. For advanced programmers and compiler writers, formal semantics provide a more precise alternative to the informal English descriptions that usually pass as language standards. In the context of formal methods such as static analysis, model checking and program proof, formal semantics are required to validate the abstract interpretations and program logics (e.g. axiomatic semantics) used to analyze and reason about programs. The verification of programming tools such as compilers, type-checkers, static analyzers and program verifiers is another area where formal semantics for the languages involved is a prerequisite. While

This work was supported by Agence Nationale de la Recherche, grant number ANR-05-SSIA-0019.

S. Blazy
ENSIIE, 1 square de la Résistance, 91025 Evry cedex, France
E-mail: Sandrine.Blazy@ensiie.fr

X. Leroy
INRIA Paris-Rocquencourt, B.P. 105, 78153 Le Chesnay, France
E-mail: Xavier.Leroy@inria.fr

formal semantics for realistic languages can be defined on paper using ordinary mathematics [31, 16, 7], machine assistance such as the use of proof assistants greatly facilitates their definition and uses.

For high-level programming languages such as Java and functional languages, there exists a sizeable body of mechanized formalizations and verifications of operational semantics, axiomatic semantics, and programming tools such as compilers and bytecode verifiers. Despite being more popular for writing systems software and embedded software, lower-level languages such as C have attracted less interest: several formal semantics for various subsets of C have been published, but only a few have been mechanized.

The present article reports on the definition of the formal semantics of a large subset of the C language called Clight. Clight features most of the types and operators of C, including pointer arithmetic, pointers to functions, and `struct` and `union` types, as well as all C control structures except `goto`. The semantics of Clight is mechanized using the Coq proof assistant [10, 4]. It is presented as a big-step operational semantics that observes both terminating and diverging executions and produces traces of input/output events. The Clight subset of C and its semantics are presented in sections 2 and 3, respectively.

The work presented in this paper is part of an ongoing project called CompCert that develops a realistic compiler for the C language and formally verifies that it preserves the semantics of the programs being compiled. A previous paper [6] reports on the development and proof of semantic preservation in Coq of the front-end of this compiler: a translator from Clight to Cminor, a low-level, imperative intermediate language. The formal verification of the back-end of this compiler, which generates moderately optimized PowerPC assembly code from Cminor is described in [28]. Section 4 describes the integration of the Clight language and its semantics within the CompCert compiler and its verification.

Formal semantics for realistic programming languages are large and complicated. This raises the question of validating these semantics: how can we make sure that they correctly capture the expected behaviors? In section 5, we argue that the correctness proof of the CompCert compiler provides an indirect but original way to validate the semantics of Clight, and discuss other approaches to the validation problem that we considered.

We finish this article by a discussion of related work in section 6, followed by future work and conclusions in section 7.

Availability The Coq development underlying this article can be consulted on-line at `http://compcert.inria.fr`.

Notations $[x,y[$ denotes the semi-open interval of integers $\{n \in \mathbb{Z} \mid x \leq n < y\}$. For functions returning "option" types, $\lfloor x \rfloor$ (read: "some x") corresponds to success with return value x, and $\emptyset$ (read: "none") corresponds to failure. In grammars, a^* denotes 0, 1 or several occurrences of syntactic category a, and $a^?$ denotes an optional occurrence of syntactic category a.

2 Abstract syntax of Clight

Clight is structured into expressions, statements and functions. In the Coq formalization, the abstract syntax is presented as inductive data types, therefore achieving a deep embedding of Clight into Coq.

Signedness: $\mathit{signedness} ::= \texttt{Signed} \mid \texttt{Unsigned}$

Integer sizes: $\mathit{intsize} ::= \texttt{I8} \mid \texttt{I16} \mid \texttt{I32}$

Float sizes: $\mathit{floatsize} ::= \texttt{F32} \mid \texttt{F64}$

Types:
$$\begin{aligned}\tau ::= {} & \texttt{int}(\mathit{intsize}, \mathit{signedness}) \\ \mid {} & \texttt{float}(\mathit{floatsize}) \\ \mid {} & \texttt{void} \\ \mid {} & \texttt{array}(\tau, n) \\ \mid {} & \texttt{pointer}(\tau) \\ \mid {} & \texttt{function}(\tau^*, \tau) \\ \mid {} & \texttt{struct}(id, \varphi) \\ \mid {} & \texttt{union}(id, \varphi) \\ \mid {} & \texttt{comp\_pointer}(id)\end{aligned}$$

Field lists: $\varphi ::= (id, \tau)^*$

Fig. 1 Abstract syntax of Clight types.

2.1 Types

The abstract syntax of Clight types is given in figure 1. Supported types include arithmetic types (integers and floats in various sizes and signedness), array types, pointer types (including pointers to functions), function types, as well as `struct` and `union` types. Named types are omitted: we assume that `typedef` definitions have been expanded away during parsing and type-checking.

The integral types fully specify the bit size of integers and floats, unlike the C types `int`, `long`, etc, whose sizes are left largely unspecified in the C standard. Typically, the parser maps `int` and `long` to size `I32`, `float` to size `F32`, and `double` to size `F64`. Currently, 64-bit integers and extended-precision floats are not supported.

Array types carry the number n of elements of the array, as a compile-time constant. Arrays with unknown sizes (τ`[]` in C) are replaced by pointer types in function parameter lists. Their only other use in C is within `extern` declarations of arrays, which are not supported in Clight.

Functions types specify the number and types of the function arguments and the type of the function result. Variadic functions and unprototyped functions (in the style of Ritchie's pre-standard C) are not supported.

In C, `struct` and `union` types are named and compared by name. This enables the definition of recursive `struct` types such as `struct s1 { int n; struct * s1 next;}`. Recursion within such types must go through a pointer type. For instance, the following is not allowed in C: `struct s2 { int n; struct s2 next;}`. To obviate the need to carry around a typing environment mapping `struct` and `union` names to their definitions, Clight `struct` and `union` types are structural: they carry a local identifier *id* and the list φ of their fields (names and types). Bit-fields are not supported. These types are compared by structure, like all other Clight types. In structural type systems, recursive types are traditionally represented with a fixpoint operator $\mu\alpha.\tau$, where α names the type $\mu\alpha.\tau$ within τ. We adapt this idea to Clight: within a `struct` or `union` type, the type $\texttt{comp\_pointer}(id)$ stands for a pointer type to the nearest enclosing `struct` or `union` type named *id*. For example, the structure `s1` defined previously in C is expressed by

$$\texttt{struct}(\texttt{s1}, (\texttt{n}, \texttt{int}(\texttt{I32}, \texttt{signed}))(\texttt{next}, \texttt{comp\_pointer}(\texttt{s1})))$$

Expressions:	$a ::= id$	variable identifier
	$\mid n$	integer constant
	$\mid f$	float constant
	$\mid \texttt{sizeof}(\tau)$	size of a type
	$\mid op_1\ a$	unary arithmetic operation
	$\mid a_1\ op_2\ a_2$	binary arithmetic operation
	$\mid *a$	pointer dereferencing
	$\mid a.id$	field access
	$\mid \&a$	taking the address of
	$\mid (\tau)a$	type cast
	$\mid a_1\ \texttt{?}\ a_2\ \texttt{:}\ a_3$	conditional expressions
Unary operators:	$op_1 ::= \texttt{-} \mid \texttt{~} \mid \texttt{!}$	
Binary operators:	$op_2 ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%}$	arithmetic operators
	$\mid \texttt{<<} \mid \texttt{>>} \mid \texttt{\&} \mid \texttt{\|} \mid \texttt{\^{}}$	bitwise operators
	$\mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{==} \mid \texttt{!=}$	relational operators

Fig. 2 Abstract syntax of Clight expressions

Incorrect structures such as `s2` above cannot be expressed at all, since `comp_pointer` let us refer to a pointer to an enclosing `struct` or `union`, but not to the `struct` or `union` directly.

Clight does not support any of the type qualifiers of C (`const`, `volatile`, `restrict`). These qualifiers are simply erased during parsing.

The following operations over types are defined: $\texttt{sizeof}(\tau)$ returns the storage size, in bytes, of type τ, and $\texttt{field\_offset}(id, \varphi)$ returns the byte offset of the field named id in a `struct` whose field list is φ, or $\emptyset$ if id does not appear in φ. The Coq development gives concrete definitions for these functions, compatible with the PowerPC ABI [48, chap. 3]. Typically, `struct` fields are laid out consecutively and padding is inserted so that each field is naturally aligned. Here are the only properties that a Clight producer or user needs to rely on:

- Sizes are positive: $\texttt{sizeof}(\tau) > 0$ for all types τ.
- Field offsets are within the range of allowed byte offsets for their enclosing `struct`: if $\texttt{field\_offset}(id, \varphi) = \lfloor \delta \rfloor$ and τ is the type associated with id in φ, then

$$[\delta, \delta + \texttt{sizeof}(\tau)[\subseteq [0, \texttt{sizeof}(\texttt{struct}\ id'\ \varphi)[.$$

- Different fields correspond to disjoint byte ranges: if $\texttt{field\_offset}(id_i, \varphi) = \lfloor \delta_i \rfloor$ and τ_i is the type associated with id_i in φ and $id_1 \neq id_2$, then

$$[\delta_1, \delta_1 + \texttt{sizeof}(\tau_1)[\cap [\delta_2, \delta_2 + \texttt{sizeof}(\tau_2)[= \emptyset.$$

- When a `struct` is a prefix of another `struct`, fields shared between the two `struct` have the same offsets: if $\texttt{field\_offset}(id, \varphi) = \lfloor \delta \rfloor$, then $\texttt{field\_offset}(id, \varphi.\varphi') = \lfloor \delta \rfloor$ for all additional fields φ'.

