Certified code generation from CPS to C

Olivier Savary Bélanger
Princeton University
olivier@galois.com

Matthew Z. Weaver
Princeton University
mzw@cs.princeton.edu

Andrew W. Appel
Princeton University
appel@princeton.edu

Abstract

CertiCoq is a verified-in-Coq extracter/compiler from Coq's Gallina language through CompCert C to assembly language, written as a functional program in Coq. Here we describe the implementation and Coq verification of its code generator, which translates from a continuation-passing style (CPS) intermediate language into CompCert Clight. We show how invariants over our CPS IR facilitate the generation of well behaved, efficient C code. A key point is our proved-correct interface to an external proved-correct (by other authors) generational garbage collector written in C. The semantics of C can be quite intricate, as can the design of a compiler-to-g.c. interface for finding roots—but the design of our CPS intermediate language facilitates a (relatively) simple implementation and correctness proof. Our measurements show that both the code generator and the generated code have good performance. Via CompCert, we have proved-correct back ends for several instruction-set architectures: x86-32, x86-64, ARM-32, ARM-64, RISC-V, and Power-PC.

CCS Concepts • **Software and its engineering → Formal software verification**; *Compilers*; *Correctness*; *Functional languages*; • **Theory of computation** → *Program verification.*

ACM Reference Format:
Olivier Savary Bélanger, Matthew Z. Weaver, and Andrew W. Appel. 2019. Certified code generation from CPS to C. In *Proceedings of (October 2019).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

If you prove your functional program in correct in Coq, then why entrust it to an unverified extraction/compilation pipeline? Neither Coq's extraction-to-Ocaml, nor the Ocaml compiler, nor Ocaml's runtime system is proved correct.

CertiCoq [1] is a verified-in-Coq extracter/compiler from Coq's *Gallina* functional language to assembly language, via CompCert C. The source program is extracted from the Coq kernel (its AST is reified from Ocaml datatype constructors to Coq datatype constructors) by MetaCoq [2]; this is the only phase that cannot be proved correct but it does no more than transliteration. After that (and with proofs!), we erase proofs, types, and related computationally irrelevant content [3]; constructors are eta-expanded so each constructor application is fully applied to all its arguments; the program is combined with its environment by let-binding all imported definitions, resulting in an untyped program in a simple de Bruijn functional language with inductive constructors. Then we convert to continuation-passing style (CPS) using a named representation, and we apply optimizations such as uncurrying [4], shrink-reduction [5], and lambda lifting; we closure-convert [6] into CPS terms in which all functions are closed (except for references to other closed functions in the global scope).

As we were not interested in verifying register allocation or supporting multiple back-ends for many target architectures, we translate our CPS into CompCert C light, and use CompCert as our verified register allocator and back-end code generator.

CertiCoq has a high-performance generational garbage collector, written in C and proved correct in Coq [7]. When one connects any compiler to a garbage collector, one must make an interface by which the compiler calls the collector, indicating where to find all the roots of the data graph—that is, the live local variables; and (when copying collection is used) one must be prepared for all variables of the program to be modified to point to their new locations.

Contributions. In this paper we describe the proof of correctness of our CPS-to-C translation phase: how we relate the operational semantics of CPS to the operational semantics of CompCert C, and how we reason about the graph transformation inherent in the call to the collector. These proofs are connected to proofs of the front-end phases via the CPS syntax and semantics, and to the CompCert back-end via the CompCert Clight syntax and semantics.

The artifact accompanying this paper has the code generator and its correctness proofs (not the rest of CertiCoq) plus the imported components of CompCert (that is, files leading up to the AST and operational semantics of CompCert Clight). We use no axioms, but CompCert Clight uses some; see the README.

2 The CPS Intermediate Representation

Continuation-passing style (CPS) is a restriction over a functional language where all calls are tail calls [8]. This is useful in intermediate languages for functional-language compilers, as it simplifies optimizations over the intermediate language and code generation in the presence of a garbage collector.

Continuation-passing style makes the control flow of programs explicit, making it easier to reason formally about order of execution and to define transformations working over different modes of execution.

$$
\begin{array}{llll}
\text{(Function Def'n)} & fd & ::= & f\ (\vec{x})\ =\ e \\
\text{(Branch)} & b & ::= & c \Rightarrow e \\
\text{(Expression)} & e & ::= & \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ e \\
 & & & \mid \mathsf{let}\ x = \mathsf{Prim}\ p\ \vec{y}\ \mathsf{in}\ e \\
 & & & \mid \mathsf{let}\ x = \mathsf{Proj}_n\ y\ \mathsf{in}\ e \\
 & & & \mid \mathsf{App}\ x\ \vec{y} \\
 & & & \mid \mathsf{let}\ \vec{fd}\ \mathsf{in}\ e \\
 & & & \mid \mathsf{match}\ x\ \mathsf{with}\ \vec{b} \\
 & & & \mid \mathsf{halt}\ x \\
\text{(Value)} & v & ::= & (c,\ \vec{v}) \mid (\rho, \vec{fd}, x) \\
\text{(Environment)} & \rho & ::= & \cdot \mid \rho, x \mapsto v
\end{array}
$$

Figure 1. Syntax of the CPS Language (L6)

L6 is a continuation-passing-style functional language with mutually recursive functions and pattern-matching. Figure 1 shows its syntax. The term "let x = Con c $\vec{y}$ in e" binds the constructor c applied to arguments $\vec{y}$ to variable x in expression e. The term "let x = Prim p $\vec{y}$ in e" binds the result of the primitive operator p on arguments $\vec{y}$ to variable x in expression e. The term "let x = Proj$_n$ y in e" binds the nth projection of y to variable x in in expression e. The term "App x $\vec{y}$" applies function x to arguments $\vec{y}$. The term "match x with $\vec{b}$" matches the constructor c of x with the branch $(c \Rightarrow e) \in \vec{b}$ labeled by c; the constructors in the branches of $\vec{b}$ must be distinct. "halt x" terminates computation by returning the value bound to x.

Although our code generator is a pure functional program, we still want it to be *efficient*, no worse than $O(n \log n)$. We represent variables using globally unique positive binary numbers. Therefore a global map from names to (some sort of) properties can refer to any binding point in the program, so that we can, for example, tabulate information about the provenance and, when possible, the name of the variable in the original program. Using positive binary numbers as identifiers allows us to implement lookup tables as *binary tries* with logarithmic access time. Globally unique variable names also allow substitution without fear of variable capture (although global uniqueness is not preserved under substitution, as it may duplicate portions of terms containing binders).

$$
\frac{\rho(x) = c\ \vec{w} \quad (c \Rightarrow e) \in \vec{b} \quad \rho \vdash e \Downarrow_k v}{\rho \vdash \mathsf{match}\ x\ \mathsf{with}\ \vec{b} \Downarrow_k v}\ \text{E\_MATCH}
$$

$$
\frac{\rho(y) = c\ \vec{w} \quad \rho; x \mapsto w_n \vdash e \Downarrow_k v}{\rho \vdash \mathsf{let}\ x = \mathsf{Proj}_n\ y\ \mathsf{in}\ e \Downarrow_k v}\ \text{E\_PROJ}
$$

$$
\frac{\rho(f) = (\rho',\ \vec{fd},\ f) \quad (f\ (\vec{x})\ =\ e) \in \vec{fd} \quad \rho'; f_i \mapsto (\rho', \vec{fd}, f_i); \vec{x} \mapsto \rho(\vec{y}) \vdash e \Downarrow_k v}{\rho \vdash \mathsf{App}\ f\ \vec{y} \Downarrow_{k+1} v}\ \text{E\_APP}
$$

$$
\frac{\forall_{y_i \in \vec{y}}, \rho(y_i) = w_i \quad \rho; x \mapsto (c, \vec{w}) \vdash e \Downarrow_k v}{\rho \vdash \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ e \Downarrow_k v}\ \text{E\_CONSTR}
$$

$$
\frac{\forall_{y_i \in \vec{y}}, \rho(y_i) = w_i \quad f\ \vec{w} = w \quad \rho; x \mapsto w \vdash e \Downarrow_k v}{\rho \vdash \mathsf{let}\ x = \mathsf{Prim}\ f\ \vec{y}\ \mathsf{in}\ e \Downarrow_k v}\ \text{E\_PRIM}
$$

$$
\frac{\rho; f_1 \mapsto (\rho, \vec{fd}, f_1); \ldots; f_n \mapsto (\rho, \vec{fd}, f_n) \vdash e \Downarrow_k v \quad \text{where names}(\vec{fd}) = \{f_1, \ldots, f_n\}}{\rho \vdash \mathsf{let}\ \vec{fd}\ \mathsf{in}\ e \Downarrow_k v}\ \text{E\_FUN}
$$

$$
\frac{\rho(x) = v}{\rho \vdash \mathsf{halt}\ x \Downarrow_k v}\ \text{E\_HALT}
$$

Figure 2. Evaluation rules of the L6 CPS language

3 The Semantics of our CPS IR

The semantics of our object language is given through a big-step,[1] environment-based judgment $\rho \vdash e \Downarrow_k v$ evaluating expressions e in environment ρ into value v in at most k continuation-calls. We will sometimes omit the argument k and just write $\rho \vdash e \Downarrow v$ when the cost is inconsequential.

