

Formal Verification of C programs using C light Operational Semantics

Nika Pona
Digamma.ai

July 3, 2019

Outline

1. Formal verification - quick intro (high-level)
2. Coq mini intro
3. CompCert
4. Reasoning about C programs in Coq
 - ▶ C light syntax
 - ▶ Operational Semantics
5. Toy example: strlen
 - ▶ Informal specification
 - ▶ Formal specification of strlen
 - ▶ Simple implementation in C
 - ▶ From C program to AST using clightgen
 - ▶ Correctness proof
6. Conclusions

Formal verification - quick intro

We want to have high assurance that our code works as intended. One of the methods is formal verification. This means we want to produce a formal proof that our code works as intended. What does it mean exactly and how do we do it?

1. We start by writing a *specification* in a formal language (like Coq or HOL) which strictly defines how our program should behave.
2. Then we derive the *semantics* of our program which describes how it actually behaves.
3. Finally we mathematically prove that semantics of our program matches our specification.

Coq intro

As a formal language we choose **Gallina**, mechanized version of Calculus of Inductive Constructions (aka dependent type theory), which is a very expressive theory well studied in mathematical logic. It is much more likely to make a mistake in a formal proof (which is typically way longer than the code), so we want assurance that our proof is correct.

Hence we use a proof assistant Coq: a program that checks that your proof is correct. It also provides an environment to make the construction of proofs easier. Coq's language is based on dependent type theory and is called Gallina.

Coq intro

[Show some basic Coq definitions and proofs]

Coq has been used to conduct some big verification projects. One of them is CompCert, a verified compiler for C, almost entirely written in Coq and proved to work according to the specification (<http://compcert.inria.fr/>).

The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors.

(Finding and Understanding Bugs in C Compilers, Yang et al., 2011)

To achieve this they formalized C syntax and semantics (C99 standard).

Nice thing about Coq is that writing a specification is basically the same as writing a program that meets that specification, since Gallina is a functional programming language. One can extract the code to OCaml or Haskell to compile and run it.

Traditional approach

A function that needs to be verified can be written and proven in Coq directly. Then the Coq code can be extracted to OCaml. This approach, while being (possibly) easier, has some downsides:

- ▶ Extracted OCaml code shows low performance
- ▶ Coq's extraction mechanism is itself not verified so it can produce bugs unless restricted to ML subset of Gallina
- ▶ The extracted portion generally requires a pure-OCaml wrapper to be used with real-life data. Therefore, extraction necessarily adds a layer that cannot be verified - only unit-tested

New approach

We decided to try to verify the implementation of ASN1C compiler that already exists. This reduces TCB and moreover we could use the same techniques in other projects. We reuse parts of CompCert for this.

- ▶ parse C code into an abstract syntax tree using C light generator of CompCert (not verified)
- ▶ write a functional specification using CompCert's model of C light
- ▶ reason about the C light program using operational semantics defined in CompCert

Back to CompCert

CompCert formalizes C syntax and its semantics resulting in CompCert C and C light languages. In particular, they formalize C integers and memory model. I will briefly explain these since they are used throughout whole development.

Libraries of CompCert

CompCert Integers

Machine integers modulo 2^N are defined as a module type in `CompCert/lib/Integers.v`. 8, 32, 64-bit integers are supported, as well as 32 and 64-bit pointer offsets.

A machine integer (type `int`) is represented as a Coq arbitrary-precision integer (type `Z`) plus a proof that it is in the range 0 (included) to modulus (excluded).

```
Record int: Type :=  
mkint { intval: Z; intrange: -1 < intval < modulus }.
```

The function `repr` takes a Coq integer and returns the corresponding machine integer. The argument is treated modulo modulus.

CompCert Integers

Integer is basically a natural number with a bound, thus we can prove an induction principle for integers

```
Lemma int_induction :  
  ∀ (P : int → Prop), P Int.zero →  
    (∀ i, P i → P (Int.add i Int.one)) →  
      ∀ i, P i.
```

Proof.

By using induction principle for non-negative integers
`natlike_ind` for `Z`.



They prove some basic arithmetic theorems available through hints
`ints` and `ptrofs`.

Memory Model

A memory model is a specification of memory states and operations over memory. Cf. `CompCert/common/Memtype.v` (interface of the memory model)

Memory states are accessed by addresses, pairs of a block identifier b and a byte offset ofs within that block. Each address is associated to permissions (current and maximal): `Freeable`, `Writable`, `Readable`, `Nonempty`, `Empty`, ranging from allowing all operations to allowing no operation respectively.