2.2 Expressions

The syntax of expressions is given in figure 2. All expressions and their sub-expressions are annotated by their static types. In the Coq formalization, expressions a are therefore pairs (b, τ) of a type τ and a term b of an inductive datatype determining the kind and

Statements:	$s ::=$	`skip`	empty statement
	$\mid$	$a_1 = a_2$	assignment
	$\mid$	$a_1 = a_2(a^*)$	function call
	$\mid$	$a(a^*)$	procedure call
	$\mid$	$s_1; s_2$	sequence
	$\mid$	`if`(a) s_1 `else` s_2	conditional
	$\mid$	`switch`(a) sw	multi-way branch
	$\mid$	`while`(a) s	"while" loop
	$\mid$	`do` s `while`(a)	"do" loop
	$\mid$	`for`(s_1, a_2, s_3) s	"for" loop
	$\mid$	`break`	exit from the current loop
	$\mid$	`continue`	next iteration of the current loop
	$\mid$	`return` $a^?$	return from current function
Switch cases:	$sw ::=$	`default` : s	default case
	$\mid$	`case` n : $s; sw$	labeled case

Fig. 3 Abstract syntax of Clight statements.

arguments of the expression. In this paper, we omit the type annotations over expressions, but write `type`(a) for the type annotating the expression a. The types carried by expressions are necessary to determine the semantics of type-dependent operators such as overloaded arithmetic operators. The following expressions can occur in left-value position: id, $*a$, and $a.id$.

Within expressions, only side-effect free operators of C are supported, but not assignment operators (`=`, `+=`, `++`, etc) nor function calls. In Clight, assignments and function calls are presented as statements and cannot occur within expressions. As a consequence, all Clight expressions always terminate and are pure: their evaluation performs no side effects. The first motivation for this design decision is to ensure determinism of evaluation. The C standard leaves evaluation order within expressions partially unspecified. If expressions can contain side-effects, different evaluation orders can lead to different results. As demonstrated by Norrish [36], capturing exactly the amount of nondeterminism permitted by the C standard complicates a formal semantics.

It is of course possible to commit on a particular evaluation order in a formal semantics for C. (Most C compiler choose a fixed evaluation order, typically right-to-left.) This is the approach we followed in an earlier version of this work [6]. Deterministic side-effects within expressions can be accommodated relatively easily with some styles of semantics (such as the big-step operational semantics of [6]), but complicate or even prevent other forms of semantics. In particular, it is much easier to define axiomatic semantics such as Hoare logic and separation logic if expressions are terminating and pure: in this case, syntactic expressions can safely be used as part of the logical assertions of the logic. Likewise, abstract interpretations and other forms of static analysis are much simplified if expressions are pure. Most static analysis and program verification tools for C actually start by pulling assignments and function calls out of expressions, and only then perform analyses over pure expressions [9, 13, 42, 8, 1, 17].

Variable declarations:	$dcl ::= (\tau\ id)^*$	name and type
Internal function definitions:	$F ::= \tau\ id(dcl_1)\,\{\,dcl_2; s\,\}$	(dcl_1 = parameters, dcl_2 = local variables)
External function declarations:	$Fe ::= \texttt{extern}\ \tau\ id(dcl)$	
Functions:	$Fd ::= F \mid Fe$	internal or external
Programs:	$P ::= dcl; Fd^*; \texttt{main} = id$	global variables, functions, entry point

Fig. 4 Abstract syntax of Clight functions and programs.

Some forms of C expressions are omitted in the abstract syntax but can be expressed as syntactic sugar:

$$\begin{array}{lrcl}
\text{array access:} & a_1[a_2] & \equiv & *(a_1 + a_2) \\
\text{indirect field access:} & a\texttt{->}id & \equiv & *(a.id) \\
\text{sequential ``and'':} & a_1\ \texttt{\&\&}\ a_2 & \equiv & a_1\ \texttt{?}\ (a_2\ \texttt{?}\ 1 : 0) : 0 \\
\text{sequential ``or'':} & a_1\ \texttt{||}\ a_2 & \equiv & a_1\ \texttt{?}\ 1 : (a_2\ \texttt{?}\ 1 : 0)
\end{array}$$

2.3 Statements

Figure 3 defines the syntax of Clight statements. All structured control statements of C (conditional, loops, Java-style `switch`, `break`, `continue` and `return`) are supported, but not unstructured statements such as `goto` and unstructured `switch` like the infamous "Duff's device" [12]. As previously mentioned, assignment $a_1 = a_2$ of an r-value a_2 to an l-value a_1, as well as function calls, are treated as statements. For function calls, the result can either be assigned to an l-value or discarded.

Blocks are omitted because block-scoped variables are not supported in Clight: variables are declared either with global scope at the level of programs, or with function scope at the beginning of functions.

The `for` loop is written $\texttt{for}(s_1, a_2, s_3)\ s$, where s_1 is executed once at the beginning of the loop, a_2 is the loop condition, s_3 is executed at the end of each iteration, and s is the loop body. In C, s_1 and s_3 are expressions, which are evaluated for their side effects. In Clight, since expressions are pure, we use statements instead. (However, the semantics requires that these statements terminate normally, but not by e.g. `break`.)

A `switch` statement consists in an expression and a list of cases. A case is a statement labeled by an integer constant (`case` n) or by the keyword `default`. Contrary to C, the default case is mandatory in a Clight `switch` statement and must occur last.

2.4 Functions and programs

A Clight program is composed of a list of declarations for global variables (name and type), a list of functions (see figure 4) and an identifier naming the entry point of the program (the `main` function in C). The Coq formalization supports a rudimentary form of initialization for global variables, where an initializer is a sequence of integer or floating-point constants; we omit this feature in this article.

Functions come in two flavors: internal or external. An internal function, written $\tau\ id(dcl_1)\,\{\,dcl_2; s\,\}$, is defined within the language. τ is the return type, id the name of

the function, dcl_1 its parameters (names and types), dcl_2 its local variables, and s its body. External functions `extern` τ $id(dcl)$ are merely declared, but not implemented. They are intended to model "system calls", whose result is provided by the operating system instead of being computed by a piece of Clight code.

3 Formal semantics for Clight

We now formalize the dynamic semantics of Clight, using natural semantics, also known as big-step operational semantics. The natural semantics observe the final result of program execution (divergence or termination), as well as a trace of the invocations of external functions performed by the program. The latter represents the input/output behavior of the program. Owing to the restriction that expressions are pure (section 2.2), the dynamic semantics is deterministic.

The static semantics of Clight (that is, its typing rules) has not been formally specified yet. The dynamic semantics is defined without assuming that the program is well-typed, and in particular without assuming that the type annotations over expressions are consistent. If they are inconsistent, the dynamic semantics can be undefined (the program goes wrong), or be defined but differ from what the C standard prescribes.

3.1 Evaluation judgements

The semantics is defined by the 10 judgements (predicates) listed below. They use semantic quantities such as values, environments, etc, that are summarized in figure 5 and explained later.

$G, E \vdash a, M \Leftarrow \ell$	(evaluation of expressions in l-value position)
$G, E \vdash a, M \Rightarrow v$	(evaluation of expressions in r-value position)
$G, E \vdash a^*, M \Rightarrow v^*$	(evaluation of lists of expressions)
$G, E \vdash s, M \stackrel{t}{\Rightarrow} out, M'$	(execution of statements, terminating case)
$G, E \vdash sw, M \stackrel{t}{\Rightarrow} out, M'$	(execution of the cases of a `switch`, terminating case)
$G \vdash Fd(v^*), M \stackrel{t}{\Rightarrow} v, M'$	(evaluation of function invocations, terminating case)
$G, E \vdash s, M \stackrel{T}{\Rightarrow} \infty$	(execution of statements, diverging case)
$G, E \vdash sw, M \stackrel{T}{\Rightarrow} \infty$	(execution of the cases of a `switch`, diverging case)
$G \vdash Fd(v^*), M \stackrel{T}{\Rightarrow} \infty$	(evaluation of function invocations, diverging case)
$\vdash P \Rightarrow B$	(execution of whole programs)

Each judgement relates a syntactic element to the result of executing this syntactic element. For an expression in l-value position, the result is a location ℓ: a pair of a block identifier b and a byte offset δ within this block. For an expression in r-value position and for a function application, the result is a value v: the discriminated union of 32-bit integers, 64-bit floating-point numbers, locations (representing the value of pointers), and the special value `undef` representing the contents of uninitialized memory. Clight does not support assignment between `struct` or `union`, nor passing a `struct` or `union` by value to a function; therefore, `struct` and `union` values need not be represented.

Following Norrish [36] and Huisman and Jacobs [21], the result associated with the execution of a statement s is an *outcome* out indicating how the execution terminated: either normally by running to completion or prematurely via a `break`, `continue` or `return` statement.

Block references:	$b \in \mathbb{Z}$	
Memory locations:	$\ell ::= (b, \delta)$	byte offset δ (a 32-bit integer) within block b
Values:	$v ::= \texttt{int}(n)$	integer value (n is a 32-bit integer)
	$\mid \texttt{float}(f)$	floating-point value (f is a 64-bit float)
	$\mid \texttt{ptr}(\ell)$	pointer value
	$\mid \texttt{undef}$	undefined value
Statement outcomes:	$out ::= \texttt{Normal}$	continue with next statement
	$\mid \texttt{Continue}$	go to the next iteration of the current loop
	$\mid \texttt{Break}$	exit from the current loop
	$\mid \texttt{Return}$	function exit
	$\mid \texttt{Return}(v)$	function exit, returning the value v
Global environments:	$G ::= (id \mapsto b)$	map from global variables to block references
	$\times (b \mapsto Fd)$	and map from function references to function definitions
Local environments:	$E ::= id \mapsto b$	map from local variables to block references
Memory states:	$M ::= b \mapsto (lo, hi, \delta \mapsto v)$	map from block references to bounds and contents
Memory quantities:	$\kappa ::= \texttt{int8signed} \mid \texttt{int8unsigned}$	
	$\mid \texttt{int16signed} \mid \texttt{int16unsigned}$	
	$\mid \texttt{int32} \mid \texttt{float32} \mid \texttt{float64}$	
I/O values:	$v_v ::= \texttt{int}(n) \mid \texttt{float}(f)$	
I/O events:	$\nu ::= id(v_v{}^* \mapsto v_v)$	name of external function, argument values, result value
Traces:	$t ::= \varepsilon \mid \nu.t$	finite traces (inductive)
	$T ::= \varepsilon \mid \nu.T$	finite or infinite traces (coinductive)
Program behaviors:	$B ::= \texttt{terminates}(t, n)$	termination with trace t and exit code n
	$\mid \texttt{diverges}(T)$	divergence with trace T

Operations over memory states:

$\texttt{alloc}(M, lo, hi) = (M', b)$	Allocate a fresh block of bounds $[lo, hi[$.
$\texttt{free}(M, b) = M'$	Free (invalidate) the block b.
$\texttt{load}(\kappa, M, b, n) = \lfloor v \rfloor$	Read one or several consecutive bytes (as determined by κ) at block b, offset n in memory state M. If successful return the contents of these bytes as value v.
$\texttt{store}(\kappa, M, b, n, v) = \lfloor M' \rfloor$	Store the value v into one or several consecutive bytes (as determined by κ) at offset n in block b of memory state M. If successful, return an updated memory state M'.