The environment ρ maps variables to values (seen in Figure 1): either constructed values $(c, \vec{v})$ or function-values $(\rho, \vec{fd}, x)$ where $\vec{fd}$ is a block of mutually recursive function definitions and x is the name of a function in the block. In fact, after closure conversion (that is, in any input to our code generator), a function value's ρ is always empty (because an environment is explicitly represented as a data structure $(c, \vec{v})$ passed as an extra argument to each function) and there is a single top-level block of functions (because there's no need for function nesting if all functions are closed).

Of course, *inside* the execution of a function, as variables are let-bound or parameter-bound, the *current* environment ρ is nontrivial in the evaluation judgment $\rho \vdash e \Downarrow v$.

Figure 2 shows the evaluation rules. Pattern-matching is broken into two operations. First, our case constructor

[1] We can use big-step semantics because all Gallina functions terminate.

"match x with $(c_1 \Rightarrow e_1, ..., c_n \Rightarrow e_n)$" determines which pattern c_i the construction bound to variable x matches, and proceeds to evaluate e_i, as seen in E_MATCH. Then, projections "let x_1 = Proj_1 x in...let x_m = Proj_m x in" are used to bind variables to the m arguments of c_i which will be replaced by the right values when evaluated as shown in rule E_PROJ.

ML's and Haskell's syntax and type systems connect case matching with projection, so that the programmer cannot mistakenly project a field from the wrong constructor. We separate projections from cases because it makes the operational semantics simpler, the optimizer simpler, and the proof simpler: our language is an *untyped intermediate language,* not a typed source language. Proofs in earlier CertiCoq phases guarantee the program will not get stuck.

Rule E_APP shows how applications are evaluated. When a function f is applied to arguments $\vec{y}$, we look up f in the environment ρ to retrieve the function closure $(\rho', \vec{fd}, f)$. Next, we find function f in $\vec{fd}$ with arguments $\vec{x}$ and function body e. We then evaluate the function body e in saved environment ρ' extended with bindings for each mutually recursive function in $\vec{fd}$ and by associating each y_i in $\vec{y}$ to their respective x_i in $\vec{x}$.

We define $\text{FV}(e)$ and $\text{BV}(e)$ to be respectively the set of free and bound variables of a term e or of a bundle of function definitions $\vec{fd}$. We also define $\text{names}(\vec{fd})$ to be all the names of functions from the bundle:

$$\text{names}(\vec{fd}) := \{f | f\ (\vec{x})\ =\ e \in \vec{fd}\}$$

An important property that is not enforced by the syntax presented in Fig. 1 is that bound names are globally unique. This property is easy to achieve and maintain; the translation from the previous intermediate language uses a state monad to assign unique variable names. We also make sure that the free variables of the top-level program are disjoint from its bound variables. This allow us, for example, to perform function inlining without worrying about variable capture. We define the proposition $\text{UB}(e)$ to assert that e has the unique binding property.

4 Data representations, abstract state

From our CPS IR, we generate stackless closure-passing Clight code. Targeting CompCert Clight rather than machine code has multiple benefits for CertiCoq. CompCert is a mature compiler targeting many architectures, including x86-32, x86-64, ARM-32, ARM-64, RISC-V, and Power-PC. CompCert is a significant proof effort, and we would have to repeat much of the same work if we were to translate down to machine code.

Although L6 is a pure functional language and Clight an impure imperative language, restrictions over L6 (described in the previous section) facilitate the generation of equivalent Clight programs.

Each function in L6 corresponds to a function in Clight—which is possible because closure-conversion has unnested everything. All calls are tail calls, and we rely on the C compiler to implement tail calls without pushing the stack (CompCert, `gcc`, and `clang`/LLVM all do tail-call optimization). An expression e in L6 (such as a function body) can be viewed as an *extended basic block,* that is, a tree of control flow, where the branches are at `match` expressions, and the leaves are tail calls. Calls to function-variables in L6 translate directly to calls to function-pointer variables in Clight.

This setup is much simpler than the C translation framework used by previous functional-to-C compilers such as sml2c [9]. In fairness, twenty years ago one could not rely upon a C compiler to do tail-call optimization, so a direct function-to-function correspondence was impractical.

4.1 Representing datatypes in C

We represent values on the heap using a representation used by many ML compilers including OCaml [10] and SML/NJ[11], leaving nullary constructors unboxed and boxing non-nullary ones. To differentiate unboxed values from pointers to boxed values, we use the last bit of unboxed values as a flag set to 1 – meanwhile, the pointers used for boxed values are word-aligned, always ending in 0.[2]

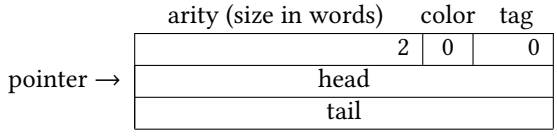
For an inductive type `T`, defined as shown in Figure 3, we assign (unboxed) ordinal `0` to `A` and `1` to `C`, and (boxed) ordinal `0` to `B` and `1` to `D`.

```
datatype T : Type
| A : T
| B : R -> T
| C : T
| D : R -> S -> T
```

Figure 3. Example of a Coq datatype

Unboxed values are kept in local memory as the integer $2 \times \text{ordinal} + 1$. Boxed values of arity n are represented as a pointer to the second of $n + 1$ contiguous memory location, each of the size of a value, providing access to the representation of each of its fields, while the first location holds a header containing the ordinal and the arity of the constructor.

We use the Ocaml data representation [12, chapter 20] in which a heap-allocated record is preceded by a header word; we show here a Cons cell:



[2] In this paper, we assume a 64-bit architecture, with 8-byte pointers. However, CertiCoq works in 32 and in 64-bit mode, and its proof is parameterized over the size of pointers.

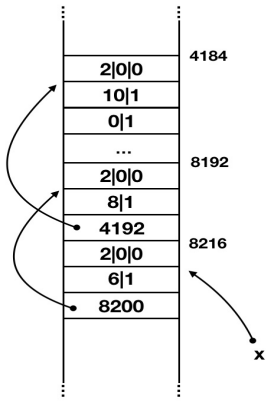
Arity is 2 because Cons carries 2 arguments; tag is 0 because Cons is the first value-carrying (boxed) constructor of its inductive type; *color* is used only by the garbage collector.

```
Cons 6
 (Cons 8
   (Cons 10
      Nil)))
```

(a) A list in Coq

```
(* Where 5 -> Nil
  and 6 -> Cons *)
Vconstr 6 [Vint 6;
 Vconstr 6 [Vint 8;
  Vconstr 6 [Vint 10;
   Vconstr 5 []]]]
```

(b) A list in L6



(c) A list in Clight

Figure 4. An inductive value through the compilation

Figure 4 shows an inductive value represented in different languages during compilation. The list `6::8::10::[]` can be represented in an inductive datatype `list` with constructor `Nil: list` and `Cons:int -> list -> list` (see Figure 4b). In L6, information about the constructors is kept in a global map – we assume here that 5 refers to `Nil` and 6 to `Cons`. Finally, as shown in Figure 4c, in Clight, non-nullary constructors are represented in the heap as boxed values. We use the notation $n|m$ to represent concatenating the bit representation of n with the one of m. Here, the list is represented as three blocks of three values each. The head of the list is pointed to by x at address 8224. The value before that, 2|0|0, is the header for `Cons`, as explained above. The first field contains the unboxed value 6, while the second contains a pointer to the next element of the list, at memory location 8200. This value has the same header, also representing `Cons`, and holds unboxed value 8 in its first field, and a pointer to memory location 4192 in its second. 4192 holds the last link in our list, with a representation of Nil (first unboxed constructor of `list`) in its second argument.

