Formal Verification of a C Compiler Front-end

Sandrine Blazy, Zaynah Dargaye, Xavier Leroy

Formal Verification of a C Compiler Front-end

Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy

INRIA Rocquencourt
78153 Le Chesnay, France
{Sandrine.Blazy, Zaynah.Dargaye, Xavier.Leroy}@inria.fr

Abstract. This paper presents the formal verification of a compiler front-end that translates a subset of the C language into the Cminor intermediate language. The semantics of the source and target languages as well as the translation between them have been written in the specification language of the Coq proof assistant. The proof of observational semantic equivalence between the source and generated code has been machine-checked using Coq. An executable compiler was obtained by automatic extraction of executable Caml code from the Coq specification of the translator, combined with a certified compiler back-end generating PowerPC assembly code from Cminor, described in previous work.

1 Introduction

Formal methods in general and program proof in particular are increasingly being applied to safety-critical software. These applications create a strong need for on-machine formalization and verification of programming language semantics and tools such as compilers, type-checkers and static analyzers. In particular, formal operational semantics are required to validate the logic of programs (e.g. axiomatic semantics) used to reason about programs. As for tools, the formal certification of compilers—that is, a proof that the generated executable code behaves as prescribed by the semantics of the source program—is needed to ensure that the guarantees obtained by formal verification of the source program carry over to the executable code.

For high-level programming languages such as Java and functional languages, there exists a considerable body of on-machine formalizations and verifications of operational semantics and programming tools such as compilers and bytecode verifiers. Despite being more popular for writing critical embedded software, lower-level languages such as C have attracted less interest: several formal semantics for various subsets of C have been published, but few have been carried on machine. (See section 5 for a review.)

The work presented in this paper is part of an ongoing project that aims at developing a realistic compiler for the C language and formally verifying that it preserves the semantics of the programs being compiled. A previous paper [8] describes the verification, using the Coq proof assistant, of the back-end of this compiler, which generates moderately optimized PowerPC assembly code from a low-level, imperative intermediate language called Cminor. The present paper

reports on the development and proof of semantic preservation in Coq of a C front-end for this compiler: a translator from Clight, a subset of the C language, to Cminor. To conduct the verification, a precise operational semantics of Clight was formalized in Coq. Clight features all C arithmetic types and operators, as well as arrays, pointers, pointers to functions, and all C control structures except `goto` and `switch`.

From a formal methods standpoint, this work is interesting in two respects. First, compilers are complex programs that perform sophisticated symbolic computations. Their formal verification is challenging, requiring difficult proofs by induction that are beyond the reach of many program provers. Second, proving the correctness of a compiler provides an indirect but original way to validate the semantics of the source language. It is relatively easy to formalize an operational semantics, but much harder to make sure that this semantics is correct and captures the intended meaning of programs. Typically, extensive testing and manual reviews of the semantics are needed. In our experience, proving the correctness of a translator to a simpler, lower-level language detects many small errors in the semantics of the source and target languages, and therefore generates additional confidence in both.

The remainder of this paper is organized as follows. Section 2 describes the Clight language and gives an overview of its operational semantics. Section 3 presents the translation from Clight to Cminor. Section 4 outlines the proof of correctness of this translation. Related work is discussed in section 5, followed by conclusions in section 6.

2 The Clight Language and its Semantics

2.1 Abstract Syntax

The abstract syntax of Clight is given in figure 1. In the Coq formalization, this abstract syntax is presented as inductive data types, therefore achieving a deep embedding of Clight into Coq.

At the level of types, Clight features all the integral types of C, along with array, pointer and function types; `struct`, `union` and `typedef` types are currently omitted. The integral types fully specify the bit size of integers and floats, unlike the semi-specified C types `int`, `long`, etc.

Within expressions, all C operators are supported except those related to structs and unions. Expressions may have side-effects. All expressions and their sub-expressions are annotated by their static types. We write a^τ for the expression a carrying type τ. These types are necessary to determine the semantics of type-dependent operators such as the overloaded arithmetic operators. Similarly, combined arithmetic-assignment operators such as `+=` carry an additional type σ (as in $(a_1 \texttt{+=}^\sigma a_2)^\tau$) representing the result type of the arithmetic operation, which can differ from the type τ of the whole expression.

At the level of statements, all structured control statements of C (conditional, loops, `break`, `continue` and `return`) are supported, but not unstructured statements (`goto`, `switch`, `longjmp`). Two kinds of variables are allowed:

Types:

signedness	::= `Signed` | `Unsigned`	
intsize	::= `I8` | `I16` | `I32`	
floatsize	::= `F32` | `F64`	
τ	::= `Tint`$(intsize, signedness)$ | `Tfloat`$(floatsize)$	
	| `Tarray`(τ, n) | `Tpointer`(τ) | `Tvoid` | `Tfunction`(τ^*, τ)	

Expressions annotated with types:

a	::= b^τ	

Unannotated expressions:

b	::= id	variable identifier
	| n | f	integer or float constant
	| `sizeof`(τ)	size of a type
	| $op_u\ a$	unary arithmetic operation
	| $a_1\ op_b\ a_2$ | $a_1\ op_r\ a_2$	binary arithmetic operation
	| `*`a	dereferencing
	| $a_1[a_2]$	array indexing
	| `&`a	address of
	| `++`a | `--`a | a`++` | a`--`	pre/post increment/decrement
	| $(\tau)a$	cast
	| $a_1 = a_2$	assignment
	| $a_1\ op_b =^\tau\ a_2$	arithmetic with assignment
	| a_1 `&&` a_2 | a_1 `||` a_2	sequential boolean operations
	| a_1, a_2	sequence of expressions
	| $a(a^*)$	function call
	| a_1 `?` a_2 `:` a_3	conditional expression
op_b	::= `+` | `-` | `*` | `/` | `%`	arithmetic operators
	| `<<` | `>>` | `&` | `|` | `^`	bitwise operators
op_r	::= `<` | `<=` | `>` | `>=` | `==` | `!=`	relational operators
op_u	::= `-` | `~` | `!`	unary operators