Operations over global environments:

$\texttt{funct}(G, b) = \lfloor b \rfloor$	Return the function definition Fd corresponding to the block b, if any.
$\texttt{symbol}(G, id) = \lfloor b \rfloor$	Return the block b corresponding to the global variable or function name id.
$\texttt{globalenv}(P) = G$	Construct the global environment G associated with the program P.
$\texttt{initmem}(P) = M$	Construct the initial memory state M for executing the program P.

Fig. 5 Semantic elements: values, environments, memory states, statement outcomes, etc

Most judgements are parameterized by a global environment G, a local environment E, and an initial memory state M. Local environments map function-scoped variables to references of memory blocks containing the values of these variables. (This indirection through memory is needed to allow the `&` operator to take the address of a variable.) These blocks are allocated at function entry and freed at function return (see rule 32 in figure 10). Likewise, the global environment G associates block references to program-global variables and functions. It also records the definitions of functions.

The memory model used in our semantics is detailed in [29]. Memory states M are modeled as a collection of blocks separated by construction and identified by integers b. Each block has lower and upper bounds lo, hi, fixed at allocation time, and associates values

Expressions in l-value position:

$$\frac{E(id) = b \text{ or } (id \notin \mathrm{Dom}(E) \text{ and } \texttt{symbol}(G,id) = \lfloor b \rfloor)}{G,E \vdash id, M \Leftarrow (b,0)}\ (1) \qquad \frac{G,E \vdash a, M \Rightarrow \texttt{ptr}(\ell)}{G,E \vdash *a, M \Leftarrow \ell}\ (2)$$

$$\frac{G,E \vdash a, M \Leftarrow (b,\delta) \quad \texttt{type}(a) = \texttt{struct}(id',\varphi) \quad \texttt{field\_offset}(id,\varphi) = \lfloor \delta' \rfloor}{G,E \vdash a.id, M \Leftarrow (b,\delta+\delta')}\ (3)$$

$$\frac{G,E \vdash a, M \Leftarrow \ell \quad \texttt{type}(a) = \texttt{union}(id',\varphi)}{G,E \vdash a.id, M \Leftarrow \ell}\ (4)$$

Expressions in r-value position:

$$G,E \vdash n, M \Rightarrow \texttt{int}(n)\ (5) \qquad G,E \vdash f, M \Rightarrow \texttt{float}(f)\ (6)$$

$$G,E \vdash \texttt{sizeof}(\tau), M \Rightarrow \texttt{int}(\texttt{sizeof}(\tau))\ (7) \qquad \frac{G,E \vdash a, M \Leftarrow \ell \quad \texttt{loadval}(\texttt{type}(a), M', \ell) = \lfloor v \rfloor}{G,E \vdash a, M \Rightarrow v}\ (8)$$

$$\frac{G,E \vdash a, M \Leftarrow \ell}{G,E \vdash \&a, M \Rightarrow \texttt{ptr}(\ell)}\ (9) \qquad \frac{G,E \vdash a_1, M \Rightarrow v_1 \quad \texttt{eval\_unop}(op_1, v_1, \texttt{type}(a_1)) = \lfloor v \rfloor}{G,E \vdash op_1\ a_1, M \Rightarrow v}\ (10)$$

$$\frac{G,E \vdash a_1, M \Rightarrow v_1 \quad G,E \vdash a_2, M_1 \Rightarrow v_2 \quad \texttt{eval\_binop}(op_2, v_1, \texttt{type}(a_1), v_2, \texttt{type}(a_2)) = \lfloor v \rfloor}{G,E \vdash a_1\ op_2\ a_2, M \Rightarrow v}\ (11)$$

$$\frac{G,E \vdash a_1, M \Rightarrow v_1 \quad \texttt{is\_true}(v_1, \texttt{type}(a_1)) \quad G,E \vdash a_2, M \Rightarrow v_2}{G,E \vdash a_1\ \texttt{?}\ a_2\ \texttt{:}\ a_3, M \Rightarrow v_2}\ (12)$$

$$\frac{G,E \vdash a_1, M \Rightarrow v_1 \quad \texttt{is\_false}(v_1, \texttt{type}(a_1)) \quad G,E \vdash a_3, M \Rightarrow v_3}{G,E \vdash a_1\ \texttt{?}\ a_2\ \texttt{:}\ a_3, M \Rightarrow v_3}\ (13)$$

$$\frac{G,E \vdash a, M \Rightarrow v_1 \quad \texttt{cast}(v_1, \texttt{type}(a), \tau) = \lfloor v \rfloor}{G,E \vdash (\tau)a, M \Rightarrow v}\ (14)$$

Fig. 6 Natural semantics for Clight expressions

to byte offsets $\delta \in [lo, hi[$. The basic operations over memory states are `alloc`, `free`, `load` and `store`, as summarized in figure 5.

Since Clight expressions are pure, the memory state is not modified during expression evaluation. It is modified, however, during the execution of statements and function calls. The corresponding judgements therefore return an updated memory state M'. They also produce a trace t of the external functions (system calls) invoked during execution. Each such invocation is described by an input/output event v recording the name of the external function invoked, the arguments provided by the program, and the result value provided by the operating system.

In addition to terminating behaviors, the semantics also characterizes divergence during the execution of a statement or of a function call. The treatment of divergence follows the coinductive natural approach of Leroy and Grall [30]. The result of a diverging execution is the trace T (possibly infinite) of input/output events performed.

In the Coq specification, the judgements of the dynamic semantics are encoded as mutually inductive predicates (for terminating executions) and mutually coinductive predicates (for diverging executions). Each defining case of each predicate corresponds exactly to an inference rule in the conventional, on-paper presentation of natural semantics. We show most of the inference rules in figures 6 to 12, and explain them in the remainder of this section.

Access modes: $\mu ::= \texttt{By\_value}(\kappa)$ access by value
$\mid \texttt{By\_reference}$ access by reference
$\mid \texttt{By\_nothing}$ no access

Associating access modes to Clight types:

$$\begin{aligned}
\mathscr{A}(\texttt{int}(\texttt{I8},\texttt{Signed})) &= \texttt{By_value}(\texttt{int8signed}) & \mathscr{A}(\texttt{array}(_,_)) &= \texttt{By_reference}\\
\mathscr{A}(\texttt{int}(\texttt{I8},\texttt{Unsigned})) &= \texttt{By_value}(\texttt{int8unsigned}) & \mathscr{A}(\texttt{function}(_,_)) &= \texttt{By_reference}\\
\mathscr{A}(\texttt{int}(\texttt{I16},\texttt{Signed})) &= \texttt{By_value}(\texttt{int16signed}) & \mathscr{A}(\texttt{struct})(_,_)) &= \texttt{By_nothing}\\
\mathscr{A}(\texttt{int}(\texttt{I16},\texttt{Unsigned})) &= \texttt{By_value}(\texttt{int16unsigned}) & \mathscr{A}(\texttt{union})(_,_)) &= \texttt{By_nothing}\\
\mathscr{A}(\texttt{int}(\texttt{I32},_)) &= \texttt{By_value}(\texttt{int32}) & \mathscr{A}(\texttt{void}) &= \texttt{By_nothing}\\
\mathscr{A}(\texttt{pointer}(_)) &= \texttt{By_value}(\texttt{int32})
\end{aligned}$$

Accessing or updating a value of type τ at location (b,δ) in memory state M:

$$\begin{aligned}
\texttt{loadval}(\tau,M,(b,\delta)) &= \texttt{load}(\kappa,M,b,\delta) && \text{if } \mathscr{A}(\tau) = \texttt{By_value}(\kappa)\\
\texttt{loadval}(\tau,M,(b,\delta)) &= \lfloor (b,\delta) \rfloor && \text{if } \mathscr{A}(\tau) = \texttt{By_reference}\\
\texttt{loadval}(\tau,M,(b,\delta)) &= \emptyset && \text{if } \mathscr{A}(\tau) = \texttt{By_nothing}\\
\texttt{storeval}(\tau,M,(b,\delta),v) &= \texttt{store}(\kappa,M,b,\delta,v) && \text{if } \mathscr{A}(\tau) = \texttt{By_value}(\kappa)\\
\texttt{storeval}(\tau,M,(b,\delta),v) &= \emptyset && \text{otherwise}
\end{aligned}$$

Fig. 7 Memory accesses.

3.2 Evaluation of expressions

Expressions in l-value position The first four rules of figure 6 illustrate the evaluation of an expression in l-value position. A variable *id* evaluates to the location $(b,0)$, where b is the block associated with *id* in the local environment E or the global environment G (rule 1). If an expression a evaluates (as an r-value) to a pointer value $\texttt{ptr}(\ell)$, then the location of the dereferencing expression $*a$ is ℓ (rule 2).

For field accesses $a.id$, the location $\ell = (b,\delta)$ of a is computed. If a has `union` type, this location is returned unchanged. (All fields of a `union` share the same position.) If a has `struct` type, the offset of field *id* is computed using the `field_offset` function, then added to δ.

From memory locations to values The evaluation of an l-value expression a in r-value position depends on the type of a (rule 8). If a has scalar type, its value is loaded from memory at the location of a. If a has array type, its value is equal to its location. Finally, some types cannot be used in r-value position: this includes `void` in C and `struct` and `union` types in Clight (because of the restriction that structs and unions cannot be passed by value). To capture these three cases, figure 7 defines the function $\mathscr{A}$ that maps Clight types to *access modes*, which can be one of: "by value", with a memory quantity κ (an access loads a quantity κ from the address of the l-value); "by reference" (an access simply returns the address of the l-value); or "by nothing" (no access is allowed). The `loadval` and `storeval` functions, also defined in figure 7, exploit address modes to implement the correct semantics for conversion of l-value to r-value (`loadval`) and assignment to an l-value (`storeval`).