4.2 Garbage Collection

C does not have automatic garbage collection, so we need to provide it ourselves. Wang *et al.* [7] have proved the correctness of a high-performance generational collector implemented in C, using the *Verifiable C* program logic [13] embedded in Coq. We need to prove the correctness of our interface to that collector—or rather, to any collector that uses the same general-purpose interface.

The compiled program must call the garbage collector whenever it determines that the free space is insufficient to allocate the next record on the heap. At that time, the collector must be able to find all the roots of the graph of reachable objects; typically these are spread over the stack frames of currently executing threads (CertiCoq generates thread-safe code). Finding the roots is (usually) a hard problem, and high-performance solutions [14] are quite complex. Furthermore, stack scanning usually requires substantial cooperation from back-end phases of the compiler—but C compilers have no support for such cooperation.

McCreight *et al.* [15] and Dargaye [16] show a way to handle those challenges, making use of a "shadow stack" [17] to store values across calls. Their intermediate languages include primitives to explicitly keep track of roots, guiding the translation to a Clight program interfacing with a garbage collector. However, in both cases, measured overhead for the generated code was high due to the expensive operations that had to be performed over any local variables that could contain a live pointer.

We largely avoid this problem by having no stack (all calls are tail calls) and by testing for out-of-heap only at function-entry. Since L6 is in CPS and closure-converted, at function entry the live roots are exactly the arguments to the function. We explain the mechanism below.

4.3 Abstract state for the generated code

We define an abstract state that we target when translating from L6, and later instantiate it as concrete Clight memory and local environment.

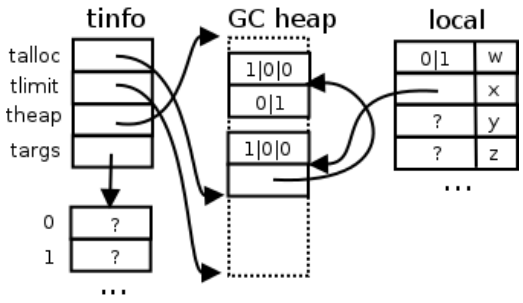


Figure 5. Abstract state for the generated code

The abstract state has three components:

tinfo (thread info), a structure containing three pointers representing the state of `GC heap` and a fourth pointer to an array containing function-arguments (roots).

GC heap, an abstract representation of a heap containing boxed values at nonoverlapping addresses.

local, a table containing mappings from variables to either a pointer to the `GC heap` or an unboxed value.

Assuming a source program using inductive type nat with constructors S and O, the example in Figure 5 shows a local environment containing the mapping (w, O), with O represented as the first unboxed constructor of nat, and $(x, S(SO))$,

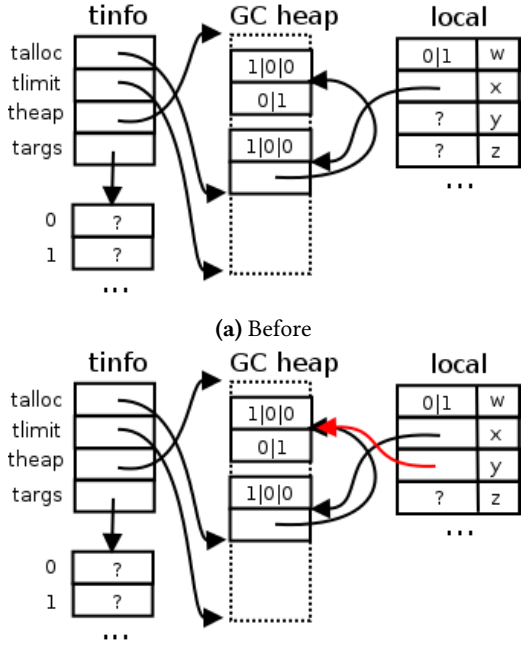
with $S(SO)$ represented as a pointer to a pair of words containing the header (S has arity 1 and is the first boxed constructor), and a pointer to a second pair of words containing the same header, and the representation of O, as discussed in Section 4.1.

The tinfo describes the state of the GC heap as we will explain below. Throughout the next section, we show the effect of each L6 operation on the abstract state, before instantiating the state in Clight and generating corresponding Clight statements.

4.4 Simulating L6 in the Abstract State

Our proof of code-generator correctness uses a simulation relation between L6 states (as shown in Figure 2) and Clight states, each of which has a *data* part (as in Figure 5) and a *control* part (a Clight statement). Here we sketch how each of the L6 transitions affects the data part of the abstract state.

Eproj "let y = Proj$_0$ x in e" executes, according to rule E_PROJ, by setting y to the first projection of x (which has been constructed using S) before executing e. Figure 6 shows the effect of executing this expression starting from the abstract state from Figure 5. In this case, where x represents $S(SO)$, y ends up pointing to the representation of SO.

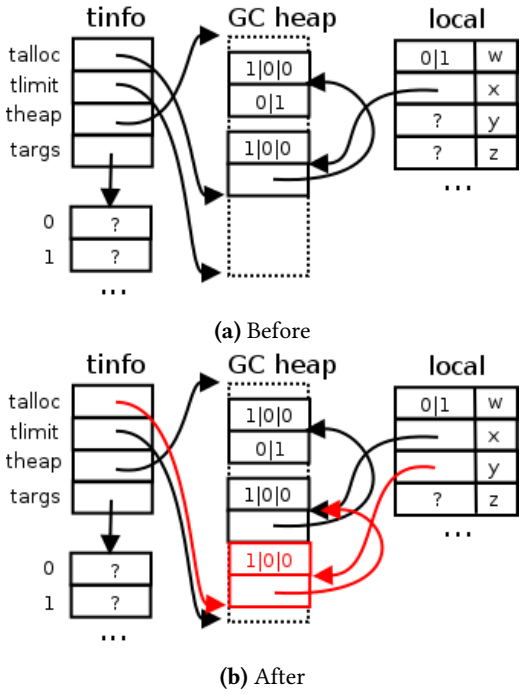


(a) Before

(b) After

Figure 6. The effect of "let y = Proj$_0$ x in e" on the abstract state

Econstr "let y = Con S x in e", according to Rule E_CONSTR, has y set to be a value constructed by applying S to the value of x. Figure 7 shows the effect of this on the abstract state. In Figure 7b, y points to a representation of $S(S(SO))$.



(a) Before

(b) After

Figure 7. The effect of "let y = Con S x in e"

Ecase "match x with $\vec{b}$" has no effect on the data part of the abstract state.

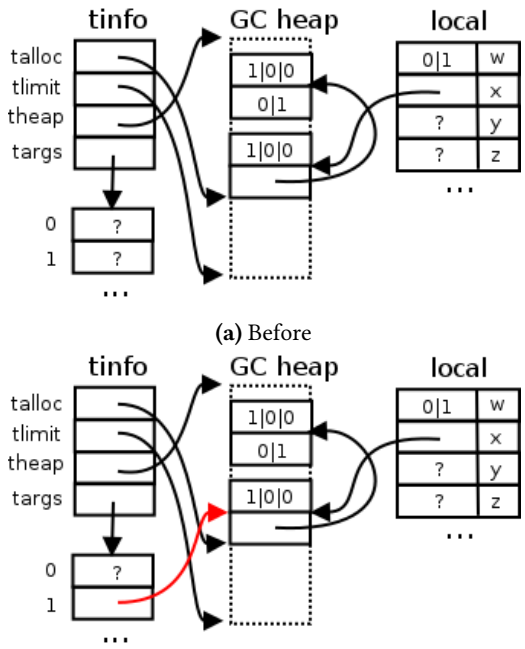
Ehalt "halt x", according to E_HALT, evaluates to the value of x. By convention, we take the second field of the arguments array to hold the return value. As shown in Figure 8, this results in targs$_1$ representing $S(SO)$.