Statements:

s	::= `skip`	empty statement
	| a;	expression evaluation
	| $s_1; s_2$	sequence
	| `if`(a) s_1 `else` s_2	conditional
	| `while`(a) s	"while" loop
	| `do` s `while`(a)	"do" loop
	| `for`$(a_1^?, a_2^?, a_3^?)$ s	"for" loop
	| `break`	exit from the current loop
	| `continue`	next iteration of the current loop
	| `return` $a^?$	return from current function

Functions:

fn	::= $(\ldots id_i : \tau_i \ldots) : \tau$	declaration of type and parameters
	{ $\ldots \tau_j\ id_j; \ldots$	declaration of local variables
	s }	function body

Fig. 1. Abstract syntax of Clight. a^* denotes 0, 1 or several occurrences of syntactic category a. $a^?$ denotes an optional occurrence of category a.

global variables and local `auto` variables declared at the beginning of a function. Block-scoped local variables and `static` variables are omitted, but can be emulated by pulling their declarations to function scope or global scope, respectively. Consequently, there is no block statement in Clight.

A Clight program consists of a list of function definitions, a list of global variable declarations, and an identifier naming the entry point of the program (the `main` function in C).

2.2 Dynamic Semantics

The dynamic semantics of Clight is specified using natural semantics, also known as big-step operational semantics. While the semantics of C is not deterministic (the evaluation order for expressions is not completely specified and compilers are free to choose between several orders), the semantics of Clight is completely deterministic and imposes a left-to-right evaluation order, consistent with the order implemented by our compiler. This choice simplifies greatly the semantics compared with, for instance, Norrish's semantics for C [10], which captures the non-determinism allowed by the ISO C specification. Our semantics can therefore be viewed as a refinement of (a subset of) the ISO C semantics, or of that of Norrish.

The semantics is defined by 7 judgements (relations):

$$
\begin{array}{rl}
G, E \vdash a, M \stackrel{l}{\Rightarrow} loc, M' & \text{(expressions in l-value position)} \\
G, E \vdash a, M \Rightarrow v, M' & \text{(expressions in r-value position)} \\
G, E \vdash a^?, M \Rightarrow v, M' & \text{(optional expressions)} \\
G, E \vdash a^*, M \Rightarrow v^*, M' & \text{(list of expressions)} \\
G, E \vdash s, M \Rightarrow out, M' & \text{(statements)} \\
G \vdash f(v^*), M \Rightarrow v, M' & \text{(function invocations)} \\
\vdash p \Rightarrow v & \text{(programs)}
\end{array}
$$

Each judgement relates a syntactic element (expression, statement, etc) and an initial memory state to the result of executing this syntactic element, as well as the final memory state at the end of execution. The various kinds of results, as well as the evaluation environments, are defined in figure 2.

For instance, executing an expression a in l-value position results in a memory location loc (a memory block reference and an offset within that block), while executing an expression a in r-value position results in the value v of the expression. Values range over 32-bit integers, 64-bit floats, memory locations (pointers), and an undefined value that represents for instance the value of uninitialized variables. 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. The invocation of a function f yields its return value v, and so does the execution of a program p.

Two evaluation environments, defined in figure 2, appear as parameters to the judgements. The local environment E maps local variables to references of memory blocks containing the values of these variables. These blocks are allocated

Values:

loc	$::= (b, n)$	location (byte offset n in block referenced by b)
v	$::= \texttt{Vint}(n)$	integer value
	$\mid \texttt{Vfloat}(f)$	floating-point value
	$\mid \texttt{Vptr}(loc)$	pointer value
	$\mid \texttt{Vundef}$	undefined value

Statement outcomes:

out	$::= \texttt{Out\_normal}$	go to the next statement
	$\mid \texttt{Out\_continue}$	go to the next iteration of the current loop
	$\mid \texttt{Out\_break}$	exit from the current loop
	$\mid \texttt{Out\_return}$	function exit
	$\mid \texttt{Out\_return}(v, \tau)$	function exit, returning the value v of type τ

Global environments:

G	$::= (id \mapsto b)$	map from global variables to block references
	$\times (b \mapsto fn)$	and map from references to function definitions

Local environments:

E	$::= id \mapsto b$	map from local variables to block references

Fig. 2. Values, outcomes, and evaluation environments

at function entry and freed at function return. The global environment G maps global variables and function names to memory references. It also maps some references (those corresponding to function pointers) to function definitions.

In the Coq specification, the 7 judgements of the dynamic semantics are encoded as mutually-inductive predicates. Each defining case of each predicate corresponds exactly to an inference rule in the conventional, on-paper presentation of natural semantics. We have one inference rule for each kind of expression and statement described in figure 1. We do not list all the inference rules by lack of space, but show some representative examples in figure 3.

The first two rules of figure 3 illustrate the evaluation of an expression in l-value position. A variable x evaluates to the location $(E(x), 0)$. If an expression a evaluates to a pointer value $\texttt{Vptr}(loc)$, then the location of the dereferencing expression $(\texttt{*}a)^\tau$ is loc.

Rule 3 evaluates an application of a binary operator op to expressions a_1 and a_2. Both sub-expressions are evaluated in sequence, and their values are combined with the `eval_binary_operation` 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. This is a partial function: it can be undefined if the types and the shapes of argument values are incompatible (e.g. a floating-point addition of two pointer values). In the Coq specification, `eval_binary_operation` is a total function returning optional values: either `None` in case of failure, or $\texttt{Some}(v)$, abbreviated as $\lfloor v \rfloor$, in case of success.