Expressions in r-value position Rules 5 to 14 of figure 6 illustrate the evaluation of an expression in r-value position. Rule 8 evaluates an l-value expression in an r-value context. The expression is evaluated to its location ℓ. From this location, a value is deduced using

$$G,E \vdash \texttt{skip},M \stackrel{\varepsilon}{\Rightarrow} \texttt{Normal},M \quad (15) \qquad G,E \vdash \texttt{break},M \stackrel{\varepsilon}{\Rightarrow} \texttt{Break},M \quad (16)$$

$$G,E \vdash \texttt{continue},M \stackrel{\varepsilon}{\Rightarrow} \texttt{Continue},M \quad (17) \qquad G,E \vdash (\texttt{return}\ \emptyset),M \stackrel{\varepsilon}{\Rightarrow} \texttt{Return},M \quad (18)$$

$$\frac{G,E \vdash a,M \Rightarrow v}{G,E \vdash (\texttt{return}\ \lfloor a \rfloor),M \stackrel{\varepsilon}{\Rightarrow} \texttt{Return}(v),M} \quad (19)$$

$$\frac{G,E \vdash a_1,M \Leftarrow \ell \qquad G,E \vdash a_2,M \Rightarrow v \qquad \texttt{storeval}(\texttt{type}(a_1),M,\ell,v) = \lfloor M' \rfloor}{G,E \vdash (a_1\ \texttt{=}\ a_2),M \stackrel{\varepsilon}{\Rightarrow} \texttt{Normal},M'} \quad (20)$$

$$\frac{G,E \vdash s_1,M \stackrel{t_1}{\Rightarrow} \texttt{Normal},M_1 \qquad G,E \vdash s_2,M_1 \stackrel{t_2}{\Rightarrow} out,M_2}{G,E \vdash (s_1;s_2),M \stackrel{t_1 \cdot t_2}{\Rightarrow} out,M_2} \quad (21)$$

$$\frac{G,E \vdash s_1,M \stackrel{t}{\Rightarrow} out,M' \qquad out \neq \texttt{Normal}}{G,E \vdash (s_1;s_2),M \stackrel{t}{\Rightarrow} out,M'} \quad (22)$$

Fig. 8 Natural semantics for Clight statements (other than loops and `switch` statements)

the `loadval` function described above. By rule 9, `&`a evaluates to the pointer value $\texttt{ptr}(\ell)$ as soon as the l-value a evaluates to the location ℓ.

Rules 10 and 11 describe the evaluation of unary and binary operations. Taking binary operations as an example, the two argument expressions are evaluated and their values v_1, v_2 are combined using the the `eval_binop` function, which takes as additional arguments the types τ_1 and τ_2 of the arguments, in order to resolve overloaded and type-dependent operators. To give the general flavor of `eval_binop`, here are the cases corresponding to binary addition:

τ_1	τ_2	v_1	v_2	$\texttt{eval\_binop}(+, v_1, \tau_1, v_2, \tau_2)$
$\texttt{int}(\_)$	$\texttt{int}(\_)$	$\texttt{int}(n_1)$	$\texttt{int}(n_2)$	$\lfloor \texttt{int}(n_1 + n_2) \rfloor$
$\texttt{float}(\_)$	$\texttt{float}(\_)$	$\texttt{float}(f_1)$	$\texttt{float}(f_2)$	$\lfloor \texttt{float}(f_1 + f_2) \rfloor$
$\texttt{ptr}(\tau)$	$\texttt{int}(\_)$	$\texttt{ptr}(b,\delta)$	$\texttt{int}(n)$	$\lfloor \texttt{ptr}(b, \delta + n \times \texttt{sizeof}(\tau)) \rfloor$
$\texttt{int}(\_)$	$\texttt{ptr}(\tau)$	$\texttt{int}(n)$	$\texttt{ptr}(b,\delta)$	$\lfloor \texttt{ptr}(b, \delta + n \times \texttt{sizeof}(\tau)) \rfloor$
otherwise				$\emptyset$

The definition above rejects mixed arithmetic such as "int + float" because the parser that generates Clight abstract syntax (described in section 4.1) never produces this: it inserts explicit casts from integers to floats in this case. However, it would be easy to add cases dealing with mixed arithmetic. Likewise, the definition above adds two single precision floats using double-precision addition, in violation of the ISO C standard. Again, it would be easy to recognize this case and perform a single-precision addition.

Rules 12 and 13 define the evaluation of conditional expressions a_1 `?` a_2 `:` a_3. The predicates `is_true` and `is_false` determine the truth value of the value of a_1, depending on its type. At a `float` type, $\texttt{float}(0.0)$ is false and any other `float` value is true. At an `int` or `ptr` type, $\texttt{int}(0)$ is false and $\texttt{int}(n)$ ($n \neq 0$) and $\texttt{ptr}(\ell)$ values are true. (The null pointer is represented as $\texttt{int}(0)$.) All other combinations of values and types are neither true nor false, causing the semantics to go wrong.

Rule 14 evaluates a cast expression $(\tau)a$. The expression a is evaluated, and its value is converted from its natural type $\texttt{type}(a)$ to the expected type τ using the partial function `cast`. This function performs appropriate conversions, truncations and sign-extensions between integers and floats. We take a lax interpretation of casts involving pointer types: if the

Outcome updates (at the end of a loop execution):

$$\texttt{Break} \overset{loop}{\rightsquigarrow} \texttt{Normal} \qquad \texttt{Return} \overset{loop}{\rightsquigarrow} \texttt{Return} \qquad \texttt{Return}(v) \overset{loop}{\rightsquigarrow} \texttt{Return}(v)$$

`while` loops:

$$\frac{G,E \vdash a,M \Rightarrow v \quad \texttt{is\_false}(v,\texttt{type}(a))}{G,E \vdash (\texttt{while}(a)\ s),M \overset{\varepsilon}{\Rightarrow} \texttt{Normal},M} \quad (23)$$

$$\frac{G,E \vdash a,M \Rightarrow v \quad \texttt{is\_true}(v,\texttt{type}(a)) \quad G,E \vdash s,M \overset{t}{\Rightarrow} out,M' \quad out \overset{loop}{\rightsquigarrow} out'}{G,E \vdash (\texttt{while}(a)\ s),M \overset{t}{\Rightarrow} out',M'} \quad (24)$$

$$\frac{G,E \vdash a,M \Rightarrow v \quad \texttt{is\_true}(v,\texttt{type}(a)) \quad G,E \vdash s,M \overset{t_1}{\Rightarrow} (\texttt{Normal} \mid \texttt{Continue}),M_1 \quad G,E \vdash (\texttt{while}(a)\ s),M_1 \overset{t_2}{\Rightarrow} out',M_2}{G,E \vdash (\texttt{while}(a)\ s),M \overset{t_1 \cdot t_2}{\Rightarrow} out',M_2} \quad (25)$$

`for` loops:

$$\frac{s_1 \neq \texttt{skip} \quad G,E \vdash s_1,M \overset{t_1}{\Rightarrow} \texttt{Normal},M_1 \quad G,E \vdash (\texttt{for}(\texttt{skip},a_2,s_3)\ s),M_1 \overset{t_2}{\Rightarrow} out,M_2}{G,E \vdash (\texttt{for}(s_1,a_2,s_3)\ s),M \overset{t_1 \cdot t_2}{\Rightarrow} out,M_2} \quad (26)$$

$$\frac{G,E \vdash a_2,M \Rightarrow v \quad \texttt{is\_false}(v,\texttt{type}(a_2))}{G,E \vdash (\texttt{for}(\texttt{skip},a_2,s_3)\ s),M \overset{\varepsilon}{\Rightarrow} \texttt{Normal},M} \quad (27)$$

$$\frac{G,E \vdash a_2,M \Rightarrow v \quad \texttt{is\_true}(v,\texttt{type}(a_2)) \quad G,E \vdash s,M \overset{t_1}{\Rightarrow} out_1,M_1 \quad out_1 \overset{loop}{\rightsquigarrow} out}{G,E \vdash (\texttt{for}(\texttt{skip},a_2,s_3)\ s),M \overset{t}{\Rightarrow} out,M_1} \quad (28)$$

$$\frac{G,E \vdash a_2,M \Rightarrow v \quad \texttt{is\_true}(v,\texttt{type}(a_2)) \quad G,E \vdash s,M \overset{t_1}{\Rightarrow} (\texttt{Normal} \mid \texttt{Continue}),M_1 \quad G,E \vdash s_3,M_1 \overset{t_2}{\Rightarrow} \texttt{Normal},M_2 \quad G,E \vdash (\texttt{for}(\texttt{skip},a_2,s_3)\ s),M_2 \overset{t_3}{\Rightarrow} out,M_3}{G,E \vdash (\texttt{for}(\texttt{skip},a_2,s_3)\ ),M \overset{t_1 \cdot t_2 \cdot t_3}{\Rightarrow} out,M_3} \quad (29)$$

Fig. 9 Natural semantics for Clight loops

source and destination types are both either pointer types or 32-bit `int` types, any pointer or integer value can be converted between these types without change of representation. However, the `cast` function fails when converting between pointer types and `float` or small integer types, for example.

3.3 Statements and function invocations, terminating case

The rules in figure 8 define the execution of a statement that is neither a loop nor a `switch` statement. The execution of a `skip` statement yields the `Normal` outcome and the empty trace (rule 15). Similarly, the execution of a `break` (resp. `continue`) statement yields the `Break` (resp. `Continue`) outcome and the empty trace (rules 16 and 17). Rules 18–19 describe the execution of a `return` statement. The execution of a `return` statement evaluates the argument of the `return`, if any, and yields a `Return` outcome and the empty trace.