Memory Type

Implementation of the memory type in
CompCert/common/Memory.v:

```
Record mem : Type := mkmem {  
  mem_contents: PMap.t (ZMap.t memval);  
  mem_access: PMap.t (Z → perm_kind → option permission);  
  nextblock: block;  
  access_max: ∀ b ofs, perm_order'' (mem_access*b ofs Max)  
              (mem_access*b ofs Cur);  
  nextblock_noaccess: ∀ b ofs k, (Plt b nextblock) →  
                                mem_access*b ofs k = None;  
  contents_default: ∀ b, fst mem_contents*b = Undef }.
```

Memory Model

The type `mem` of memory states has the following 4 basic operations over memory states:

`load` : read a memory chunk at a given address;

`store` : store a memory chunk at a given address;

`alloc` : allocate a fresh memory block;

`free` : invalidate a memory block.

A load succeeds if and only if the access is valid for reading. The value returned by `load` belongs to the type of the memory quantity accessed etc.

Applicative finite maps

Memory model and environments used in evaluation of the statements are modelled as applicative finite maps. Cf.

`CompCert/lib/Maps.v`

The two main operations are `set k d m`, which returns a map identical to `m` except that `d` is associated to `k`, and `get k m` which returns the data associated to key `k` in map `m`. There are two distinguish two kinds of maps:

1. the `get` operation returns an option type, either `None`
2. the `get` operation always returns a data. If no data was explicitly associated with the key, a default data provided at map initialization time is returned.

C light Syntax

C light types

Each expression of CompCert C has a type. There are:

- ▶ numeric types (integers and floats)
- ▶ pointers
- ▶ arrays
- ▶ function types
- ▶ composite types (struct and union).

An integer type is a pair of a signed/unsigned flag and a bit size: 8, 16, or 32 bits, or the special `IBool` size standing for the C99 `Bool` type. 64-bit integers are treated separately.

C light types

Cf. CompCert/cfrontend/Ctypes.v.

```
Inductive type : Type :=  
| Tvoid: type  
| Tint: intsize → signedness → attr → type  
| Tlong: signedness → attr → type  
| Tfloat: floatsize → attr → type  
| Tpointer: type → attr → type  
| Tarray: type → Z → attr → type  
| Tfnction: typelist → type → calling_convention → type  
| Tstruct: ident → attr → type  
| Tunion: ident → attr → type
```

C light types

C light types are subset of CompCert C types, we ignore the attributes.

Definition `tvoid` := `Tvoid`.

Definition `tschar` := `Tint I8 Signed noattr`.

Definition `tuchar` := `Tint I8 Unsigned noattr`.

Definition `tshort` := `Tint I16 Signed noattr`.

Definition `tushort` := `Tint I16 Unsigned noattr`.

Definition `tint` := `Tint I32 Signed noattr`.

Definition `tuint` := `Tint I32 Unsigned noattr`.

Definition `tbool` := `Tint IBool Unsigned noattr`.

Definition `tlong` := `Tlong Signed noattr`.

Definition `tulong` := `Tlong Unsigned noattr`.

Definition `tfloat` := `Tfloat F32 noattr`.

Definition `tdouble` := `Tfloat F64 noattr`.

Definition `tptr` (`t`: type) := `Tpointer t noattr`.

Definition `tarray` (`t`: type) (`sz`: Z) := `Tarray t sz noattr`.

C light Expressions

- ▶ (long) integer constant
- ▶ double/single float constant
- ▶ (temporary) variable
- ▶ pointer dereference (*)
- ▶ address-of operator (&)
- ▶ unary operation
- ▶ binary operation
- ▶ type cast
- ▶ access to a member of a struct or union
- ▶ size of a type
- ▶ alignment of a type

C light Expressions

All expressions and their sub-expressions are annotated by their static types. Within expressions, only side-effect free operators are supported, moreover, assignments and function calls are statements and cannot occur within expressions.

As a consequence, all Clight expressions always terminate and are pure: their evaluation have no side effects. This ensures determinism of evaluation.