Rule 4 rule shows the evaluation of an l-value expression in a r-value context. The expression is evaluated to its location loc, with final memory state M'. The

Expressions in l-value position:

$$\frac{E(x) = b}{G, E \vdash x^\tau, M \stackrel{l}{\Rightarrow} (b, 0), M} \quad (1) \qquad \frac{G, E \vdash a, M \Rightarrow \mathtt{Vptr}(loc), M'}{G, E \vdash (*a)^\tau, M \stackrel{l}{\Rightarrow} loc, M'} \quad (2)$$

Expressions in r-value position:

$$\frac{G, E \vdash a_1^{\tau_1}, M \Rightarrow v_1, M_1 \quad G, E \vdash a_2^{\tau_2}, M_1 \Rightarrow v_2, M_2 \quad \mathtt{eval\_binary\_operation}(op, v_1, \tau_1, v_2, \tau_2) = \lfloor v \rfloor}{G, E \vdash (a_1^{\tau_1}\, op\, a_2^{\tau_2})^\tau, M \Rightarrow v, M_2} \quad (3)$$

$$\frac{G, E \vdash a^\tau, M \stackrel{l}{\Rightarrow} loc, M' \quad \mathtt{loadval}(\tau, M', loc) = \lfloor v \rfloor}{G, E \vdash a^\tau, M \Rightarrow v, M'} \quad (4)$$

$$\frac{G, E \vdash a^\tau, M \stackrel{l}{\Rightarrow} loc, M_1 \quad G, E \vdash b^\sigma, M_1 \Rightarrow v_1, M_2 \quad \mathtt{cast}(v_1, \sigma, \tau) = \lfloor v \rfloor \quad \mathtt{storeval}(\tau, M_2, loc, v) = \lfloor M_3 \rfloor}{G, E \vdash (a^\tau \ \mathtt{=}\ b^\sigma)^\tau, M \Rightarrow v, M_3} \quad (5)$$

Statements:

$$G, E \vdash \mathtt{break}, M \Rightarrow \mathtt{Out\_break}, M \quad (6)$$

$$\frac{G, E \vdash s_1, M \Rightarrow \mathtt{Out\_normal}, M_1 \quad G, E \vdash s_2, M_1 \Rightarrow out, M_2}{G, E \vdash (s_1; s_2), M \Rightarrow out, M_2} \quad (7)$$

$$\frac{G, E \vdash s_1, M \Rightarrow out, M' \quad out \neq \mathtt{Out\_normal}}{G, E \vdash (s_1; s_2), M \Rightarrow out, M'} \quad (8)$$

$$\frac{G, E \vdash a, M \Rightarrow v, M' \quad \mathtt{is\_false}(v)}{G, E \vdash (\mathtt{while}(a)\ s), M \Rightarrow \mathtt{Out\_normal}, M'} \quad (9)$$

$$\frac{G, E \vdash a, M \Rightarrow v, M_1 \quad \mathtt{is\_true}(v) \quad G, E \vdash s, M_1 \Rightarrow \mathtt{Out\_break}, M_2}{G, E \vdash (\mathtt{while}(a)\ s), M \Rightarrow \mathtt{Out\_normal}, M_2} \quad (10)$$

$$\frac{G, E \vdash a, M \Rightarrow v, M_1 \quad \mathtt{is\_true}(v) \quad G, E \vdash s, M_1 \Rightarrow out, M_2 \quad out \in \{\mathtt{Out\_normal}, \mathtt{Out\_continue}\} \quad G, E \vdash (\mathtt{while}(a)\ s), M_2 \Rightarrow out', M_3}{G, E \vdash (\mathtt{while}(a)\ s), M \Rightarrow out', M_3} \quad (11)$$

Fig. 3. Selected rules of the dynamic semantics of Clight

value at location *loc* in M' is fetched using the `loadval` function (see section 2.3) and returned.

Rule 5 evaluates an assignment expression. An assignment expression a^τ `=` b^σ evaluates the l-value a to a location *loc*, then the r-value b to a value v_1. This value is cast from its natural type σ to the expected type τ using the partial function `cast`. This function performs appropriate conversions, truncations and sign-extensions over integers and floats, and may fail for undefined casts. The result v of the cast is then stored in memory at location *loc*, resulting in the final memory state M_3, and returned as the value of the assignment expression.

The bottom group of rules in figure 3 are examples of statement executions. The execution of a break statement yields an `Out_break` outcome (rule 6). The execution of a sequence of two statements starts with the execution of the first statement, yielding an outcome that determines if the second statement must be executed or not (rules 7 and 8). Finally, rules 9–11 describe the execution of a `while` loop. Once the condition of a while loop is evaluated to a value v, the execution of the loop terminates normally if v is false. If v is true, the loop body is executed, yielding an outcome *out*. If *out* is `Out_break`, the loop terminates normally. If *out* is `Out_normal` or `Out_continue`, the whole loop is reexecuted in the memory state modified by the execution of the body.

2.3 The Memory Model of the Semantics

The memory model used in the dynamic semantics is described in [1]. It is a compromise between a low-level view of memory as an array of bytes and a high-level view as a mapping from abstract references to contents. In our model, the memory is arranged in independent blocks, identified by block references b. A memory state M maps references b to block contents, which are themselves mappings from byte offsets to values. Each block has a low bound $L(M, b)$ and a high bound $H(M, b)$, determined at allocation time and representing the interval of valid byte offsets within this block. This memory model guarantees separation properties between two distinct blocks, yet enables pointer arithmetic within a given block, as prescribed by the ISO C specification. The same memory model is common to the semantics of all intermediate languages of our certified compiler.

The memory model provides 4 basic operations:

`alloc`$(M, lo, hi) = (M', b)$
: Allocate a fresh block of bounds $[lo, hi)$. Returns extended memory M' and reference b to fresh block.

`free`$(M, b) = M'$
: Free (invalidate) the block b.