Rule 20 executes an assignment statement. An assignment statement a_1 `=` a_2 evaluates the l-value a_1 to a location ℓ and the r-value a_2 to a value v, then stores v at ℓ using the `storeval` function of figure 7, producing the final memory state M'. We assume that the types of a_1 and a_2 are identical, therefore no implicit cast is performed during assignment, unlike in C. (The Clight parser described in section 4.1 inserts an explicit cast on the r-value

Function calls:

$$\frac{\begin{array}{c} G,E \vdash a_{fun},M \Rightarrow \mathtt{ptr}(b,0) \qquad G,E \vdash a_{args},M \Rightarrow v_{args} \\ \mathtt{funct}(G,b) = \lfloor Fd \rfloor \qquad \mathtt{type\_of\_fundef}(Fd) = \mathtt{type}(a_{fun}) \qquad G \vdash Fd(v_{args}),M \stackrel{t}{\Rightarrow} v_{res},M' \end{array}}{G,E \vdash a_{fun}(a_{args}),M \stackrel{t}{\Rightarrow} v_{res},M'} \quad (30)$$

$$\frac{\begin{array}{c} G,E \vdash a,M \Leftarrow \ell \qquad G,E \vdash a_{fun},M \Rightarrow \mathtt{ptr}(b,0) \qquad G,E \vdash a_{args},M \Rightarrow v_{args} \\ \mathtt{funct}(G,b) = \lfloor Fd \rfloor \qquad \mathtt{type\_of\_fundef}(Fd) = \mathtt{type}(a_{fun}) \qquad G \vdash Fd(v_{args}),M \stackrel{t}{\Rightarrow} v_{res},M_1 \\ \mathtt{storeval}(\mathtt{type}(a),M_1,\mathtt{ptr}(\ell),v_{res}) = \lfloor M_2 \rfloor \end{array}}{G,E \vdash a = a_{fun}(a_{args}),M \stackrel{t}{\Rightarrow} v_{res},M_2} \quad (31)$$

Compatibility between values, outcomes and return types:

$$\mathtt{Normal}, \mathtt{void} \,\#\, \mathtt{undef} \qquad \mathtt{Return}, \mathtt{void} \,\#\, \mathtt{undef} \qquad \mathtt{Return}(v),\ \tau \,\#\, v \text{ when } \tau \neq \mathtt{void}$$

Function invocations:

$$\frac{\begin{array}{c} F = \tau\ id(dcl_1)\,\{\,dcl_2; s\,\} \\ \mathtt{alloc\_vars}(M,dcl_1+dcl_2,E) = (M_1,b^*) \qquad \mathtt{bind\_params}(E,M_1,dcl_1,v_{args}) = M_2 \\ G,E \vdash s,M_2 \stackrel{t}{\Rightarrow} out,M_3 \qquad out,\ \tau \# v_{res} \end{array}}{G \vdash F(v_{args}),M \stackrel{t}{\Rightarrow} v_{res},\mathtt{free}(M_3,b^*)} \quad (32)$$

$$\frac{Fe = \mathtt{extern}\ \tau\ id(dcl) \qquad v = id(v_{args},v_{res})}{G \vdash Fe(v_{args}),M \stackrel{v}{\Rightarrow} v_{res},M} \quad (33)$$

Fig. 10 Natural semantics for function calls

a_2 when necessary.) Note that `storeval` fails if a_1 has a `struct` or `union` type: assignments between composite data types are not supported in Clight.

The execution of a sequence of two statements starts with the execution of the first statement, thus yielding an outcome that determines whether the second statement must be executed or not (rules 21 and 22). The resulting trace is the concatenation of both traces originating from both statement executions.

The rules in figure 9 define the execution of `while` and `for` loops. (The rules describing the execution of `dowhile` loops resemble the rules for `while` loops and are omitted in this paper.) Once the condition of a `while` loop is evaluated to a value v, if v is false, the execution of the loop terminates normally, with an empty trace (rules 23 and 27). If v is true, the loop body s is executed, thus yielding an outcome *out* (rules 24, 25, 28 and 29). If *out* is `Normal` or `Continue`, the whole loop is re-executed in the memory state modified by the first execution of the body. In s, the execution of a `continue` statement interrupts the current execution of the loop body and triggers the next iteration of s. If *out* is `Break`, the loop terminates normally; if *out* is `Return`, the loop terminates prematurely with the same outcome (rules 24 and 28). The $\stackrel{loop}{\rightsquigarrow}$ relation models this evolution of outcomes after the premature end of the execution of a loop body.

Rules 26–29 describe the execution of a $\mathtt{for}(s_1,a_2,s_3)\ s$ loop. Rule 26 executes the initial statement s_1 of a `for` loop, which must terminate normally. Then, the loop with an empty initial statement is executed in a way similar to that of a `while` loop (rules 27–29). If the body s terminates normally or by performing a `continue`, the statement s_3 is executed before re-executing the `for` loop. As in the case of s_1, it must be the case that s_3 terminates normally.

$$\frac{G,E \vdash s_1,M \stackrel{T}{\Rightarrow} \infty}{G,E \vdash s_1;s_2,M \stackrel{T}{\Rightarrow} \infty} \quad (34) \qquad \frac{G,E \vdash s_1,M \stackrel{t}{\Rightarrow} \texttt{Normal},M_1 \quad G,E \vdash s_2,M_1 \stackrel{T}{\Rightarrow} \infty}{G,E \vdash s_1;s_2,M \stackrel{t.T}{\Rightarrow} \infty} \quad (35)$$

$$\frac{G,E \vdash a,M \Rightarrow v \quad \texttt{is\_true}(v,\texttt{type}(a)) \quad G,E \vdash s,M \stackrel{T}{\Rightarrow} \infty}{G,E \vdash (\texttt{while}(a)\ s),M \stackrel{T}{\Rightarrow} \infty} \quad (36)$$

$$\frac{G,E \vdash a,M \Rightarrow v \quad \texttt{is\_true}(v,\texttt{type}(a)) \quad G,E \vdash s,M \stackrel{t}{\Rightarrow} (\texttt{Normal} \mid \texttt{Continue}),M_1 \quad G,E \vdash (\texttt{while}(a)\ s),M_1 \stackrel{T}{\Rightarrow} \infty}{G,E \vdash (\texttt{while}(a)\ s),M \stackrel{t.T}{\Rightarrow} \infty} \quad (37)$$

$$\frac{G,E \vdash a_{fun},M \Rightarrow \texttt{ptr}(b,0) \quad G,E \vdash a_{args},M \Rightarrow v_{args} \quad \texttt{funct}(G,b) = \lfloor Fd \rfloor \quad \texttt{type\_of\_fundef}(Fd) = \texttt{type}(a_{fun}) \quad G \vdash Fd(v_{args}),M \stackrel{T}{\Rightarrow} \infty}{G,E \vdash a_{fun}(a_{args}),M \stackrel{T}{\Rightarrow} \infty} \quad (38)$$

$$\frac{F = \tau\ id(dcl_1)\,\{dcl_2; s\} \quad \texttt{alloc\_vars}(M,dcl_1+dcl_2,E) = (M_1,b^*) \quad \texttt{bind\_params}(E,M_1,dcl_1,v_{args}) = M_2 \quad G,E \vdash s,M_2 \stackrel{T}{\Rightarrow} \infty}{G \vdash F(v_{args}),M \stackrel{T}{\Rightarrow} \infty} \quad (39)$$

Fig. 11 Natural semantics for divergence (selected rules)

We omit the rules for $\texttt{switch}(a)$ *sw* statements, which are standard. Based on the integer value of *a*, the appropriate case of *sw* is selected, and the corresponding suffix of *sw* is executed like a sequence, therefore implementing the "fall-through" behavior of `switch` cases. A `Break` outcome for one of the cases terminates the `switch` normally.

The rules of figure 10 define the execution of a call statement $a_{fun}(a_{args})$ or $a = a_{fun}(a_{args})$. The expression a_{fun} is evaluated to a function pointer $\texttt{ptr}(b,0)$, and the reference b is resolved to the corresponding function definition Fd using the global environment G. This function definition is then invoked on the values of the arguments a_{args} as per the judgment $G \vdash Fd(v_{args}),M \stackrel{v_{res}}{\Rightarrow} t,M'$. If needed, the returned value v_{res} is then stored in the location of the l-value a (rules 30 and 31).

The invocation of an internal Clight function F (rule 32) allocates the memory required for storing the formal parameters and the local variables of F, using the `alloc_vars` function. This function allocates one block for each variable $id : \tau$, with lower bound 0 and upper bound $\texttt{sizeof}(\tau)$, using the `alloc` primitive of the memory model. These blocks initially contain `undef` values. Then, the `bind_params` function iterates the `storeval` function in order to initialize formal parameters to the values of the corresponding arguments.

The body of F is then executed, thus yielding an outcome (fourth premise). The return value of F is computed from this outcome and from the return type of F (fifth premise): for a function returning `void`, the body must terminate by `Normal` or `Return` and the return value is `undef`; for other functions, the body must terminate by $\texttt{Return}(v)$ and the return value is v. Finally, the memory blocks b^* that were allocated for the parameters and local variables are freed before returning to the caller.

A call to an external function *Fe* simply generates an input/output event recorded in the trace resulting from that call (rule 33).

$$\frac{\begin{array}{c} G = \mathtt{globalenv}(P) \qquad M = \mathtt{initmem}(P) \\ \mathtt{symbol}(G,\mathtt{main}(P)) = \lfloor b \rfloor \qquad \mathtt{funct}(G,b) = \lfloor f \rfloor \qquad G \vdash f(\mathtt{nil}), M \stackrel{t}{\Rightarrow} \mathtt{int}(n), M' \end{array}}{\vdash P \Rightarrow \mathtt{terminates}(t,n)} \quad (40)$$

$$\frac{\begin{array}{c} G = \mathtt{globalenv}(P) \qquad M = \mathtt{initmem}(P) \\ \mathtt{symbol}(G,\mathtt{main}(P)) = \lfloor b \rfloor \qquad \mathtt{funct}(G,b) = \lfloor f \rfloor \qquad G \vdash f(\mathtt{nil}), M \stackrel{T}{\Rightarrow} \infty \end{array}}{\vdash P \Rightarrow \mathtt{diverges}(T)} \quad (41)$$

Fig. 12 Observable behaviors of programs

3.4 Statements and function invocations, diverging case

Figure 11 shows some of the rules that model divergence of statements and function invocations. As denoted by the double horizontal bars, these rules are to be interpreted *coinductively*, as greatest fixpoints, instead of the standard inductive interpretation (smallest fixpoints) used for the other rules in this paper. In other words, just like terminating executions correspond to finite derivation trees, diverging executions correspond to infinite derivation trees [30].

A sequence $s_1; s_2$ diverges either if s_1 diverges, or if s_1 terminates normally and s_2 diverges (rules 34 and 35). Likewise, a loop diverges either if its body diverges, or if it terminates normally or by `continue` and the next iteration of the loop diverges (rules 36 and 37). A third case of divergence corresponds to an invocation of a function whose body diverges (rules 38 and 39).