C light Expressions

```
Inductive expr : Type :=  
| Econst_int: int → type → expr  
| Econst_float: float → type → expr  
| Econst_single: float32 → type → expr  
| Econst_long: int64 → type → expr  
| Evar: ident → type → expr  
| Etempvar: ident → type → expr  
| Ederef: expr → type → expr  
| Eaddrof: expr → type → expr  
| Eunop: unary_operation → expr → type → expr  
| Ebinop: binary_operation → expr → expr → type → expr  
| Ecast: expr → type → expr  
| Efield: expr → ident → type → expr  
| Esizeof: type → type → expr  
| Ealignof: type → type → expr  
}.
```

1

C light Expressions: Examples

```
(* 0 *)  
(Econst_int Int.zero tint)
```

```
(* 0 + 1 *)  
(Ebinop Oadd (Econst_int Int.zero tint)  
 (Econst_int (Int.repr 1) tint) (tint))
```

```
(* int *p *)  
(Etempvar _p (tptr tint))
```

```
(* (*p) *)  
(Ederef (Etempvar _p (tptr tint)) tint)
```

C light Statements

Inductive statement : Type :=

- | Sskip : statement (** do nothing **)
- | Sassign : expr → expr → statement
(** assignment lvalue = rvalue **)
- | Sset : ident → expr → statement
(** assignment tempvar = rvalue **)
- | Scall: option ident → expr → list expr → statement
- | Sbuiltin: option ident → external_function → typelist
→ list expr → statement (** builtin invocation **)
- | Ssequence : statement → statement → statement
- | Sifthenelse : expr → statement → statement → statement
- | Sloop: statement → statement → statement (** infinite loop **)
- | Sbreak : statement
- | Scontinue : statement
- | Sreturn : option expr → statement
- | Sswitch : expr → labeled_statements → statement
- | Slabel : label → statement → statement
- | Sgoto : label → statement

C light Statements

C loops are derived:

Definition $\text{Swhile } (e: \text{expr}) (s: \text{statement}) :=$
 $\text{Sloop } (\text{Ssequence } (\text{Sifthenelse } e \text{ Sskip Sbreak}) s) \text{ Sskip}.$

Definition $\text{Sdowhile } (s: \text{statement}) (e: \text{expr}) :=$
 $\text{Sloop } s (\text{Sifthenelse } e \text{ Sskip Sbreak}).$

Definition $\text{Sfor } (s1: \text{statement}) (e2: \text{expr}) (s3: \text{statement})$
 $(s4: \text{statement}) := \text{Ssequence } s1$
 $(\text{Sloop } (\text{Ssequence } (\text{Sifthenelse } e2 \text{ Sskip Sbreak}) s3) s4).$

C light Statements: Examples

```
(* int s = 1; *)  
  (Sset _s (Econst_int (Int.repr 1) tint))  
  
(* return s; *)  
  (Sreturn (Some (Etempvar _s tint)))  
  
(* while (s) {s = s - 1;} *)  
  (Swhile (Etempvar _s tint)  
    (Ssequence  
      (Sset _s (Ebinop Osub (Etempvar _input tint)  
        (Econst_int (Int.repr 1) tint) tint))))))
```

Unsupported Features

- ▶ `extern` declaration of arrays
- ▶ structs and unions cannot be passed by value
- ▶ type qualifiers (`const`, `volatile`, `restrict`) are erased at parsing
- ▶ within expressions no side-effects nor function calls (meaning all C light expressions always terminate and are pure)
- ▶ statements: in `for(s1, a, s2)` `s1` and `s2` are statements, that do not terminate by break
- ▶ `extern` functions are only declared and not defined, used to model system calls

Semantics of C light

Operational Semantics

Our goal is to prove that programs written in C light behave as intended. To do this we need to formalize the notion of meaning of a C light program. We do this using what is called **operational semantics**.

An operational semantics is a mathematical model of programming language execution. It is basically an interpreter defined formally.


Here I will talk about the formalization of big-step operational semantics used for all intermediate languages of CompCert.

Operational Semantics

Each syntactic element (expressions and statements) is related to the intended result of executing this element.

Expressions are deterministically mapped to memory locations or values (integers, bool etc).

The execution of statements depends on memory state and values stored in the local environment and produces **outcomes** (break, normal, return), updated memory and local environment. Moreover, **trace** of external calls is recorded².

²(cf. CompCert/cfrontend/ClightBigstep.v). 

Semantic Elements: Values

We assign primitive values to constants and then compositionally compute values of expressions and outcomes of statements.

A CompCert C value is either:

- ▶ a machine integer;
- ▶ a floating-point number;
- ▶ a pointer: a pair of a memory address and an integer offset with respect to this address;
- ▶ the `Vundef` value denoting an arbitrary bit pattern, such as the value of an uninitialized variable.

