Proving C program correct using C light operational semantics

Outline

1. Formal verification - quick intro (high-level)
2. Coq mini intro
3. Approach
   - Particular approach we consider: reasoning about C programs in Coq
   - Base PL concepts mini intro: syntax, AST, semantics.
4. Toy example: strlen Informal specification (man page)
   - Formal specification of strlen (relational)
   - Simple implementation in C
   - From C program to AST using clightgen
   - Semantics of C program semantics and its equivalence to specification
   - Undefined behaviours in C and guarding against them
5. Conclusions

We want to have high assurance that our code works as intended. One of the methods is formal verification. It is a broad term that includes many techniques. Here I will talk about deductive 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?

On one hand we have C implementation of some function, on the other hand we have our ideas about what it supposed to do – its specification. To formally verify some function we need to:

1. Write it's specification in some formal language
2. Write the implementation in the same formal language
3. Formalize the notion of "meeting the specification" (partial correctness, total correctness)
4. Prove that your implementation meets the specification

CompCert example

Coq intro

Explain what was done before: disadvantages and advantages of purely functional approach (Illya)

- reason about the actual implementation
- parse C code into an abstract syntax tree using C light generator of CompCert (not verified)
- reason about the C light program using operational semantics

Concrete vs Abstract syntax
We write a C program in concrete C syntax, which is designed to be used by a parser (a $+$ b). Abstract syntax tree: nodes are constructors, leaves are atoms (plus (a,b)). todo: more on AST
Deep embedding of C light to Coq $:=$ the abstract syntax is defined as inductive datatypes

C light syntax

types

Expressions of C light

```
Inductive expr : Type :=
| Econst_int: int → type → expr (* integer literal *)
| Econst_float: float → type → expr (* double float literal *)
| Econst_single: float32 → type → expr (* single float *)
| Econst_long: int64 → type → expr (* long integer literal *)
| Evar: ident → type → expr (* variable *)
| Etempvar: ident → type → expr (* temporary variable *)
| Ederef: expr → type → expr (* pointer dereference (*) *)
| Eaddrof: expr → type → expr (* address-of operator (&) *)
| Eunop: unary_operation → expr → type → expr
(* unary operation *)
| Ebinop: binary_operation → expr → expr → type → expr
(* binary operation *)
| Ecast: expr → type → expr (* type cast *)
| Efield: expr → ident → type → expr
(* access to a member of a struct or union *)
| Esizeof: type → type → expr (* size of a type *)
| Ealignof: type → type → expr. (* alignment of a type *)
```

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)
```

Note that in C light all expressions are **pure**. Variable assignements and function calls are statements.

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_λ ction → typelist → list ex
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
```

Statements

```
Definition Swhile (e: expr) (s: statement) :=
  Sloop (Ssequence (Sifthenelse e Sskip Sbreak) s) Sskip.

Definition Sdowhile (s: statement) (e: expr) :=
  Sloop s (Sifthenelse e Sskip Sbreak).

Definition Sfor (s1: statement) (e2: expr) (s3: statement) (s4: sta
Ssequence s1 (Sloop (Ssequence (Sifthenelse e2 Sskip Sbreak) s3) s4
```

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

there are more - see p. 2-7 of Mechanized Sem. for details. (TODO)

Limitations

Partial correctness (safety? liveness?)

Our goal is to prove that programs written in C light behave as intented. 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. We start from assigning primitive values to constants and then compositionally assign values to expressions and statement.

- final result of a program execution - trace of invocation of external functions - deterministic (since expressions are pure) Evaluation done in a context with global vars (G), local vars (E) and memory state (M). Rules described in Fig.6-10. of Mech Sem

A CompCert C value is either[1]:

- 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.

[1] This is a common semantics used for all intermediate languages of CompCert, such as C minor etc.

Values

```
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

Integers

Formalizations of machine integers modulo 2^N defined as a module type in CompCert `lib/Integers.v`.

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 }.
```

8, 32, 64-bit integers are supported, as well as 32 and 64-bit pointer offsets.

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.

□

Memory model

defined in CompCert `common/Memory.v`

a type `mem` of memory states, the following 4 basic operations over memory states, and their properties:

- **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.

C light semantics

We define evaluation relation between statements of C light and CompCert C values

Informal spec: strlen

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.

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

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

CONFORMING TO
POSIX.1-2001, POSIX.1-2008, C89, C99, C11, SVr4, 4.3BSD.

To formalize the spec we need a formal model of C integers, pointers and memory model

Formal 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 <= MaxInteger →
                    strlen m b ofs n + 1.
```

From C program to AST using clightgen

```
#include <stddef.h>

size_t strlen(const unsigned char *s)
{
  size_t i = 0;

  while(*s++)
      i++;

  return i;
}
```

C light AST (loop of strlen)

```
Definition f_strlen_loop := {|
fn_params := ((_s, (tptr tuchar)) :: nil);
fn_temps := ((_i, tuint) :: (_t1, (tptr tuchar)) :: (_t2, tuchar) :
fn_body :=
(Sloop
(Ssequence
(Ssequence
(Ssequence
  (Sset _t1 (Etempvar _s (tptr tuchar)))
  (Sset _s
    (Ebinop Oadd (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 Oadd (Etempvar _i tuint) (Econst_int (Int.repr 1)
 tint)
  tuint)))
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:
∀ len m b ofs le, strlen m b ofs len → ∃ t l',
  le!_input = Some (Vptr b ofs) →
  exec_stmt le m f_strlen t le' m (Out_return (Some (Vint len))).
```

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

Correctness cont'd

```
Lemma strlen_loop_correct: ∀ len m b ofs le,
strlen m b ofs len → ∃ t le',
le!_output = Some (Vint 0) →
le!_input = Some (Vptr b ofs) →
exec_stmt ge e le m f_strlen_loop t le' m Out_normal
                          ∧ le'!_output = Some (Vint len).
```

Proof.

We prove a generalization of this statement

```
Lemma strlen_loop_correct_gen : ∀ len m b ofs le,
strlen m b ofs + i len → ∃ t le',
le!_output = Some (Vint i) →
le!_input = Some (Vptr b ofs + i) →
exec_stmt ge e le m f_strlen_loop t le' m Out_normal
                          ∧ le'!_output = Some (Vint len + i).
```

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

□

Conclusion

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