3.5 Program executions

Figure 12 defines the execution of a program P and the determination of its observable behavior. A global environment and a memory state are computed for P, where each global variable is mapped to a fresh memory block. Then, the main function of P is resolved and applied to the empty list of arguments. If this function invocation terminates with trace t and result value $\mathtt{int}(n)$, the observed behavior of P is $\mathtt{terminates}(t,n)$ (rule 40). If the function invocation diverges with a possibly infinite trace T, the observed behavior is $\mathtt{diverges}(T)$ (rule 41).

4 Using Clight in the CompCert compiler

In this section, we informally discuss how Clight is used in the CompCert verified compiler [27, 6, 28].

4.1 Producing Clight abstract syntax

Going from C concrete syntax to Clight abstract syntax is not as obvious as it may sound. After an unsuccessful attempt at developing a parser, type-checker and simplifier from scratch, we elected to reuse the CIL library of Necula *et al.* [33]. CIL is written in OCaml and provides the following facilities:

1. A parser for ISO C99 (plus GCC and Microsoft extensions), producing a parse tree that is still partially ambiguous.
2. A type-checker and elaborator, producing a precise, type-annotated abstract syntax tree.
3. A simplifier that replaces many delicate constructs of C by simpler constructs. For instance, function calls and assignments are pulled out of expressions and lifted to the statement level. Also, block-scoped variables are lifted to function scope or global scope.
4. A toolkit for static analyses and transformations performed over the simplified abstract syntax tree.

While conceptually distinct, (2) and (3) are actually performed in a single pass, avoiding the creation of the non-simplified abstract syntax tree.

Thomas Moniot and the authors developed (in OCaml) a simple translator that produces Clight abstract syntax from the output of CIL. Much information produced by CIL is simply erased, such as type attributes and qualifiers. `struct` and `union` types are converted from the original named representation to the structural representation used by Clight. String literals are turned into global, initialized arrays of characters. Finally, constructs of C that are unsupported in Clight are detected and meaningful diagnostics are produced.

The simplification pass of CIL sometimes goes too far for our needs. In particular, the original CIL transforms all C loops into `while(1) { ... }` loops, sometimes inserting `goto` statements to implement the semantics of `continue`. Such CIL-inserted `goto` statements are problematic in Clight. We therefore patched CIL to remove this simplification of C loops and natively support `while`, `do` and `for` loops

CIL is an impressive but rather complex piece of code, and it has not been formally verified. One can legitimately wonder whether we can trust CIL and our hand-written translator to preserve the semantics of C programs. Indeed, two bugs in this part of CompCert were found during testing: one that we introduced when adding native support for `for` loops; another that is present in the unmodified CIL version 1.3.6, but was corrected since then.

We see two ways to address this concern. First, we developed a pretty-printer that displays Clight abstract syntax tree in readable, C concrete syntax. This printer makes it possible to conduct manual reviews of the transformations performed by CIL. Moreover, experiment shows that re-parsing and re-transforming the simplified C syntax printed from the Clight abstract syntax tree reaches a fixed point in one iteration most of the time. This does not prove anything but nonetheless instills some confidence in the approach.

A more radical way to establish trust in the CIL-based Clight producer would be to formally verify some of the simplifications performed. A prime candidate is the simplification of expressions, which transforms C expressions into equivalent pairs of a statement (performing all side effects of the expression) and a pure expression (computing the final value). Based on initial experiments on a simple "while" language, the Coq verification of this simplification appears difficult but feasible. We leave this line of work for future work.

4.2 Compiling Clight

The CompCert C compiler is structured in two parts: a front-end compiler translates Clight to an intermediate language called Cminor, without performing any optimizations; a back-end compiler generates PowerPC assembly code from the Cminor intermediate representation, performing good register allocation and a few optimizations. Both parts are composed of multiple passes. Each pass is proved to preserve semantics: if the input program P has observable behavior B, and the pass translates P to P' without reporting a compile-time error, then the output program P' has the same observable behavior B. The proofs of semantic

preservation are conducted with the Coq proof assistant. To facilitate the proof, the compiler passes are written directly in the specification language of Coq, as pure, recursive functions. Executable Caml code for the compiler is then generated automatically from the functional specifications by Coq's extraction facility.

The back-end part of CompCert is described in great detail in [28]. We now give an overview of the front-end, starting with a high-level overview of Cminor, its target intermediate language. (Refer to [28, section 4] for detailed specifications of Cminor.)

Cminor is a low-level imperative language, structured like Clight into expressions, statements, and functions. A first difference with Clight is that arithmetic operators are not overloaded and their behavior is independent of the static types of their operands: distinct operators are provided for integer arithmetic and floating-point arithmetic. Conversions between integers and floats are explicit. Likewise, address computations are explicit in Cminor, as well as individual load and store operations. For instance, the C expression `a[x]` where `a` is a pointer to `int` is expressed as `load(int32, a +`$_i$ `x *`$_i$ `4)`, making explicit the memory quantity being addressed (`int32`) as well as the address computation.

At the level of statements, Cminor has only 5 control structures: if-then-else conditionals, infinite loops, `block`-`exit`, early return, and `goto` with labeled statements. The `exit` n statement terminates the $(n+1)$ enclosing `block` statements.

Within Cminor functions, local variables can only hold scalar values (integers, pointers, floats) and they do not reside in memory. This makes it easy to allocate them to registers later in the back-end, but also prohibits taking a pointer to a local variable like the C operator `&` does. Instead, each Cminor function declares the size of a stack-allocated block, allocated in memory at function entry and automatically freed at function return. The expression `addrstack`(n) returns a pointer within that block at constant offset n. The Cminor producer can use this block to store local arrays as well as local scalar variables whose addresses need to be taken.

To translate from Clight to Cminor, the front-end of CompCert C therefore performs the following transformations:

1. Resolution of operator overloading and materialization of all type-dependent behaviors. Based on the types that annotate Clight expressions, the appropriate flavors (integer or float) of arithmetic operators are chosen; conversions between ints and floats, truncations and sign-extensions are introduced to reflect casts; address computations are generated based on the types of array elements and pointer targets; and appropriate memory chunks are selected for every memory access.
2. Translation of `while`, `do` and `for` loops into infinite loops with blocks and early exits. The `break` and `continue` statements are translated as appropriate `exit` constructs.
3. Placement of Clight variables, either as Cminor local variables (for local scalar variables whose address is never taken), sub-areas of the Cminor stack block for the current function (for local non-scalar variables or local scalar variables whose address is taken), or globally allocated memory areas (for global variables).

In the first version of the front-end, developed by Zaynah Dargaye and the authors and published in [6], the three transformations above were performed in a single pass, resulting in a large and rather complex proof of semantic preservation. To make the proofs more manageable, we split the front-end in two passes: the first performs transformations (1) and (2) above, and the second performs transformation (3). A new intermediate language called C#minor was introduced to connect the two passes. C#minor is similar to Cminor, except that it supports a `&` operator to take the address of a local variable. Accordingly,

the semantics of C#minor, like that of Clight, allocates one memory block for each local variable at function entrance, while the semantics of Cminor allocates only one block.

To account for this difference in allocation patterns, the proof of semantic preservation for transformation (3) exploits the technique of *memory injections* formalized in [29, section 5.4]. It also involves nontrivial reasoning about separation between memory blocks and between sub-areas of a block. The proof requires about 2200 lines of Coq, plus 800 lines for the formalization of memory injections.

The proof of transformations (1) and (2) is more routine: since the memory states match exactly between the original Clight and the generated C#minor, no clever reasoning over memory states, blocks and pointers is required. The Coq proof remains relatively large (2300 lines), but mostly because many cases need to be considered, especially when resolving overloaded operators.

5 Validating the Clight semantics

Developing a formal semantics for a real-world programming language is no small task; but making sure that the semantics captures the intended behaviors of programs (as described, for example, by ISO standards) is even more difficult. The smallest mistake or omission in the rules of the semantics can render it incomplete or downright incorrect. Below, we list a number of approaches that we considered to validate a formal semantics such as that of Clight. Many of these approaches were prototyped but not carried to completion, and should be considered as work in progress.

5.1 Manual reviews

The standard way to build confidence in a formal specification is to have it reviewed by domain experts. The size of the semantics for Clight makes this approach tedious but not downright impossible: about 800 lines of Coq for the core semantics, plus 1000 lines of Coq for dependencies such as the formalizations of machine integers, floating-point numbers, and the memory model. The fact that the semantics is written in a formal language such as Coq instead of ordinary mathematics is a mixed blessing. On the one hand, the type-checking performed by Coq guarantees the absence of type errors and undefined predicates in the specification, while such trivial errors are common in hand-written semantics. On the other hand, domain experts might not be familiar with the formal language used and could prefer more conventional presentations as e.g. inference rules. (We have not yet found any C language expert who is comfortable with Coq, while several of them are fluent with inference rules.) Manual transliteration of Coq specifications into LaTeX inference rules (as we did in this paper) is always possible but can introduce or (worse) mask errors. Better approaches include automatic generation of LaTeX from formal specifications, like Isabelle/HOL and Ott do [35, 43].

5.2 Proving properties of the semantics

The primary use of formal semantics is to prove properties of programs and meta-properties of the semantics. Such proofs, especially when conducted on machine, are effective at revealing errors in the semantics. For example, in the case of strongly-typed languages, type

soundness proofs (showing that well-typed programs do not go wrong) are often used for this purpose. In the case of Clight, a type soundness proof is not very informative, since the type system of C is coarse and unsound to begin with: the best we could hope for is a subject reduction property, but the progress property does not hold. Less ambitious sanity checks include "common sense" properties such as those of the `field_offset` function mentioned at end of section 2.1, as well as determinism of evaluation, which we obtained as a corollary of the verification of the CompCert compiler [28, sections 2.1 and 13.3].