Semantic Elements: Values

Definition of values in Coq: `CompCert/common/Values.v`

```
Inductive val: Type :=  
| Vundef: val  
| Vint: int → val  
| Vlong: int64 → val  
| Vfloat: float → val  
| Vsingle: float32 → val  
| Vptr: block → ptrofs → val.
```

- ▶ float type is formalized in Flocq library
- ▶ int and ptrofs types are defined in CompCert's Integers library

Evaluation of expressions

The evaluation of constants is straightforward: map to the same integer/float value:

```
Inductive eval_expr: expr → val → Prop :=  
| eval_Econst_int:  ∀ i ty,  
  eval_expr (Econst_int i ty) (Vint i)  
| eval_Econst_float:  ∀ f ty,  
  eval_expr (Econst_float f ty) (Vfloat f)  
| eval_Econst_single:  ∀ f ty,  
  eval_expr (Econst_single f ty) (Vsingle f)  
| eval_Econst_long:  ∀ i ty,  
  eval_expr (Econst_long i ty) (Vlong i)
```

Evaluation of expressions

The evaluation of constants is straightforward: map to the same integer/float value:

```
| eval_Etempvar:  $\forall$  id ty v,  
  le!id = Some v  $\rightarrow$   
  eval_expr (Etempvar id ty) v
```

Evaluation of expressions

For unary and binary expression, the semantics of each operation is defined and applied to the values of the operands:

```
| eval_Eunop:  $\forall$  op a ty v1 v,  
    eval_expr a v1  $\rightarrow$   
    sem_unary_operation op v1 (typeof a) m = Some v  $\rightarrow$   
    eval_expr (Eunop op a ty) v
```

Evaluation of expressions

`eval_lvalue` `ge e m a b ofs` defines the evaluation of expression `a` in l-value position given global and local environments and memory `m`. The result is the memory location `[b, ofs]` that contains the value of the expression `a`.

```
eval_lvalue: expr → block → ptrofs → Prop :=  
| eval_Evar_local: ∀ id l ty,  
  e!id = Some(l, ty) →  
    eval_lvalue (Evar id ty) l Ptrofs.zero  
| eval_Evar_global: ∀ id l ty,  
  e!id = None →  
    Genv.find_symbol ge id = Some l →  
    eval_lvalue (Evar id ty) l Ptrofs.zero  
| eval_Ederef: ∀ a ty l ofs,  
  eval_expr a (Vptr l ofs) →  
    eval_lvalue (Ederef a ty) l ofs
```

Execution of statements

`exec_stmt ge e m1 s t m2 out` describes the execution of the statement `s`. `out` is the outcome for this execution. `m1` is the initial memory state, `m2` the final memory state. `t` is the trace of input/output events performed during this evaluation³

³`CompCert/cfrontend/ClightBigstep.v`

Evaluation of expressions

“Do nothing” always evaluated to normal outcome:

```
| exec_Sskip:  ∀ e le m,  
               exec_stmt e le m Sskip  
               E0 le m Out_normal
```

Setting a value v to some id results in modified temporary environment:

```
| exec_Sset:    ∀ e le m id a v,  
               eval_expr ge e le m a v →  
               exec_stmt e le m (Sset id a)  
               E0 (PTree.set id v le) m Out_normal
```


Evaluation of expressions

A more complicated example: loop. If we reached a break or return on the first (or second) statement, then the loop output is normal or return:

```
| exec_Sloop_stop1: ∀ e le m s1 s2 t le' m' out' out,  
  exec_stmt e le m s1 t le' m' out' →  
  out_break_or_return out' out →  
  exec_stmt e le m (Sloop s1 s2) t le' m' out
```

Or we loop again if both statements result in normal outcome:

```
| exec_Sloop_loop: ∀ e le m s1 s2 t1 le1 m1 out1 t2 le2  
m2 t3 le3 m3 out,  
  exec_stmt e le m s1 t1 le1 m1 out1 →  
  out_normal_or_continue out1 →  
  exec_stmt e le1 m1 s2 t2 le2 m2 Out_normal →  
  exec_stmt e le2 m2 (Sloop s1 s2) t3 le3 m3 out →  
  exec_stmt e le m (Sloop s1 s2)  
    (t1**t2**t3) le3 m3 out
```

To see operational semantics in action, see factorial example in `asn1fpcq/c/poc/factorial`.

Toy example: `strlen` function

Toy example: `strlen` function

A more interesting example that involves memory model is `strlen` function that calculates the length of a C string.

Informal Specification

From the GNU C Reference Manual:

... A string constant is of type “array of characters”. All string constants contain a null termination character as their last character.

... The `strlen()` function calculates the length of the string pointed to by `s`, excluding the terminating null byte.

... The `strlen()` function returns the number of bytes in the string pointed to by `s`.

Conforming to ... C99, C11, SVr4, 4.3BSD.