Eapp "App f w x" calls function f y z = e on arguments w and x using the calling convention shared by all functions with tag t (see E_APP). Figure 9 shows the effect of evaluating this expression on the abstract state, assuming that t corresponds to a calling convention where the first argument is kept in cell 0 and the second in cell 1 of the arguments array.[3] As shown in Figure 9, this is done in two steps: First, the arguments w and x are copied to targs according to t, which results in targs$_0$ representing O and targs$_1$ representing $S(SO)$ (see Figure 9b). Then, as shown in Figure 9c, we restore targs$_0$ and targs$_1$ to the function parameters y and z before proceeding with the execution of e.

4.5 From abstract state to Clight memory

We realize the data part of the abstract state using a Clight memory and a Clight local-variable environment. Figure 10 shows how we map different portions of the abstract state and of the program to disjoint portions of memory.

[3] As we will later show, a portion of the arguments array is passed directly as C arguments, and further compiled to be passed in registers



(a) Before

(b) After

Figure 8. The effect of "halt x"

One area of the heap holds, for every function f in the source program, function information finfo_f and function code fcode_f. A disjoint area holds tinfo and the targs array. Finally, a third area holds the boxed values which were contained in the GC heap from Figure 5.

4.6 Interface with garbage collector

Rather than integrating a specific garbage collector with our generated code, and proving our code generation phase correct with respect to that particular garbage collector, we provide a general interface for garbage collection, and prove our code generation phase correct with respect to a more general notion of garbage collection.

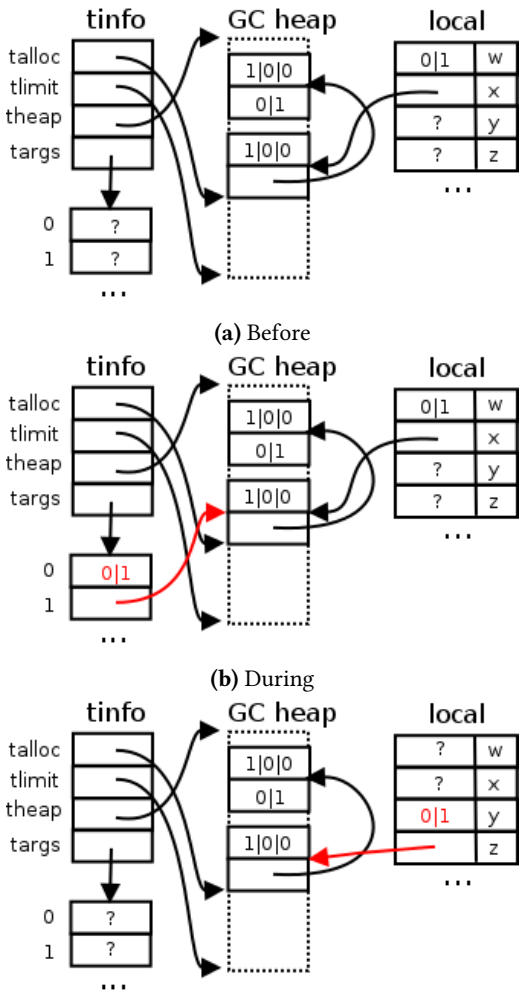
As shown in Figure 10, tinfo is a C structure containing, three pointers describing the state of the garbage-collected portion of memory, and then a pointer to argument (live roots) array:

talloc, a pointer to the next allocatable word of memory in the "nursery" generation;

tlimit, a pointer to the end of the allocatable portion of nursery;

theap, a pointer to the garbage collector's own description of its memory regions; the format of this data is left abstract to the CertiCoq compiler

targs, a pointer to an array where function-arguments are passed in every call that this thread makes (CertiCoq-compiled code is thread-safe).



(a) Before

(b) During

(c) After

Figure 9. The effect of "'App f w x" on the abstract state

The first k arguments are passed in registers, i.e., as function-parameters in the C source code; beyond that, we use slots in targs. Just before any call to the g.c., those (up to) k arguments are stored into targs.

Then the specification of the garbage collector can assume:

1. Any live portions of GC heap are reachable from the live roots in targs– which is to say, since the live roots correspond to the environment computed by closure conversion, functions are fully closed after closure conversion.
2. Between calls to the collector, the generated code may allocate (by increasing the value of talloc) but it will never increase talloc beyond tlimit. A call to the collector requests n bytes, where n is any value up to the nursery size 2^{16} bytes by default. We parameterize CertiCoq by the nursery size, and refuse to compile functions which would request more than this amount.

On return, the collector guarantees that

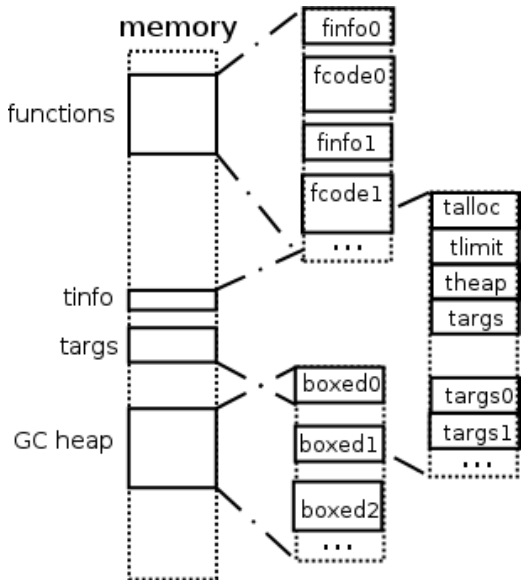


Figure 10. Mapping of the abstract state in a Clight memory

1. the arguments array contains Clight values representing the same L6 values as before;
2. tlimit – talloc/*gen*.

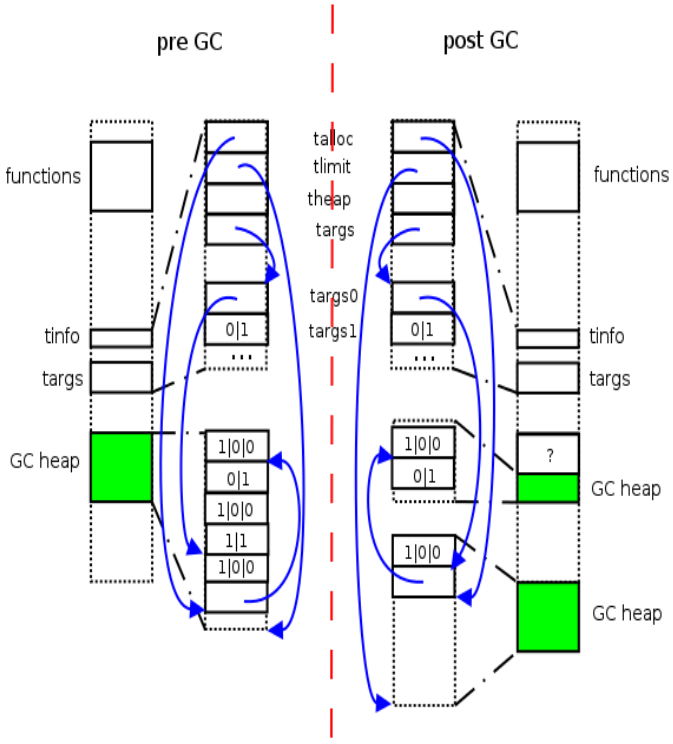


Figure 11. The heap before and after garbage collection

Figure 11 shows a heap before and after garbage collection. At left, the heap contains an area with function declarations and finfo, followed by tinfo and the arguments array targs and finally the GC heap, whose area is represented in green. In the blown up view of tinfo, talloc points to the start of the free area in the GC heap, tlimit to the end. We do not represent the value of theap in the picture – it is used by the garbage collector to keep track of its memory regions, and left abstract to our compiler. Finally, targs points to the start of the arguments array. In the arguments array, the first slot is taken by a pointer to the representation of $S(SO)$ in the GC heap. The second slot is taken by a representation of O.