5.3 Verified translations

Extending the previous approach to proving properties involving two formal semantics instead of one, we found that proving semantics preservation for a translation from one language to another is effective at exposing errors not only in the translation algorithm, but also in the semantics of the two languages involved. If the translation "looks right" to compiler experts and the semantics of the target language has already been debugged, such a proof of semantic preservation therefore generates confidence in the semantics of the source language. In the case of CompCert, the semantics of the Cminor intermediate language is smaller (300 lines) and much simpler than that of Clight; subsequent intermediate languages in the back-end such as RTL are even simpler, culminating in the semantics of the PPC assembly language, which is a large but conceptually trivial transition function [28]. The existence of semantic-preserving translations between these languages therefore constitutes an indirect validation of their semantics.

Semantic preservation proofs and type soundness proofs detect different kinds of errors in semantics. For a trivial example, assume that the Clight semantics erroneously interprets the `+` operator at type `int` as integer subtraction. This error would not invalidate an hypothetical type soundness proof, but would show up immediately in the proof of semantic preservation for the CompCert front-end, assuming of course that we did not commit the same error in the translations nor in the semantics of Cminor. On the other hand, a type soundness proof can reveal that an evaluation rule is missing (this shows up as failures of the progress property). A semantic preservation proof can point out a missing rule in the semantics of the target language but not in the semantics of the source language, since it takes as hypothesis that the source program does not go wrong.

5.4 Testing executable semantics

Just like programs, formal specifications can be tested against test suites that exemplifies expected behaviors. An impressive example of this approach is the HOL specification of the TCP/IP protocol by Sewell *et al.* [5], which was extensively validated against network traces generated by actual implementations of the protocol.

In the case of formal semantics, testing requires that the semantics is *executable*: there must exist an effective way to determine the result of a given program in a given initial environment. The Coq proof assistant does not provide efficient ways to execute a specification written using inductive predicates such as our semantics for Clight. (But see [11] for ongoing work in this direction.) As discussed in [3], the `eauto` tactic of Coq, which performs Prolog-style resolution, can sometimes be used as the poor man's logic interpreter to execute inductive predicates. However, the Clight semantics is too large and not syntax-directed enough to render this approach effective.

On the other hand, Coq provides excellent facilities for executing specifications written as recursive functions: an interpreter is built in the Coq type-checker to perform conversion tests; Coq 8.0 introduced a bytecode compiler to a virtual machine, speeding up the evaluation of Coq terms by one order of magnitude [15]; finally, the extraction facility of Coq can also be used to generate executable Caml code. The recommended approach to execute a Coq specification by inductive predicates, therefore, is to define a reference interpreter as a Coq function, prove its equivalence with the inductive specification, and evaluate applications of the function. Since Coq demands that all recursive functions terminate, these interpretation functions are often parameterized by a nonnegative integer counter n bounding the depth of the evaluation. Taking the execution of Clight statements as an example, the corresponding interpretation function is of the shape

$$\texttt{exec\_stmt}(W,n,G,E,M,s) = \texttt{Bottom}(t) \mid \texttt{Result}(t,out,M') \mid \texttt{Error}$$

where n is the maximal recursion depth, G,E,M are the initial state, and s the statement to execute. The result of execution is either `Error`, meaning that execution goes wrong, or $\texttt{Result}(t,out,M')$, meaning that execution terminates with trace t, outcome out and final memory state M', or $\texttt{Bottom}(t)$, meaning that the maximal recursion depth was exceeded after producing the partial trace t. To handle the non-determinism introduced by input/output operations, `exec_stmt` is parameterized over a *world* W: a partial function that determines the result of an input/output operation as a function of its arguments and the input/output operation previously performed [28, section 13.1].

The following two properties characterize the correctness of the `exec_stmt` function with respect to the inductive specification of the semantics:

$$G,E \vdash s,M \stackrel{t}{\Rightarrow} out,M' \;\wedge\; W \models t \;\Leftrightarrow\; \exists n,\ \texttt{exec\_stmt}(W,n,G,E,M,s) = \texttt{Result}(t,out,M)$$

$$G,E \vdash s,M \stackrel{T}{\Rightarrow} \infty \;\wedge\; W \models T \;\Leftrightarrow\; \forall n, \exists t,\ \texttt{exec\_stmt}(W,n,G,E,M,s) = \texttt{Bottom}(t) \;\wedge\; t \text{ is a prefix of } T$$

Here, $W \models t$ means that the trace t is consistent with the world W, in the sense of [28, section 13.1]. See [30] for detailed proofs of these properties in the simpler case of call-by-value λ-calculus without traces. The proof of the second property requires classical reasoning with the axiom of excluded middle.

We are currently implementing the approach outlined above, although it is not finished at the time of this writing. Given the availability of the CompCert verified compiler, one may wonder what is gained by using a reference Clight interpreter to run tests, instead of just compiling them with CompCert and executing the generated PowerPC assembly code. We believe that nothing is gained for test programs with well-defined semantics. However, the reference interpreter enables us to check that programs with undefined semantics do go wrong, while the CompCert compiler can (and often does) turn them into correct PowerPC code.

5.5 Equivalence with alternate semantics

Yet another way to validate a formal semantics is to write several alternate semantics for the same language, using different styles of semantics, and prove logical implications between them. In the case of the Cminor intermediate language and with the help of Andrew Appel, we developed three semantics: (1) a big-step operational semantics in the style of the Clight semantics described in the present paper [27]; (2) a small-step, continuation-based semantics

[2, 28]; (3) an axiomatic semantics based on separation logic [2]. Semantics (1) and (3) were proved correct against semantics (2). Likewise, for Clight and with the help of Keiko Nakata, we prototyped (but did not complete yet) three alternate semantics to the big-step operational semantics presented in this paper: (1) a small-step, continuation-based semantics; (2) the reference interpreter outlined above; (3) an axiomatic semantics.

Proving the correctness of a semantics with respect to another is an effective way to find mistakes in both. For instance, the correctness of an axiomatic semantics against a big-step operational semantics without traces can be stated as follows: if $\{P\}s\{Q\}$ is a valid Hoare triple, then for all initial states G, E, M satisfying the precondition P, either the statement s diverges ($G, E \vdash s, M \Rightarrow \infty$) or it terminates ($G, E \vdash s, M \Rightarrow out, M'$) and the outcome out and the final state G, E, M' satisfy postcondition Q. The proof of this property exercises all cases of the big-step operational semantics and is effective at pointing out mistakes and omissions in the latter. Extending this approach to traces raises delicate issues that we have not solved yet. First, the axiomatic semantics must be extended with ways for the postconditions Q to assert properties of the traces generated by the execution of the statement s. A possible source of inspiration is the recent work by Hoare and O'Hearn [20]. Second, in the case of a loop such as $\{P\}$ `while`(a) s $\{Q\}$, we must not only show that the loop either terminates or diverges without going wrong, as in the earlier proof, but also prove the existence of the corresponding traces of events. In the diverging case, this runs into technical problems with Coq's guardedness restrictions on coinductive definitions and proofs.

In the examples given above, the various semantics were written by the same team and share some elements, such as the memory model and the semantics of Clight expressions. Mistakes in the shared parts will obviously not show up during the equivalence proofs. Relating two independently-written semantics would provide a more convincing validation. In our case, an obvious candidate for comparison with Clight is the Cholera semantics of Norrish [36]. There are notable differences between our semantics and Cholera, discussed in section 6, but we believe that our semantics is a refinement of the Cholera model. A practical issue with formalizing this intuition is that Cholera is formalized in HOL while our semantics is formalized in Coq.

6 Related work

Mechanized semantics for C The work closest to ours is Norrish's Cholera project [36], which formalizes the static and dynamic semantics of a large subset of C using the HOL proof assistant. Unlike Clight, Cholera supports side effects within expressions and accounts for the partially specified evaluation order of C. For this purpose, the semantics of expressions is given in small-step style as a non-deterministic reduction relation, while the semantics of statements is given in big-step style. Norrish used this semantics to characterize precisely the amount of non-determinism allowed by the C standard [37]. Also, the memory model underlying Cholera is more abstract than that of Clight, leaving unspecified a number of behaviors that Clight specifies.

Tews *et al* [46, 47] developed a denotational semantics for a subset of the C++ language. The semantics is presented as a shallow embedding in the PVS prover. Expressions and statements are modeled as state transformers: functions from initial states to final states plus value (for expressions) or outcome (for statements). The subset of C++ handled is close to our Clight, with a few differences: side effects within expressions are allowed (and treated using a fixed evaluation order); the behavior of arithmetic operations in case of overflow is

not specified; the `goto` statement is not handled, but the state transformer approach could be extended to do so [45].

Using the Coq proof assistant, Giménez and Ledinot [14] define a denotational semantics for a subset of C appropriate as target language for the compilation of the Lustre synchronous dataflow language. Owing to the particular shape of Lustre programs, the subset of C does not contain general loops nor recursive functions, but only counted `for` loops. Pointer arithmetic is not supported.

As part of the Verisoft project [40], the semantics of a subset of C called C0 has been formalized using Isabelle/HOL, as well as the correctness of a compiler from C0 to DLX assembly language [26, 44, 41]. C0 is a type-safe subset of C, close to Pascal, and significantly smaller than Clight: for instance, there is no pointer arithmetic, nor `break` and `continue` statements. A big-step semantics and a small-step semantics have been defined for C0, the latter enabling reasoning about non-terminating executions.

Paper and pencil semantics for C Papaspyrou [39] develops a monadic denotational semantics for most of ISO C. Non-determinism in expression evaluation is modeled precisely. The semantics was validated by testing with the help of a reference interpreter written in Haskell.

Nepomniaschy *et al.* [34] define a big-step semantics for a subset of C similar to Pascal: it supports limited uses of `goto` statements, but not pointer arithmetic.

Abstract state machines have been used to give semantics for C [16] and for C# [7]. The latter formalization is arguably the most complete (in terms of the number of language features handled) formal semantics for an imperative language.

Other examples of mechanized semantics Proof assistants were used to mechanize semantics for languages that are higher-level than C. Representative examples include [23, 25] for Standard ML, [38] for a subset of OCaml, and [24] for a subset of Java. Other Java-related mechanized verifications are surveyed in [18]. Many of these semantics were validated by conducting type soundness proofs.

Subsets of C Many uses of C in embedded or critical applications mandate strict coding guidelines restricting programmers to a "safer" subset of C [19]. A well-known example is MISRA C [32]. MISRA C and Clight share some restrictions (such as structured `switch` statements with `default` cases at the end), but otherwise differ significantly. For instance, MISRA C prohibits recursive functions, but permits all uses of `goto`. More generally, the restrictions of MISRA C and related guidelines are driven by software engineering considerations and the desire for tool-assisted checking, while the restrictions of Clight stem from the desire to keep its formal semantics manageable.