`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.

`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'.

The memory chunks κ appearing in `load` and `store` operations describe concisely the size, type and signedness of the memory quantities involved:

$$
\begin{array}{rll}
\kappa ::= & \texttt{Mint8signed} \mid \texttt{Mint8unsigned} & \\
\mid & \texttt{Mint16signed} \mid \texttt{Mint16unsigned} & \text{small integers} \\
\mid & \texttt{Mint32} & \text{integers and pointers} \\
\mid & \texttt{Mfloat32} \mid \texttt{Mfloat64} & \text{floats}
\end{array}
$$

In the semantics of C, those quantities are determined by the C types of the datum being addressed. The following $\mathcal{A}$ ("access mode") function mediates between C types and the corresponding memory chunks:

$$
\begin{aligned}
\mathcal{A}(\texttt{Tint}(\texttt{I8}, \texttt{Signed})) &= \texttt{By\_value}(\texttt{Mint8signed}) \\
\mathcal{A}(\texttt{Tint}(\texttt{I8}, \texttt{Unsigned})) &= \texttt{By\_value}(\texttt{Mint8unsigned}) \\
\mathcal{A}(\texttt{Tint}(\texttt{I16}, \texttt{Signed})) &= \texttt{By\_value}(\texttt{Mint16signed}) \\
\mathcal{A}(\texttt{Tint}(\texttt{I16}, \texttt{Unsigned})) &= \texttt{By\_value}(\texttt{Mint16unsigned}) \\
\mathcal{A}(\texttt{Tint}(\texttt{I32}, \_)) = \mathcal{A}(\texttt{Tpointer}(\_)) &= \texttt{By\_value}(\texttt{Mint32}) \\
\mathcal{A}(\texttt{Tarray}(\_, \_)) = \mathcal{A}(\texttt{Tfunction}(\_, \_)) &= \texttt{By\_reference} \\
\mathcal{A}(\texttt{Tvoid}) &= \texttt{By\_nothing}
\end{aligned}
$$

Integer, float and pointer types involve an actual memory `load` when accessed, as captured by the `By_value` cases. However, accesses to arrays and functions return the location of the array or function, without any `load`; this is indicated by the `By_reference` access mode. Finally, expressions of type `void` cannot be accessed at all. This is reflected in the definitions of the `loadval` and `storeval` functions used in the dynamic semantics:

$$
\begin{aligned}
\texttt{loadval}(\tau, M, (b, n)) &= \texttt{load}(\kappa, M, b, n) && \text{if } \mathcal{A}(\tau) = \texttt{By\_value}(\kappa) \\
\texttt{loadval}(\tau, M, (b, n)) &= \lfloor b, n \rfloor && \text{if } \mathcal{A}(\tau) = \texttt{By\_reference} \\
\texttt{loadval}(\tau, M, (b, n)) &= \texttt{None} && \text{if } \mathcal{A}(\tau) = \texttt{By\_nothing} \\
\texttt{storeval}(\tau, M, (b, n), v) &= \texttt{store}(\kappa, M, b, n, v) && \text{if } \mathcal{A}(\tau) = \texttt{By\_value}(\kappa) \\
\texttt{storeval}(\tau, M, (b, n), v) &= \texttt{None} && \text{otherwise}
\end{aligned}
$$

2.4 Static Semantics (Typing Rules)

We have also formalized in Coq typing rules and a type checking algorithm for Clight. The algorithm is presented as a function from abstract syntax trees without type annotations to the abstract syntax trees with type annotations over expressions given in figure 1. We omit the typing rules by lack of space. Note that the dynamic semantics are defined for arbitrarily annotated expressions, not just well-typed expressions; however, the semantics can get stuck or produce results that disagree with ISO C when given an incorrectly-annotated expression. The translation scheme presented in section 3 demands well-typed programs and may fail to preserve semantics otherwise.

3 Translation from Clight to Cminor

3.1 Overview of Cminor

The Cminor language is the target language of our front-end compiler for C and the input language for our certified back-end. We now give a short overview of Cminor; see [8] for a more detailed description, and [7] for a complete formal specification.

Cminor is a low-level imperative language, structured like our subset of C into expressions, statements, and functions. We summarize the main differences with Clight. First, 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. Arithmetic is always performed over 32-bit integers and 64-bit floats; explicit truncation and sign-extension operators are provided to implement smaller integral types. Finally, the combined arithmetic-with-assignment operators of C (`+=`, `++`, etc) are not provided. For instance, the C expression `i += f` where `i` is of type `int` and `f` of type `double` is expressed as `i = intoffloat(floatofint(i) +`$_f$ `f)`.

Address computations are explicit, as well as individual load and store operations. For instance, the C expression `a[i]` where `a` is a pointer to `int` is expressed as `load(int32, a +`$_i$ `i *`$_i$ `4)`, making explicit the memory chunk being addressed (`int32`) as well as the computation of the address.

At the level of statements, Cminor has only 4 control structures: if-then-else conditionals, infinite loops, `block-exit`, and early return. The `exit` n statement terminates the $(n+1)$ enclosing `block` statements. These structures are lower-level than those of C, but suffice to implement all reducible flow graphs.

Within Cminor functions, local variables can only hold scalar values (integers, pointers, floats) and they do not reside in memory, making it impossible to take 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.[1]

The semantics of Cminor is defined in big-step operational style and resembles that of Clight. The following evaluation judgements are defined in [7]:

$$\begin{aligned} G, sp, L \vdash a, E, M &\to v, E', M' && \text{(expressions)} \\ G, sp, L \vdash a^*, E, M &\to v^*, E', M' && \text{(expression lists)} \\ G, sp \vdash s, E, M &\to out, E', M' && \text{(statements)} \\ G \vdash fn(v^*), M &\to v, M' && \text{(function calls)} \\ \vdash prog &\to v && \text{(whole programs)} \end{aligned}$$

[1] While suboptimal in terms of performance of generated code, this systematic stack allocation of local variables whose addresses are taken is common practice for moderately optimizing C compilers such as `gcc` versions 2 and 3.