On the right, representing the state of the heap after garbage collection (assuming the roots were slots 0 and 1), the areas are unchanged, except for the GC heap – the new GC heap may only contain part of the old GC heap or newly allocated portions of memory. The pointers in talloc and targs have been updated to reflect the new GC heap.

5 Code generation

Our code generator is a Gallina function that translates L6 ASTs into CompCert Clight ASTs. After closure conversion and hoisting, all functions are part of the same mutually recursive bundle $\vec{fd}$, and we know that the body of the program `e` does not contain function declarations. Moreover, we know that for any function `(f,`$\vec{y}$`,e')` in $\vec{fd}$, `e'` does not contain function declarations.

The translator processes let $\vec{fd}$ in e by

1. computing the arity (number of function-parameters) and the maximum number of words allocated by each function, producing a map $\theta : var \to \mathbb{N} \times \mathbb{N}$ with this information.
2. generating forward declarations for all functions in $\vec{fd}$, as they may represent mutually recursive functions in Coq (and in C).
3. For each function in $\vec{fd}$, of the form `(f,`$\vec{y}$`,e')`, creating a Clight function `f` with `tinfo` as argument and `codegen(e')` as function body
4. generating a Clight function body, converting e using codegen.

5.1 Code generation for L6 functions

We represent each L6 function as a Clight function, taking as first parameter a tinfo pointer holding the information needed to execute the function. Up to k parameters of the L6 function are passed as additional C function parameters; the rest are stored in slots of the targs array. (This is designed to match LLVM's "ghc" calling convention; see below.)

Every function f is associated with a structure finfo_f containing the maximum number of words the function could allocate, followed by its arity, and, for each of its arguments, the slot-numbers used in the arguments array when garbage collection is called. For example, a function myfunc with three arguments held in slots 0,1,2 and allocating at most ten words:

```
value const f_info_myfunc[5] = {10, 3, 0, 1, 2, };
```

If the function's arity is larger than k, the additional arguments are stored in their appropriate slots of the targs array during function calls.

On function entry, we generate code to test whether the maximum number of memory words that could be allocated by the function[4] is more than the difference between tlimit and talloc. If so, we need to call the garbage collector (described in 4.2) before restoring the arguments to local memory, and proceeding with the body of the function.

```
void f(struct thread_info *tinfo,x,y){
// local variable declarations ...
args = tinfo->args;
if (!(*f_code_info <= tinfo->limit - tinfo->alloc)) {
  // store (up to k) parameters into args ...
  args[0] = x;
  args[1] = y;
  garbage_collect(f_info, tinfo);
  // load (up to k) parameters from args ...
  x = args[0];
  y = args[1];
}
alloc = tinfo->alloc;
limit = tinfo->limit;

// load parameters beyond k from args into local vars
z = args[2];

// function body ...
}
```

Figure 12. Start of a function generated by our back end (with k = 2)

5.2 The code generation algorithm

The function codegen $\Delta\ \theta\ e$ translates L6 expression e into a Clight statement. L6 is designed to make this translation fairly direct: for example, projections map directly to field access. Others, such as function application, are a bit more involved, as we will describe.

The other arguments of codegen are,

Δ maps constructor tags to the name, arity and ordinal of the constructor and to the name and tag of its datatype.

θ maps the name of each function to its arity, calling-convention, and the maximum number of values its body could allocate. Knowing which variable represents a function is also important to determine if a variable represents a value in local memory, or if it represents the address of a function in the heap.

Maximum number of arguments per function. Source-level Gallina functions take exactly one argument; after CPS conversion there may be up to two; uncurrying and lambda lifting may add more; and closure conversion can add one more. We can limit uncurrying and lambda lifting to ensure that no L6 function has more than, for example, 2^{10} arguments. Therefore the targs array can be of fixed size 2^{10}.

Maximum number of arguments per constructor. Our header representation (see Section 4.1) allows for 54 bits for the arity of constructors. CertiCoq will refuse to compile a program if it uses a data constructor with more than 2^{54} arguments.

Maximum number of constructors per datatype. Our header representation (see Section 4.1) allows for 8 bits for the ordinal of a boxed constructor, and 63 for unboxed constructors. CertiCoq will refuse to compile a program if it refers to an inductive datatype with more than $2^8 - 1$ non-nullary constructors or more than $2^{63}-1$ nullary constructors (or $2^{31} - 1$ on 32-bit machines).

Full program compilation. Our proof of correctness assumes we are compiling a closed program of nonfunctional type. We could do separate compilation and linking by "lambda binding imported modules" [18], in which each module is a closed higher-order function, and linking is just function application.

Axiomatized (external) function. As detailed previously, we make use of the last bits of Clight values to differentiate between integers (arity-0 constructors) and aligned pointers. Unfortunately, the Clight semantics does not allow for this distinction. We thus axiomatize an "inlineable external function" isptr to return *true* on aligned (divisible by word size) addresses. To realize this axiom as a theorem would require an extension to the CompCert correctness proof (but not a change to the behavior of the CompCert compiler).

Correctness of the conversion environments. When starting code generation, we receive global environments describing the name and components of inductive datatypes found in the program. By this point of the compiler, we can assume

- every constructor found in the program is represented in Δ, and
- every constructor is applied to the number of arguments corresponding to its arity as recorded in Δ.

5.3 Invariants in the proof of correctness

To generalize the statement of correctness (see Theorem 5.5), we need invariants asserting the correspondence between the L6 structures, the conversion environments, and the Clight state. We now detail the invariant used in the proof.

Allocatable space. In the expression simulation relation, we assert that there is enough writable space between the allocation pointer and the limit pointer to allocate all the values assigned in the expression:

Theorem 5.1 (Sufficient allocatable space)**.**

$$(8 \times m \leq \mathit{tlimit} - \mathit{talloc})$$

[4]Because an L6 "function" is a tree of control flow, there is a bounded, statically determinable number of allocations of fixed, statically known size.

This fact is reasserted at each function entry using, when needed, the proof that running the garbage collector results in enough space. This stays true throughout evaluation – since we provision memory for the heaviest path of the function, any step preserves or reduces the sum of the space used currently and the space needed until the end of the function.

Separation of spaces in memory. In a disjoint area of the heap, we keep the structure describing the current state of the allocatable space and information about the running program (as described in Section 4.6). It is important that this space is separated from the allocatable space, and that both of these are separated from the portion of memory holding the code portion of functions.

5.4 Specification of the interface with garbage collection

At the proof level, we axiomatize the effect of garbage collection on the provided interface (described in Section 4.6). We separate the logical portion of the proof from the spatial component:

Before garbage collection, we have a list of roots $\vec{v_7}$ held in the arguments array of tinfo at position described in finfo pointing to the garbage-collected area L (which we referred to as "GC heap" previously in this section) of a memory m and representing a list of L6 values $\vec{v_6}$.

After garbage collection, in the same arguments array of tinfo, and at the same position described in finfo, we have a list of roots $\vec{v_7}'$ pointing to a modified garbage-collected area L' of a memory m' representing the same list of L6 values $\vec{v_6}$. We also know that the space between the new talloc and tlimit pointer of the updated tinfo is writable, and at least the required size as described in finfo.

Theorem 5.2 (program_gc_inv, correctness of g. c.)**.**

$$\begin{aligned}&GC_{finfo_f}\ tinfo\ m = (m', tinfo') \implies\\ &\quad(\forall i \in finfo_f, v_6 \simeq^{\text{val}}_{\theta,m} targs[i] \Rightarrow v_6 \simeq^{\text{val}}_{\theta,m'} targs[i]) \wedge\\ &\quad tlimit' - talloc' \geq finfo_f[0] \wedge\\ &\quad(\forall talloc'_o \leq o < tlimit'_o, \text{Writable}(m\ talloc'_b\ o))\end{aligned}$$

On the spatial side, before garbage collection, L contains all the pointers reachable from $\vec{v_7}$, and is disjoint from tinfo and the area in which functions are allocated. After garbage collection, any location in m not in L and tinfo is unchanged. tinfo is still allocated at the same location in m', but the values it holds may have changed. Finally, all pointers reachable from $\vec{v_7}'$ are contained in L', which is disjoint from tinfo and the area in which functions are allocated.