Several tools for static analysis and deductive verification of C programs use simplified subsets of C as intermediate representations. We already discussed the CIL intermediate representation [33]. Other examples include the Frama-C intermediate representation [8], which extends CIL's with logical assertions, and the Newspeak representation [22]. CIL is richer than Clight and accurately represents all of ISO C plus some extensions. Newspeak is lower-level than Clight and targeted more towards static analysis than towards compilation.

7 Conclusions and future work

In this article, we have formally defined the Clight subset of the C programming language and its dynamic semantics. While there is no general agreement on the formal semantics of

the C language, we believe that Clight is a reasonable proposal that works well in the context of the formal verification of a compiler. We hope that, in the future, Clight might be useful in other contexts such as static analyzers and program provers and their formal verification.

Several extensions of Clight can be considered. One direction, discussed in [29], is to relax the memory model so as to model byte- and bit-level accesses to in-memory data representations, as is commonly done in systems programming.

Another direction is to add support for some of the C constructs currently missing, in particular the `goto` statement. The main issue here is to formalize the dynamic semantics of `goto` in a way that lends itself well to proofs. Natural semantics based on statement outcomes can be extended with support for `goto` by following the approach proposed by Tews [45], but at the cost of nearly doubling the size of the semantics. Support for `goto` statements is much easier to add to transition semantics based on continuations, as the Cminor semantics exemplifies [28, section 4]. However, such transition semantics do not lend themselves easily to proving transformations of loops such as those performed by the front-end of CompCert (transformation 2 in section 4.2).

Finally, the restriction that Clight expressions are pure is both a blessing and a curse: on the one hand, it greatly simplifies all further processing of Clight, be it compilation, static analysis or program verification; on the other hand, programmers cannot be expected to directly write programs where all expressions are pure, requiring nontrivial, untrusted program transformations in the Clight parser. One way to address this issue would be to define an extension of Clight, tentatively called Cmedium, that supports side effects within expressions, and develop and prove correct a translation from Cmedium to Clight.

References

1. Aiken, A., Bugrara, S., Dillig, I., Dillig, T., Hackett, B., Hawkins, P.: An overview of the Saturn project. In: PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp. 43–48. ACM Press (2007)
2. Appel, A.W., Blazy, S.: Separation logic for small-step Cminor. In: Theorem Proving in Higher Order Logics, 20th Int. Conf. TPHOLs 2007, *Lecture Notes in Computer Science*, vol. 4732, pp. 5–21. Springer (2007)
3. Appel, A.W., Leroy, X.: A list-machine benchmark for mechanized metatheory (extended abstract). In: Proc. Int. Workshop on Logical Frameworks and Meta-Languages (LFMTP'06), *Electronic Notes in Computer Science*, vol. 174/5, pp. 95–108 (2007)
4. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions. EATCS Texts in Theoretical Computer Science. Springer (2004)
5. Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., Wansbrough, K.: Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In: 33rd Symposium on Principles of Programming Languages, pp. 55–66. ACM Press (2006)
6. Blazy, S., Dargaye, Z., Leroy, X.: Formal verification of a C compiler front-end. In: FM 2006: 14th Int. Symp. on Formal Methods, *Lecture Notes in Computer Science*, vol. 4085, pp. 460–475. Springer (2006)
7. Börger, E., Fruja, N., Gervasi, V., Stärk, R.F.: A high-level modular definition of the semantics of C#. Theoretical Computer Science **336**(2-3), 235–284 (2005)
8. CEA LIST: FRAMA-C: Framework for modular analysis of C. Software and documentation available on the Web (2008). URL `http://frama-c.cea.fr/`
9. Condit, J., Harren, M., McPeak, S., Necula, G.C., Weimer, W.: CCured in the real world. In: PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, pp. 232–244. ACM Press (2003)
10. Coq development team: The Coq proof assistant. Software and documentation available on the Web (1989-2008). URL `http://coq.inria.fr/`
11. Delahaye, D., Dubois, C., Étienne, J.F.: Extracting purely functional contents from logical inductive types. In: Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, *Lecture Notes in Computer Science*, vol. 4732, pp. 70–85. Springer (2007)

12. Duff, T.: On Duff's device (1988). URL `http://www.lysator.liu.se/c/duffs-device.html`. Message to the comp.lang.c Usenet group
13. Filliâtre, J.C., Marché, C.: Multi-prover verification of C programs. In: 6th Int. Conference on Formal Engineering Methods, ICFEM 2004, *Lecture Notes in Computer Science*, vol. 3308, pp. 15–29 (2004)
14. Gimenez, E., Ledinot, E.: Semantics of a subset of the C language (2004). URL `http://coq.inria.fr/contribs/minic.html`. Coq contributed library
15. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: International Conference on Functional Programming (ICFP 2002), pp. 235–246. ACM Press (2002)
16. Gurevich, Y., Huggins, J.: The semantics of the C programming language. In: Computer Science Logic, 6th Workshop, CSL '92, *Lecture Notes in Computer Science*, vol. 702, pp. 274–308. Springer (1993)
17. Hardekopf, B., Lin, C.: The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. SIGPLAN Notices **42**(6), 290–299 (2007)
18. Hartel, P.H., Moreau, L.: Formalizing the safety of Java, the Java virtual machine, and Java card. ACM Computing Surveys **33**(4), 517–558 (2001)
19. Hatton, L.: Safer language subsets: an overview and a case history, MISRA C. Information & Software Technology **46**(7), 465–472 (2004)
20. Hoare, T., O'Hearn, P.W.: Separation logic semantics for communicating processes. In: Proceedings of the First International Conference on Foundations of Informatics, Computing and Software (FICS 2008), *Electronic Notes in Computer Science*, vol. 212, pp. 3–25 (2008)
21. Huisman, M., Jacobs, B.: Java program verification via a Hoare logic with abrupt termination. In: Fundamental Approaches to Software Engineering, 3rd Int. Conf. FASE 2000, *Lecture Notes in Computer Science*, vol. 1783, pp. 284–303. Springer (2000)
22. Hymans, C., Levillain, O.: Newspeak, doubleplussimple minilang for goodthinkful static analysis of C. Technical note 2008-IW-SE-00010-1, EADS (2008)
23. van Inwegen, M., Gunter, E.L.: HOL-ML. In: Higher Order Logic Theorem Proving and its Applications, 6th International Workshop, HUG '93, *Lecture Notes in Computer Science*, vol. 780, pp. 61–74. Springer (1993)
24. 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 (2006)
25. Lee, D.K., Crary, K., Harper, R.: Towards a mechanized metatheory of Standard ML. In: 34th Symposium on Principles of Programming Languages, pp. 173–184. ACM Press (2007)
26. Leinenbach, D., Paul, W., Petrova, E.: Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In: IEEE Conference on Software Engineering and Formal Methods (SEFM'05), pp. 2–11. IEEE Computer Society Press (2005)
27. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd ACM symposium on Principles of Programming Languages, pp. 42–54. ACM Press (2006)
28. Leroy, X.: A formally verified compiler backend (2008). URL `http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf`. Submitted for publication
29. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. Journal on Automated Reasoning **41**(1), 1–31 (2008)
30. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Information and Computation (2007). URL `http://dx.doi.org/10.1016/j.ic.2007.12.004`. To appear
31. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The definition of Standard ML (revised). The MIT Press (1997)
32. Motor Industry Software Reliability Association: MISRA-C. `http://www.misra-c.com/` (2004)
33. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Compiler Construction, 11th International Conference, CC 2002, *Lecture Notes in Computer Science*, vol. 2304, pp. 213–228. Springer (2002)
34. Nepomniaschy, V.A., Anureev, I.S., Promsky, A.V.: Towards verification of C programs: Axiomatic semantics of the C-kernel language. Programming and Computer Software **29**(6), 338–350 (2003)
35. Nipkow, T., Paulson, L.C.: Isabelle/Hol: A Proof Assistant for Higher-Order Logic. Springer (2004)
36. Norrish, M.: C formalised in HOL. Ph.D. thesis, University of Cambridge (1998). Technical report UCAM-CL-TR-453
37. Norrish, M.: Deterministic expressions in C. In: Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, *Lecture Notes in Computer Science*, vol. 1576, pp. 147–161. Springer (1999)
38. Owens, S.: A sound semantics for OCamllight. In: Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, *Lecture Notes in Computer Science*, vol. 4960, pp. 1–15. Springer (2008)

39. Papaspyrou, N.: A formal semantics for the C programming language. Ph.D. thesis, National Technical University of Athens (1998)
40. Paul, W., et al.: The Verisoft project (2003–2008). URL `http://www.verisoft.de/`
41. Schirmer, N.: Verification of sequential imperative programs in Isabelle/HOL. Ph.D. thesis, Technische Universität München (2006)
42. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ESEC/FSE-13: Proceedings of the 10th European software engineering conference, pp. 263–272. ACM Press (2005)
43. Sewell, P., Zappa Nardelli, F., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strnisa, R.: Ott: effective tool support for the working semanticist. In: Proceedings of the 12th International Conference on Functional Programming, pp. 1–12. ACM Press (2007)
44. Strecker, M.: Compiler verification for C0. Tech. rep., Université Paul Sabatier, Toulouse (2005)
45. Tews, H.: Verifying Duff's device: A simple compositional denotational semantics for goto and computed jumps (2004). URL `http://www.cs.ru.nl/~tews/Goto/goto.pdf`. Draft paper
46. Tews, H., Weber, T., Völp, M.: A formal model of memory peculiarities for the verification of low-level operating-system code. In: Proceedings of the International Workshop on Systems Software Verification (SSV'08), *Electronic Notes in Computer Science*, vol. 217, pp. 79–96 (2008)
47. Tews, H., Weber, T., Völp, M., Poll, E., van Eekelen, M., van Rossum, P.: Nova micro-hypervisor verification. Robin project deliverable D13, Radboud Universiteit Nijmegen (2008). URL `http://robin.tudos.org/D.13%20Formal%20Verification.pdf`
48. Zucker, S., Karhi, K.: System V application binary interface, PowerPC processor supplement. Tech. Rep. 802-3334-10, SunSoft (1995)