The main difference with the semantics of Clight is that the local evaluation environment E maps local variables to their values, instead of their memory addresses; consequently, E is modified during evaluation of expressions and statements. Additional parameters are sp, the reference to the stack block for the current function, and L, the environment giving values to variables `let`-bound within expressions.

3.2 Overview of the Translation

The translation from our subset of Clight to Cminor performs three basic tasks:

- Resolution of operator overloading and explication 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, both explicit in the source and implicit in the semantics of Clight; address computations are generated based on the types of array elements and pointer targets; and appropriate memory chunks are selected for every memory access.
- Translation of `while`, `do...while` and `for` loops into infinite loops with blocks and early exits. The statements `break` and `continue` are translated as appropriate `exit` constructs, as shown in figure 4.
- 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).[2]

The translation is specified as Coq functions from Clight abstract syntax to Cminor abstract syntax, defined by structural recursion. From these Coq functions, executable Caml code can be mechanically generated using the Coq extraction facility, making the specification directly executable. Several translation functions are defined: $\mathcal{L}$ and $\mathcal{R}$ for expressions in l-value and r-value position, respectively; $\mathcal{S}$ for statements; and $\mathcal{F}$ for functions. Some representative cases of the definitions of these functions are shown in figure 4, giving the general flavor of the translation.

The translation can fail when given invalid Clight source code, e.g. containing an assignment between arrays. To enable error reporting, the translation functions return option types: either `None` denoting an error, or $\lfloor x \rfloor$ denoting successful translation with result x. Systematic propagation of errors is achieved using a monadic programming style (the `bind` combinator of the error monad),

[2] It would be semantically correct to stack-allocate all local variables, like the C0 verified compiler does [6, 12]. However, keeping scalar local variables in Cminor local variables as much as possible enables the back-end to generate much more efficient machine code.

Casts ($\mathcal{C}^{\sigma}_{\tau}(e)$ casts e from type τ to type σ):

$$\mathcal{C}^{\sigma}_{\tau}(e) = \mathcal{C}_2(\mathcal{C}_1(e,\tau,\sigma),\sigma)$$

$$\mathcal{C}_1(e,\tau,\sigma) = \begin{cases} \texttt{floatofint}(e), & \text{if } \tau = \texttt{Tint}(\_,\texttt{Signed}) \text{ and } \sigma = \texttt{Tfloat}(\_); \\ \texttt{floatofintu}(e), & \text{if } \tau = \texttt{Tint}(\_,\texttt{Unsigned}) \text{ and } \sigma = \texttt{Tfloat}(\_); \\ \texttt{intoffloat}(e), & \text{if } \tau = \texttt{Tfloat}(\_) \text{ and } \sigma = \texttt{Tint}(\_,\_); \\ e & \text{otherwise} \end{cases}$$

$$\mathcal{C}_2(e,\sigma) = \begin{cases} \texttt{cast8signed}(e), & \text{if } \sigma = \texttt{Tint}(\texttt{I8},\texttt{Signed}); \\ \texttt{cast8unsigned}(e), & \text{if } \sigma = \texttt{Tint}(\texttt{I8},\texttt{Unsigned}); \\ \texttt{cast16signed}(e), & \text{if } \sigma = \texttt{Tint}(\texttt{I16},\texttt{Signed}); \\ \texttt{cast16unsigned}(e), & \text{if } \sigma = \texttt{Tint}(\texttt{I16},\texttt{Unsigned}); \\ \texttt{singleoffloat}(e), & \text{if } \sigma = \texttt{Tfloat}(\texttt{F32}); \\ e & \text{otherwise} \end{cases}$$

Expressions in l-value position:

$$\mathcal{L}_\gamma(x) = \texttt{addrstack}(\delta) \quad \text{if } \gamma(x) = \texttt{Stack}(\delta)$$
$$\mathcal{L}_\gamma(x) = \texttt{addrglobal}(x) \quad \text{if } \gamma(x) = \texttt{Global}$$
$$\mathcal{L}_\gamma(\texttt{*}e) = \mathcal{R}_\gamma(e)$$
$$\mathcal{L}_\gamma(e_1[e_2]) = \mathcal{R}_\gamma(e_1 \texttt{ + } e_2)$$

Expressions in r-value position:

$$\mathcal{R}_\gamma(x) = x \quad \text{if } \gamma(x) = \texttt{Local}$$
$$\mathcal{R}_\gamma(e^{\tau}) = \texttt{load}(\kappa, \mathcal{L}_\gamma(e^{\tau})) \quad \text{if } \mathcal{L}_\gamma(e) \text{ is defined and } \mathcal{A}(\tau) = \texttt{By\_value}(\kappa)$$
$$\mathcal{R}_\gamma(e^{\tau}) = \mathcal{L}_\gamma(e^{\tau}) \quad \text{if } \mathcal{L}_\gamma(e) \text{ is defined and } \mathcal{A}(\tau) = \texttt{By\_reference}$$
$$\mathcal{R}_\gamma(x^{\tau} \texttt{ = } e^{\sigma}) = x \texttt{ = } \mathcal{C}^{\tau}_{\sigma}(\mathcal{R}(e^{\sigma})) \quad \text{if } \gamma(x) = \texttt{Local}$$
$$\mathcal{R}_\gamma(e_1^{\tau} \texttt{ = } e_2^{\sigma}) = \texttt{store}(\kappa, \mathcal{L}_\gamma(e_1^{\tau}), \mathcal{C}^{\tau}_{\sigma}(\mathcal{R}(e_2^{\sigma}))) \quad \text{if } \mathcal{A}(\tau) = \texttt{By\_value}(\kappa)$$
$$\mathcal{R}_\gamma(\texttt{\&}e) = \mathcal{L}_\gamma(e)$$
$$\mathcal{R}_\gamma(e_1^{\tau} \texttt{ + } e_2^{\sigma}) = \mathcal{R}_\gamma(e_1^{\tau}) \texttt{ +}_i \mathcal{R}_\gamma(e_2^{\sigma}) \quad \text{if } \tau \text{ and } \sigma \text{ are integer types}$$
$$\mathcal{R}_\gamma(e_1^{\tau} \texttt{ + } e_2^{\sigma}) = \mathcal{C}^{\texttt{double}}_{\tau}(\mathcal{R}_\gamma(e_1^{\tau})) \texttt{ +}_f \mathcal{C}^{\texttt{double}}_{\sigma}(\mathcal{R}_\gamma(e_2^{\sigma})) \quad \text{if } \tau \text{ or } \sigma \text{ are float types}$$
$$\mathcal{R}_\gamma(e_1^{\tau} \texttt{ + } e_2^{\sigma}) = \mathcal{R}_\gamma(e_1^{\tau}) \texttt{ +}_i \mathcal{R}_\gamma(e_2^{\sigma}) \texttt{ *}_i \texttt{ sizeof}(\rho) \quad \text{if } \tau \text{ is a pointer or array of } \rho$$