Theorem 5.3 (program_gc_inv, spatial assumptions w.r.t. g.c. correctness)**.**

$$\begin{aligned}&GC_{finfo_f}\ tinfo\ m = (m', tinfo') \wedge\\ &(\forall talloc_o \leq o < tlimit_o, \neg L\ talloc_b\ o) \Rightarrow\\ &\quad\exists L', (\forall talloc'_o \leq o < tlimit'_o, \neg L'\ talloc'_b\ o) \wedge\\ &\quad(\forall i \in finfo_f, \forall b\ o, \text{reachable}_{m'}\ targs[i]\ b\ o \Rightarrow L'\ b\ o)\end{aligned}$$

5.5 A correct generational garbage collector

The generational garbage collector developed for CertiCoq has been proved correct by Wang *et al.* [7]. We showed that their representation of garbage collection is compatible with our interface. However, we have not yet proven the spatial portion of the interface. This is because the garbage collector has been proved correct using the VST program logic [19], while the code generator is proved correct directly over the semantics of *Clight*. The proof could be completed by unfolding the definition of VST's Hoare triple (semax), as described in *Program Logics for Certified Compilers* [13]. Doing so would allow us to show that only the portions concerned with garbage compilation (which is to say, tinfo, targs and the GC heap) have been affected by garbage collection.

5.6 Forward simulation between L6 and Clight

We prove correctness of the code generator by a simulation between L6 and Clight. The proof goes by induction on the L6 big-step evaluation derivation, in well-formed L6 and Clight environments.

The top-level statement of correctness states that if program P evaluates to value v, and that s_P is generated from code generation over P, then in an appropriate initial memory m, s_P steps to a memory m' containing (in targs_1) a representation of v:

Theorem 5.4 (correctness of code generation)**.**

$$\begin{pmatrix}\cdot \vdash P \Downarrow v \wedge\\ \text{codegen}_{\Delta,\theta}\ P = s_P \wedge\\ \text{init}(m)\end{pmatrix} \implies \begin{pmatrix}\exists m'.\\ \cdot, \cdot \vdash s_P, m \Rightarrow^{\epsilon} \cdot, m' \wedge\\ (\cdot, v) \simeq^{\text{val}}_{\theta,m'} targs_1.\end{pmatrix}$$

We generalize this statement to an open term e by relating its evaluation context ρ with a Clight triple (G, m, l) containing a global environment, a memory and a local environment:

Theorem 5.5 (repr_bs_L6_L7_related, generalized statement of correctness for code generation)**.**

$$\begin{pmatrix}\text{UB}(\rho, e) \wedge\\ \text{INV}_{m,l}(tinfo) \wedge\\ \text{INV}_e(\Delta, \theta) \wedge\\ \rho \vdash e \Downarrow v \wedge\\ \text{codegen}_{\Delta,\theta}\ e = s_e \wedge\\ \rho \simeq^{\text{env}}_{\theta,e} (G, m, l)\end{pmatrix} \implies \begin{pmatrix}\exists m',\\ G, l \vdash s_e, m \Rightarrow^{\epsilon} \cdot, m' \wedge\\ (\rho, v) \simeq^{\text{val}}_{\theta,m'} targs_1.\end{pmatrix}$$

$\text{UB}(\rho, e)$ asserts that the unique binding property holds globally over e and every value in ρ.

$\text{INV}_e(\Delta, \theta)$ records the assumptions in the proof of correctness described in Section 5.2, while $\text{INV}_{m,l}(\text{tinfo})$ ensures the preservation of the invariants described in Section 5.3.

The memory relation $\rho \simeq^{\text{env}}_{\theta,e} (G, m, l)$ states that any function f in ρ has a corresponding Clight function held at Vptr $G(f)$ 0 in m, and any other free variable x of e has a correspoding Clight value in the local environment l and

memory m according to $\rho(x) \simeq^{\text{val}}_{\theta,m} l(x)$ (which captures the data representation described in Section 4). We write $\rho(x) \simeq^{\text{val-id}}_{G,m,l} x$ as the value-identifier relation over both functions (held in G) and other values (in l).

We provide here the details of each case of the proof, corresponding to the evaluation rule of the semantics of L6 included in Figure 2:

$\rho \vdash \text{halt } x \Downarrow v$ By the evaluation derivation (E_HALT), we know that $\rho(x) = v$. Since x is free in halt x, by the memory relation, we know that $v \simeq^{\text{val-id}}_{G,m,l} x$. By $\rightsquigarrow$, the value corresponding to v will be placed in the first slot of the arguments array, as required.

$\rho \vdash \text{let } x = \text{Proj}_n\ y \text{ in } e \Downarrow v$ By the evaluation derivation (E_PROJ), we know that $\rho(y) = c\ \vec{w}$, and that $\rho; x \mapsto w_n \vdash e \Downarrow v$. Since y is free in let $x = \text{Proj}_n\ y$ in e, by the memory relation, we know that $v \simeq^{\text{val-id}}_{G,m,l} x$. Since v is a boxed constructor, we have $l\ y = \text{Vptr } b\ o$ and $(c,\ \vec{w}) \simeq^{\text{val}}_{\theta,m} \text{Vptr } b\ o$. By inversion on the value relation, we could only be in the boxed constructor case VR_BCON, and $m[b, o + (n \times 8)] = v^7_n \wedge w_n \simeq^{\text{val}}_{\theta,m} v^7_n$). Extending l with $x \mapsto v^7_n$ by stepping through the assignment statements provides a local environment and memory related to $\rho; x \mapsto w_n$, and we can apply the induction hypothesis on $\rho; x \mapsto w_n \vdash e \Downarrow v$ and codegen$(e) = s$.

$\rho \vdash \text{let } x = \text{Con } c\ \vec{y} \text{ in } e \Downarrow v$ The constructor case of code generation relies on assumptions we are holding about allocatable space in the Clight memory. By the evaluation derivation (E_CONSTR), we know that $\forall_{y_i \in \vec{y}}, \rho(y_i) = w_i$ and $\rho; x \mapsto (c, \vec{w}) \vdash e \Downarrow v$. Here, we need to consider two different cases, corresponding to our value representation described in Section 4.1:

If $\text{ord}_\Delta\ c = 0$, then we are generating Clight code "Sset x $((\text{ord}_\Delta\ c \ll 1) + 1)$; s" where codegen$(e) = s$. Stepping through the assignment statement updated the local environment to $l, x \mapsto \text{hdr}_\Delta\ (c)$, with $\rho; x \mapsto (c, \vec{w}) \simeq^{\text{env}}_{\theta,e} (G, m, l; x \mapsto \text{hdr}_\Delta\ (c))$. Correctness follows by induction hypothesis on codegen$(e) = s$ and $\rho; x \mapsto (c, \vec{w}) \vdash e \Downarrow v$.

If $\text{ord}_\Delta\ c \neq 0$, we are generating code to allocate a boxed value: "codegen(let $x = \text{Con } c\ \vec{y}$ in e) = Sset $x{-}1$ $((\text{arr}_\Delta\ c \ll 8) + \text{ord}_\Delta\ c)$; Sset $y_i\ v_{i+1}$; s". In this case, because of our assumption about the memory, we know that there is enough allocatable space in m after the allocation pointer for $|\vec{y}| + 1$ value-sized blocks. First, we place the header of c in $m[alloc]$. Then, for each $y_i \in \vec{y}$, we have $w_i \simeq^{\text{val-id}}_{G,m,l} y_i$ by the memory relation, and we store that value at $m[alloc + (i \times |_|val)]$. Finally, we set x to point after the header, at $m[alloc + 8)]$, establishing all the necessary pieces for VR_BCON to hold for $c\ \vec{y}$, and extending the memory relation to have $\rho; x \mapsto (c, \vec{w}) \simeq^{\text{env}}_{\theta,e} (G, m, l; x \mapsto \text{Vptr } b_{alloc}\ o_{alloc} + 8)$. Before using the induction hypothesis on codegen$(e) = s$ and $\rho; x \mapsto (c, \vec{w}) \vdash e \Downarrow v$, we update the allocation pointer's offset to to $o_{alloc} + (|\vec{y}| + 1) \times 8$, and reestablish the assumption that is enough allocatable space for the allocation in the heaviest path in e after the new allocation pointer.