Simple C Implementation

```
#include <stddef.h>
```

```
size_t strlen(const unsigned char *s)
```

```
{
```

```
    size_t i = 0;
```

```
    while(*s++)
```

```
        i++;
```

```
    return i;
```

```
}
```

Formal Specification

Here we use relational specification. It has an advantage that we only describe what is true and we don't have to worry about termination. Moreover, we can use *inversion* to prove lemmas about the spec. Alternatively, we could specify the same function using a functional spec.

```
Inductive strlen (m : mem) (b : block) (ofs : ptrofs) : int →
Prop :=
| LengthZero: load m [b,ofs] = Some 0 → strlen m b ofs 0
| LengthSucc: ∀ (n : int) (c : char),
               strlen m b ofs + 1 n →
               load m [b,ofs] = Vint c →
               c <> Int.zero →
               n + 1 <= UINT MAX →
               strlen m b ofs n + 1.
```

Note that we are guarding against integer overflow here.

Formal Specification

```
Fixpoint strlen_fun (m : mem) (b : block) (ofs : Z) (l : int)
  (inrange : N) {struct inrange} : option int :=
  match inrange with
  | 0 ⇒ None (* out of int range *)
  | S n ⇒ match load m b ofs with
          | Some v ⇒
              if is_null v
              then Some l else strlen_fun m b (ofs + 1) (l + 1) n
          | None ⇒ None
        end
  end.
```

```
Definition strlen_fun_spec (m : mem) (b : block) (ofs : Z)
:=  strlen m b ofs 0 INTSIZE.
```


C light AST (loop of strlen)

```
Definition f_strlen_loop :=  
  (Sloop (Ssequence  
    (Ssequence  
      (Ssequence  
        (Sset _t1 (Etempvar _s (tptr tuchar)))  
        (Sset _s  
          (Ebinop 0add (Etempvar _t1 (tptr tuchar))  
            (Econst_int (Int.repr 1) tint) (tptr tuchar))))))  
      (Ssequence  
        (Sset _t2 (Ederef (Etempvar _t1 (tptr tuchar)) tuchar))  
        (Sifthenelse (Etempvar _t2 tuchar) Sskip Sbreak)))  
    (Sset _i  
      (Ebinop 0add (Etempvar _i tint) (Econst_int (Int.repr 1)  
        tint)  
        tint))) Sskip) |}.  
|}.
```

Correctness

We prove that for all strings our program computes correct result.
In particular:

Theorem

For all addresses $[b, ofs]$ where a valid C string of length len is stored, the C light AST f_strlen evaluates to len .

Lemma `strlen_correct`:

$\forall len\ m\ b\ ofs\ le, \text{strlen}\ m\ b\ ofs\ len \rightarrow \exists t\ l',$
 $le!_input = \text{Some} (Vptr\ b\ ofs) \rightarrow$
 $\text{exec_stmt}\ le\ m\ f_strlen\ t\ le'\ m (\text{Out_return} (\text{Some} (Vint\ len))).$

To prove this statement we have to prove that loop works correctly.

Correctness cont'd

Lemma `strlen_loop_correct`: $\forall \text{len } m \text{ b ofs le},$
 $\text{strlen } m \text{ b ofs len} \rightarrow \exists t \text{ le'},$
 $\text{le!_output} = \text{Some (Vint 0)} \rightarrow$
 $\text{le!_input} = \text{Some (Vptr b ofs)} \rightarrow$
 $\text{exec_stmt ge e le m f_strlen_loop t le' m Out_normal}$
 $\quad \wedge \text{le'!_output} = \text{Some (Vint len)}.$

Proof.

We prove a generalization of this statement



Lemma `strlen_loop_correct_gen`: $\forall \text{len } m \text{ b ofs le},$
 $\text{strlen } m \text{ b ofs} + i \text{ len} \rightarrow \exists t \text{ le'},$
 $\text{le!_output} = \text{Some (Vint } i) \rightarrow$
 $\text{le!_input} = \text{Some (Vptr b ofs} + i) \rightarrow$
 $\text{exec_stmt ge e le m f_strlen_loop t le' m Out_normal}$
 $\quad \wedge \text{le'!_output} = \text{Some (Vint len} + i).$

by int-induction on *len* and *i*.



Conclusion

Thus we have proved that on all strings of length smaller than `UINT_MAX`, `strlen` works correctly.

-  Sandrine Blazy and Xavier Leroy, Mechanized Semantics for the Clight Subset of the C Language, 2009
-  Xavier Leroy and Sandrine Blazy and Gordon Stewart, The CompCert Memory Model, Version 2, 2012