Statements:

$$\mathcal{S}_\gamma(\texttt{while}(e)\ s) = \texttt{block\{ loop\{ if } (\texttt{!}\mathcal{R}_\gamma(e)) \texttt{ exit 0; block\{ } \mathcal{S}_\gamma(s) \texttt{ \}\}\}}$$
$$\mathcal{S}_\gamma(\texttt{do } s \texttt{ while}(e)) = \texttt{block\{ loop\{ block\{ } \mathcal{S}_\gamma(s) \texttt{ \}; if } (\texttt{!}\mathcal{R}_\gamma(e)) \texttt{ exit 0 \}\}}$$
$$\mathcal{S}_\gamma(\texttt{for}(e_1;e_2;e_3)\ s) = \mathcal{R}_\gamma(e_1); \texttt{ block\{ loop\{ if } (\texttt{!}\mathcal{R}_\gamma(e_2)) \texttt{ exit 0; block\{ } \mathcal{S}_\gamma(s) \texttt{ \}; } \mathcal{R}_\gamma(e_3) \texttt{ \}\}}$$
$$\mathcal{S}_\gamma(\texttt{break}) = \texttt{exit 1}$$
$$\mathcal{S}_\gamma(\texttt{continue}) = \texttt{exit 0}$$

Fig. 4. Selected translation rules

as customary in purely functional programming. This monadic "plumbing" is omitted in figure 4 for simplicity.

Most translation functions are parameterized by a translation environment γ reflecting the placement of Clight variables. It maps every variable x to either `Local` (denoting the Cminor local variable named x), `Stack`(δ) (denoting a sub-area of the Cminor stack block at offset δ), or `Global` (denoting the address of the Cminor global symbol named x). This environment is constructed at the beginning of the translation of a Clight function. The function body is scanned for occurrences of `&x` (taking the address of a variable). Local variables that are not scalar or whose address is taken are assigned `Stack`(δ) locations, with δ chosen so that distinct variables map to non-overlapping areas of the stack block. Other local variables are set to `Local`, and global variables to `Global`.

4 Proof of Correctness of the Translation

4.1 Relating Memory States

To prove the correctness of the translation, the major difficulty is to relate the memory states occurring during the execution of the Clight source code and that of the generated Cminor code. The semantics of Clight allocates a distinct block for every local variable at function entry. Some of those blocks (those for scalar variables whose address is not taken) have no correspondence in the Cminor memory state; others become sub-block of the Cminor stack block for the function.

To account for these differences in allocation patterns between the source and target code, we introduce the notion of *memory injections*. A memory injection α is a function from Clight block references b to either `None`, meaning that this block has no counterpart in the Cminor memory state, or $\lfloor b', \delta \rfloor$, meaning that the block b of the Clight memory state corresponds to a sub-block of block b' at offset δ in the Cminor memory state.

A memory injection α defines a relation between Clight values v and Cminor values v', written $\alpha \vdash v \approx v'$ and defined as follows:

$$\alpha \vdash \texttt{Vint}(n) \approx \texttt{Vint}(n) \qquad \alpha \vdash \texttt{Vfloat}(n) \approx \texttt{Vfloat}(n) \qquad \alpha \vdash \texttt{Vundef} \approx v$$

$$\frac{\alpha(b) = \lfloor b', \delta \rfloor \qquad i' = i + \delta \pmod{2^{32}}}{\alpha \vdash \texttt{Vptr}(b, i) \approx \texttt{Vptr}(b', i')}$$

This relation captures the relocation of pointer values implied by α. It also enables `Vundef` Clight values to become more defined Cminor values during the translation, in keeping with the general idea that compilation can particularize some undefined behaviors.

The memory injection α also defines a relation between Clight and Cminor memory states, written $\alpha \vdash M \approx M'$, consisting of the conjunction of the following conditions:

- Matching of block contents: if $\alpha(b) = \lfloor b', \delta \rfloor$ and $L(M,b) \leq i < H(M,b)$, then $L(M',b') \leq i + \delta < H(M',b')$ and $\alpha \vdash v \approx v'$ where v is the contents of block b at offset i in M and v' the contents of b' at offset i' in M'.
- No overlap: if $\alpha(b_1) = \lfloor b'_1, \delta_1 \rfloor$ and $\alpha(b_2) = \lfloor b'_2, \delta_2 \rfloor$ and $b_1 \neq b_2$, then either $b'_1 \neq b'_2$, or the intervals $[L(M,b_1) + \delta_1, H(M,b_1) + \delta_1)$ and $[L(M,b_2) + \delta_2, H(M,b_2) + \delta_2)$ are disjoint.
- Fresh blocks: $\alpha(b) = \texttt{None}$ for all blocks b not yet allocated in M.