$\rho \vdash \text{match } x \text{ with } \vec{b} \Downarrow v$ Code generation for case-statement relies on the correctness of our constructor environment, ensuring that each constructor c of an inductive type has a distinct $\text{hdr}_\Delta\ c$. It also relies on the axiomatized semantics of isptr properly distinguishing between our representation of boxed and unboxed values (see Section 4.1). By the evaluation derivation (case E_MATCH), we know that $\rho(x) = c\ \vec{w}$, $(c \Rightarrow e) \in \vec{b}$ and $\rho \vdash e \Downarrow v$. Well-formedness of $\vec{b}$ ensures that only one case matches c. By the environment relation, we have $v \simeq^{\text{val}}_{\theta,m} l\ x$ (as x is matched on, and thus cannot be a function). If c is a nullary constructor, we are in the unboxed case (VR_UCON) and we switch on the header held unboxed in l. Otherwise, $l\ x$ is a pointer, and we can recover the header and switch on it. In both cases, the recursion is done on $\rho \vdash e \Downarrow v$ and codegen$(e) = s$ with a Clight continuation skipping through the remaining case of the switch.

$\rho \vdash \text{App } f\ \vec{y} \Downarrow v$ Application is by far the most complicated case of this proof, as it relies on multiple assumptions of correctness for environments in order to properly save and restore arguments from the arguments array, to perform a call to the right location in memory, and to reestablish environment assumptions about the new code block by calling, if needed, garbage collection.

As mentioned previously, the first k arguments to f are passed as function-parameters, while any further arguments are passed through the arguments array. $\rightsquigarrow$ generates statements to place the values corresponding to $\text{skip}_k \vec{y}$ in appropriate slots of the arguments array.By the memory relation, we have corresponding Clight values in g (in the case of functions) or l (for constructors) for any $y_i \in \vec{y}$.

Then, we step through the call to f, held in the global memory of the Clight program p. The expression relation ensures that a corresponding function info finfois available in the global environment, and that f maps to a sequence of statements consisting of a conditional statement for garbage collection, assignment statements restoring any extra arguments (if more than k) of f into local memory, followed by the translation of the body of the function.

At this point, we step through the conditional statement inserted by the code generator to ensure enough allocatable memory is available in the garbage-collected area for the heaviest path of e, the body of function f. If there is enough space, we proceed with the proof using $m' = m$. Otherwise, we store in the arguments array the arguments which were passed as function-parameters, before garbage collection is called, and we use its axiomatized semantics to prove that the memory after collection, m', is suitable and related to $\rho, \vec{x} \mapsto \vec{y}$ under the memory relation. We provide in section 4.2 details about the interface and axiomatized

semantics for garbage collection. For the code generation proof, we concentrate on three properties of m':

1. m is related to m' over $\vec{y}$.
2. tinfo has been properly updated, and there is enough space in m' between talloc and tlimit.
3. nothing outside in tinfo, targs or the garbage-collected area has changed between m and m'.

This ensures all of the assumptions about m are still true about m', in addition to enough space in m' being available.

We then restore the arguments of f from the arguments array to the new local environment l'. Since f is closure-converted, we know that all of the free-variables of its body are bound as arguments (potentially in the environment argument). Meanwhile, because f is hoisted, any function in ρ is also present in ρ', so that portion of the memory relation is preserved. Taken together, these two steps ensure that the memory relation $\rho' \simeq^{\text{env}}_{\theta,e} (G, m', l')$ holds.

	Compiler	Params	Frame Pointer	Calling Convention	Time
1	CompCert	1	yes	standard	16.77 sec.
2	CompCert	6	yes	standard	15.11 sec.
3	LLVM	1	yes	standard	13.55
4	LLVM	1	no	standard	12.57
5	LLVM	6	yes	standard	10.99
6	LLVM	6	no	standard	9.96
7	LLVM	1	yes	ghc	11.95
8	LLVM	1	no	ghc	11.37
9	LLVM	6	yes	ghc	9.23
10	LLVM	6	no	ghc	8.76
11	LLVM	12	yes	ghc	9.11
12	LLVM	12	no	ghc	8.79
13	Ocaml	bytecode			17.84
14	Ocaml	native			2.36

Figure 13. VeriStar runtime evaluation

6 Performance and Evaluation

We measure performance on the VeriStar benchmark, a Gallina program for paramodulation-based resolution theorem proving for entailments in separation logic [22], 825 non-blank noncomment lines of code (not including embedded proofs). L6-to-Clight code generation takes 148 milliseconds (plus time for front-end phases of Certicoq and for CompCert, see [23]). In comparison, Coq's standard extraction to ocaml takes 124 ms followed by either 132 ms for ocaml byte-code compilation or 487 ms for ocaml native-code compilation.

Execution time of VeriStar (running on three difficult decision problems) on x86-64 is shown in Figure 13.

Performance is substantially affected by how many function-parameters we pass in registers, so we measure several C-compiler configurations.

Kranz [20] observed that a CPS-based compiler should pass arguments in registers—pass *many* arguments of *lambda-lifted* functions in registers. When compiling through C, we can pass only as many arguments in registers as the C calling convention permits. All-tail-call programs (or mostly-tail-call programs such as Haskell compiled via GHC through LLVM) do not perform well in a standard C calling convention: few parameters are passed in registers (6, on x86-64, the rest pushed on the stack), and there is substantial useless memory traffic restoring callee-save registers that will never be needed. For this reason, when GHC adopted LLVM as its back end [21], its designers implemented a special "ghc" calling convention in LLVM with 12 parameters passed in registers (the rest on the stack) and *no* callee-save registers. Furthermore, most C compilers support a "no frame pointer" compilation option; most C programs (and certainly ours) have no need for frame-pointer pushing and popping.

We use one C parameter for tinfo, and up to 5 more (or 11 more in ghc-cc) can be used for parameters; the rest use slots in the targs array. Since CompCert does not support ghc-cc, we also experiment with LLVM as a back end.

As one can see comparing rows 5 and 12, configuration settings no-frame-pointer and ghc-cc make a big difference: a factor of 1.25 in performance. Unfortunately, CompCert does not support either option. Comparing rows 2 and 5, informal examinations of the assembly-language outputs of CompCert and LLVM show that much of CompCert's slower performance may *also* be connected to tail-call overhead. Put together, this demonstrates strong motivation for adding per-function custom calling conventions to CompCert.

Even so, we do not match the performance of Ocaml native code: Our compiler is missing several important optimizations in phases earlier than the code generator.

7 Related Work

CakeML [27] is the most closely related work to our own: It is a proved-correct-in-HOL4 extracter/compiler/collector for ML (where pure ML also serves as the logical language of HOL4). Like CertiCoq, CakeML is a multiphase optimizing compiler with a garbage-collection interface, and they report good performance results.[5] Like CertiCoq, CakeML proves correct linkage to a proved-correct generational collector, but their collector must be written directly in a low-level StackLang intermediate language—it cannot be written in C. CakeML is direct-style and uses a stack; to find live roots, at every nontail call they push an index into a table describing which slots in the current stack frame are live. CakeML's intermediate DataLang is similar in some ways to our L6

[5] On "Large, Pure" benchmarks [27, Fig. 12] such as Knuth-Bendix that are rather smaller than our own VeriStar benchmark, their performance is worse than the optimizing MLton compiler by about a factor of 2—although this report was before they measured (presumably) improved performance from a generational collector.