The memory injection relations have nice commutation properties with respect to the basic operations of the memory model. For instance:

- Commutation of loads: if $\alpha \vdash M \approx M'$ and $\alpha \vdash \texttt{Vptr}(b,i) \approx \texttt{Vptr}(b',i')$ and $\texttt{load}(\kappa, M, b, i) = \lfloor v \rfloor$, there exists v' such that $\texttt{load}(\kappa, M', b', i') = \lfloor v' \rfloor$ and $\alpha \vdash v \approx v'$.
- Commutation of stores to mapped blocks: if $\alpha \vdash M \approx M'$ and $\alpha \vdash \texttt{Vptr}(b,i) \approx \texttt{Vptr}(b',i')$ and $\alpha \vdash v \approx v'$ and $\texttt{store}(\kappa, M, b, i, v) = \lfloor M_1 \rfloor$, there exists M'_1 such that $\texttt{store}(\kappa, M', b', i', v') = \lfloor M'_1 \rfloor$ and $\alpha \vdash M_1 \approx M'_1$.
- Invariance by stores to unmapped blocks: if $\alpha \vdash M \approx M'$ and $\alpha(b) = \texttt{None}$ and $\texttt{store}(\kappa, M, b, i, v) = \lfloor M_1 \rfloor$, then $\alpha \vdash M_1 \approx M'$.

To enable the memory injection α to grow incrementally as new blocks are allocated during execution, we define the relation $\alpha' \geq \alpha$ (read: α' extends α) by $\forall b,\ \alpha'(b) = \alpha(b) \vee \alpha(b) = \texttt{None}$. The injection relations are preserved by extension of α. For instance, if $\alpha \vdash M \approx M'$, then $\alpha' \vdash M \approx M'$ for all α' such that $\alpha' \geq \alpha$.

4.2 Relating Execution Environments

Execution environments differ in structure between Clight and Cminor: the Clight environment E maps local variables to references of blocks containing the values of the variables, while in Cminor the environment E' for local variables map them directly to values. We define a matching relation $\mathit{EnvMatch}(\gamma, \alpha, E, M, E', sp)$ between a Clight environment E and memory state M and a Cminor environment E' and reference to a stack block sp as follows:

- For all variables x of type τ, if $\gamma(x) = \texttt{Local}$, then $\alpha(E(x)) = \texttt{None}$ and there exists v such that $\texttt{load}(\kappa(\tau), M, E(x), 0) = \lfloor v \rfloor$ and $\alpha \vdash v \approx E'(x)$.
- For all variables x of type τ, if $\gamma(x) = \texttt{Stack}(\delta)$, then $\alpha \vdash \texttt{Vptr}(E(x), 0) \approx \texttt{Vptr}(sp, \delta)$.
- For all $x \neq y$, we have $E(x) \neq E(y)$.
- If $\alpha(b) = \lfloor sp, \delta \rfloor$ for some block b and offset δ, then b is in the range of E.

The first two conditions express the preservation of the values of local variables during compilation. The last two rule out unwanted sharing between environment blocks and their images through α.

At any point during execution, several function calls may be active and we need to ensure matching between the environments of each call. For this,

we introduce abstract call stacks, which are lists of 4-tuples (γ, E, E', sp) and record the environments of all active functions. A call stack cs is globally consistent with respect to C memory state M and memory injection α, written $\mathit{CallInv}(\alpha, M, cs)$, if $\mathit{EnvMatch}(\gamma, \alpha, E, M, E', sp)$ holds for all elements (γ, E, E', sp) of cs. Additional conditions, omitted for brevity, enforce separation between Clight environments E and between Cminor stack blocks sp belonging to different function activations in cs.

4.3 Proof by Simulation

The proof of semantic preservation for the translation proceeds by induction over the Clight evaluation derivation and case analysis on the last evaluation rule used. The proof shows that, assuming suitable consistency conditions over the abstract call stack, the generated Cminor expressions and statements evaluate in ways that simulate the evaluation of the corresponding Clight expressions and statements.

We give a slightly simplified version of the simulation properties shown by induction over the Clight evaluation derivation. Let G' be the global Cminor environment obtained by translating all function definitions in the global Clight environment G. Assume $\mathit{CallInv}(\alpha, M, (\gamma, E, E', sp).cs)$ and $\alpha \vdash M \approx M'$. Then there exists a Cminor environment E'_1, a Cminor memory state M'_1 and a memory injection $\alpha_1 \geq \alpha$ such that

- (R-values) If $G, E \vdash a, M \Rightarrow v, M_1$, there exists v' such that $G', sp, L \vdash \mathcal{R}_\gamma(a), E', M' \rightarrow v', E'_1, M'_1$ and $\alpha_1 \vdash v \approx v'$.
- (L-values) If $G, E \vdash a, M \stackrel{l}{\Rightarrow} loc, M_1$, there exists v' such that $G', sp, L \vdash \mathcal{L}_\gamma(a), E', M' \rightarrow v', E'_1, M'_1$ and $\alpha_1 \vdash \mathtt{Vptr}(loc) \approx v'$.
- (Statements) If $G, E \vdash s, M \Rightarrow out, M_1$ and τ_r is the return type of the function, there exists out' such that $G', sp \vdash \mathcal{S}_\gamma(s), E', M' \rightarrow out', E'_1, M'_1$ and $\alpha_1, \tau_r \vdash out \approx out'$.

Moreover, the final Clight and Cminor states satisfy $\alpha_1 \vdash M_1 \approx M'_1$ and $\mathit{CallInv}(\alpha_1, M_1, (\gamma, E, E'_1, sp).cs)$.

In the case of statements, the relation between Clight and Cminor outcomes is defined as follows:

$$\alpha, \tau_r \vdash \mathtt{Out\_normal} \approx \mathtt{Out\_normal} \qquad \alpha, \tau_r \vdash \mathtt{Out\_continue} \approx \mathtt{Out\_exit}(0)$$
$$\alpha, \tau_r \vdash \mathtt{Out\_break} \approx \mathtt{Out\_exit}(1) \qquad \alpha, \tau_r \vdash \mathtt{Out\_return} \approx \mathtt{Out\_return}$$

$$\frac{\alpha \vdash \mathtt{cast}(v, \tau, \tau_r) \approx v'}{\alpha, \tau_r \vdash \mathtt{Out\_return}(v, \tau) \approx \mathtt{Out\_return}(v')}$$

In addition to the outer induction over the Clight evaluation derivation, the proofs proceed by copious case analysis, over the placement $\gamma(x)$ for accesses to variables x, and over the types of the operands for applications of overloaded operators. As a corollary of the simulation properties, we obtain the correctness theorem for the translation:

Theorem 1. *Assume the Clight program p is well-typed and translates without errors to a Cminor program p'. If $\vdash p \Rightarrow v$, and if v is an integer or float value, then $\vdash p' \rightarrow v$.*

This semantic preservation theorem applies only to terminating programs. Our choice of big-step operational semantics prevents us from reasoning over non-terminating executions.

The whole proof represents approximately 6000 lines of Coq statements and proof scripts, including 1000 lines (40 lemmas) for the properties of memory injections, 1400 lines (54 lemmas) for environment matching and the call stack invariant, 1400 lines (50 lemmas) for the translations of type-dependent operators and memory accesses, and 2000 lines (51 lemmas, one per Clight evaluation rule) for the final inductive proof of simulation. By comparison, the source code of the Clight to Cminor translator is 800 lines of Coq function definitions. The proof is therefore 7.5 times bigger than the code it proves. The whole development (design and semantics of Clight; development of the translator; proof of its correctness) took approximately 8 person.months.

5 Related Work

Several formal semantics of C-like languages have been defined. Norrish [10] gives a small-step operational semantics, expressed using the HOL theorem prover, for a subset of C comparable to our Clight. His semantics captures exactly the non-determinism (partially unspecified evaluation order) allowed by the ISO C specification, making it significantly more complex than our deterministic semantics. Papaspyrou [11] addresses non-determinism as well, but using denotational semantics with monads. Abstract state machines have been used to give on-paper semantics for C [4, 9] and more recently for C# [3].

Many correctness proofs of program transformations have been published, both on paper and machine-checked using proof assistants; see [2] for a survey. A representative example is [5], where a non-optimizing byte-code compiler from a subset of Java to a subset of the Java Virtual Machine is verified using Isabelle/HOL. Most of these correctness proofs apply to source languages that are either smaller or semantically cleaner than C.

The work that is closest to ours is part of the Verisoft project [6, 12]. Using Isabelle/HOL, they formalize the semantics of C0 (a subset of the C language) and a compiler from C0 down to DLX assembly code. C0 is a type-safe subset of C, close to Pascal, and significantly smaller than our Clight: there is no pointer arithmetic, nor side effects, nor premature execution of statements and there exists only a single integer type, thus avoiding operator overloading. They provide both a big step semantics and a small step semantics for C0, the latter enabling reasoning about non-terminating and concurrent executions, unlike our big-step semantics. Their C0 compiler is a single pass compiler that generates unoptimized machine code. It is more complex than our translation from Clight to Cminor, but considerably simpler than our whole certified compiler.

6 Concluding Remarks

The C language is not pretty; this shows up in the relative complexity of our formal semantics and translation scheme. However, this complexity remains manageable with the tools (the Coq proof assistant) and the methodology (big-step semantics; simulation arguments; extraction of an executable compiler from its functional Coq specification) that we used.

Future work includes 1- handling a larger subset of C, especially `struct` types; and 2- evaluating the usability of the semantics for program proof and static analysis purposes. In particular, it would be interesting to develop axiomatic semantics (probably based on separation logic) for Clight and validate them against our operational semantics.

References

1. S. Blazy and X. Leroy. Formal verification of a memory model for C-like imperative languages. In *Proc. of Int. Conf. on Formal Engineering Methods (ICFEM)*, volume 3785 of *LNCS*, pages 280–299, Manchester, UK, Nov. 2005. Springer-Verlag.
2. M. A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, 2003.
3. E. Börger, N. Fruja, V.Gervasi, and R. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 336(2-3):235–284, 2005.
4. Y. Gurevich and J. Huggins. The semantics of the C programming language. In *Proc. of CSL'92 (Computer Science Logic)*, volume 702 of *LNCS*, pages 274–308. Springer Verlag, 1993.
5. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Mar. 2004. To appear in ACM TOPLAS.
6. D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler. In *Proc. Conf. on Software Engineering and Formal Methods (SEFM)*, pages 2–11, Koblenz, Germany, Sept. 2005. IEEE Computer Society Press.
7. X. Leroy. The Compcert certified compiler back-end – commented Coq development. Available on-line at `http://cristal.inria.fr/~xleroy`, 2006.
8. X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proc. Symp. Principles Of Programming Languages (POPL)*, pages 42–54, Charleston, USA, Jan. 2006. ACM Press.
9. V. Nepomniaschy, I. Anureev, and A. Promsky. Verification-oriented language C-light and its structural operational semantics. In *Ershov Memorial Conference*, pages 103–111, 2003.
10. M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, Dec. 1998.
11. N. Papaspyrou. *A formal semantics for the C programming language*. PhD thesis, National Technical University of Athens, Feb. 1998.
12. M. Strecker. Compiler verification for C0. Technical report, Université Paul Sabatier, Toulouse, Apr. 2005.