CPS, in each case "the last language with functional-style abstract data," [27, §5], but DataLang does not have scoped variables (as do λ-calculus and CPS); and the semantics of DataLang has explicit reasoning about garbage-collector permutation of the data, whereas our L6 avoids any mention of g.c., postponing that to Clight code generation.

Oeuf [24] is a verified extraction pipeline for a restricted subset of Gallina to Cminor, an intermediate representation of CompCert. This project has been developed concurrently to CertiCoq. It does not support user-defined datatypes, limiting the users to a predefined set of base types. It avoids dealing with extraction concerns by requiring its source terms to be written using eliminators for the provided base types. It also assumes unbounded memories, which we don't – we formalized the interface with garbage collection, and our proof links with the proof of a verified garbage collector. On the other hand, Oeuf's correctness statement allows reasoning about code that calls Oeuf-compiled code, which is not supported currently by CertiCoq's correctness statement.

GCminor [15] is an intermediate language extending CompCert Cminor with primitives to interact with a garbage-collected heap, together with a library to define and prove the correctness of garbage collectors. A similar setup is presented in the thesis of Dargaye [16]. They do not use CPS, so source function calls are also C function calls. This approach suffers from high runtime overhead (factor of 2 compared with GHC), but it was not clear how much was from the shadow stack (to make pointers findable by g.c.), how much from doing a function call to allocate each block, how much simply from targeting C.

PVS2C [25] presents a code generator from PVS, an interactive proof assistant based on higher-order logic, to the C programming language. Unlike Gallina, PVS is impure, supporting references and array updates. To support this, and to avoid memory leaks, they implement a reference counting runtime system keeping track of the number of live references to each object, and collecting objects when their reference count drops to zero. A formal model for reference counting is presented by the same author in an earlier paper [26], and shown to not impact the execution of well-typed PVS programs. It instruments the operational semantics with explicit operations to add and subtract from reference counts kept in a new portion of the state. Our solution is more general; while we implement a generational copying garbage collector, which is significantly faster than reference counting, our garbage collection interface could be used by a reference counting collector.

Cogent [28] is a project that aims to generate correct C code from a specification language embedded in HOL4. Compiling a Cogent program generates a C program and a proof, in HOL4, that the semantics of the C program correspond to the original program. In addition to using different proof techniques, with proof-carrying code in place of simulation proofs in CertiCoq, the Cogent language is much more restrictive than Gallina, being limited to malloc-free functional programs, and aimed at the development of system software, for which a garbage collector would not be appropriate.

Other optimizing compilers have been developed, but not proved correct, from functional languages to C. We explore in the rest of this section the similarities and differences in compilation techniques used with CertiCoq.

Directly relevant is the sml2c project [9], which generated code from the SML intermediate representation, a direct inspiration for L6, into the C programming language. Fewer C compilers supported tail-call elimination at the time sml2c was developed, they instead relied on a dispatch loop.

Zinc→K2 is an optimizing compiler from CaML-light to C[29]. The research effort of this project is centered on optimizing function calls through *explicit specialization*. The compiler also uses a one-bit tag to differentiate between pointers and values, together with a copying garbage collector. A major different with CertiCoq is that their intermediate representation is not in continuation-passing style, so that they do not benefit from function entry having a defined sets of roots. Instead, their collector must deal with ambiguous roots spanning the whole accessible heap, a costly process.

8 Conclusion

In this paper, we presented a code generation phase from a functional intermediate language to a subset of the C programming language.

While code generators have been developed before between a functional language and an imperative one, the design of our intermediate language allows for a straightforward translation to C, which impacts both the performance of the generated code and the size of the proof.

Our garbage collection interface separates the challenge of finding roots from the correctness of garbage compilation, resulting in a more modular proof of correctness.

References

[1] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. CertiCoq: A verified compiler for Coq. *CoqPL*, 2017.

[2] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq project. In *ITP 2018: the 9th International Conference on Interactive Theorem Proving*, July 2018.

[3] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Theo Winterhalter. Coq Coq correct! Verification of type checking and erasure for Coq, in Coq. In *POPL'20 (conditionally accepted)*, January 2020.

[4] John Li. Verifying the uncurry phase of the CertiCoq compiler. Independent Work Report, 2018.

[5] Olivier Savary Bélanger and Andrew W. Appel. Shrink fast correctly! In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, PPDP '17, pages 49–60, New York, NY, USA, 2017. ACM.

[6] Zoe Paraskevopoulou and Andrew W. Appel. Closure conversion is safe for space. In *ICFP 2019: International Conference on Functional Programming*, 2019.

[7] Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. Certifying graph-manipulating C programs via localizations within data structures. In *Proceedings of the ACM on Programming Languages*, OOPSLA, 2019.

[8] Guy L. Steele, Jr. Rabbit: A compiler for Scheme. Technical report, MIT, Cambridge, MA, USA, 1978.

[9] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required : Compiling Standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2), June 1992.

[10] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release*, 2019. URL http://caml.inria.fr/pub/docs/manual-ocaml/. Version 4.08.

[11] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992.

[12] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml*. O'Reilly Media Inc., 2014.

[13] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, New York, NY, USA, 2014. ISBN 110704801X, 9781107048010.

[14] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically typed language. *SIGPLAN Not.*, 27(7): 273–282, July 1992.

[15] Andrew McCreight, Tim Chevalier, and Andrew Tolmach. A certified framework for compiling and executing garbage-collected languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 273–284, 2010.

[16] Zaynah Dargaye. *Vérification formelle d'un compilateur optimisant pour langages fonctionnels*. PhD thesis, Paris 7 – Denis Diderot, 2009.

[17] Fergus Henderson. Accurate garbage collection in an uncooperative environment. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, pages 150–156, New York, NY, USA, 2002. ACM. ISBN 1-58113-539-4. doi: 10.1145/512429.512449. URL http://doi.acm.org/10.1145/512429.512449.

[18] Matthias Blume and Andrew W. Appel. Lambda-splitting: A higher-order approach to cross-module optimizations. In *Proc. ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, pages 112–124, New York, June 1997. ACM Press.

[19] Andrew W. Appel. Verified software toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, pages 1–17, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19717-8. URL http://dl.acm.org/citation.cfm?id=1987211.1987212.

[20] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. *SIGPLAN Notices (Proc. Sigplan '86 Symp. on Compiler Construction)*, 21(7):219–33, July 1986.

[21] David A Terei and Manuel MT Chakravarty. An llvm backend for ghc. *ACM Sigplan Notices (Haskell Symposium 2010)*, 45(11):109–120, 2010.

[22] Gordon Stewart, Lennart Beringer, and Andrew W. Appel. Verified heap theorem prover by paramodulation. *ICFP'12: SIGPLAN Notices*, 47(9):3–14, 2012.

[23] Anonymized. PhD thesis, 2019.

[24] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. Oeuf: Minimizing the Coq extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 172–185, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5586-5. doi: 10.1145/3167089. URL http://doi.acm.org/10.1145/3167089.

[25] Natarajan Shankar. A brief introduction to the PVS2C code generator. In Natarajan Shankar and Bruno Dutertre, editors, *Automated Formal Methods*, volume 5 of *Kalpa Publications in Computing*, pages 109–116, 2018.

[26] Gaspard Férey and Natarajan Shankar. Code generation using a formal model of reference counting. In *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*, pages 150–165, 2016.

[27] Adam Sandberg Ericsson, Magnus O. Myreen, and Johannes Åman Pohjola. A verified generational garbage collector for CakeML. *Journal Automated Reasoning (JAR)*, 2018. doi: 10.1007/s10817-018-9487-z. URL https://link.springer.com/content/pdf/10.1007%2Fs10817-018-9487-z.pdf.

[28] Liam O'Connor, Christine Rizkallah, Zilin Chen, Sidney Amani, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Alex Hixon, Gabriele Keller, Toby C. Murray, and Gerwin Klein. COGENT: certified compilation for a functional systems language. *CoRR*, 2016.

[29] Regis Cridlig. An optimizing ML to C Compiler. In SIGPLAN Workshop on ML and its Applications, 1992.