Formal Verification of Computer Programs

A Primer. (part 1)

Vadim Zaliva [1] Nika Pona [2]

[1]Carnegie Mellon University

[2]Digamma.ai

Abstract

In this presentation we will talk about formal verification of programs written in C using Coq proof assistant. We will introduce Coq's basic functionality and on a toy example show how to use it to verify functional and imperative programs, and how the two approaches differ. We will also show a motivational example of verifying a function from existing C code.

What are formal methods?

Formal verification

We want to have high assurance that the code we wrote works as intended and is bug-free. One of the methods to do this is **formal verification**, which amounts to producing *a formal proof* of correctness.

What does it mean and how do we do it?

1. We write the *specification* in *a formal language* which unambiguously defines how our program should behave.
2. Then we model our program and its actual behaviour, that is, we define the *semantics* of our program.
3. Finally we mathematically prove that the behaviour of our program matches the specification.

In what follows we will talk about formal verfication using **Coq proof assistant**.

Verifying C code: a motivating example

`asn_strtoimax_lim` of ASN.1C compiler

At `Digamma.ai` we are working on formal verification of existing **imperative** programs using Coq. We took a function `asn_strtoimax_lim` from `asn1c` compiler to test our approach on a simple real-life example.

Informal specification from the comments:

> *Parse the number in the given string until the given *end position, returning the position after the last parsed character back using the same (*end) pointer. WARNING: This behavior is different from the standard strtol/strtoimax(3).*

asn_strtoimax_lim **code**

```
enum asn_strtox_result_e
asn_strtoimax_lim(const char *str, const char **end, intmax_t *intp) {
        int sign = 1;
        intmax_t value;

#define ASN1_INTMAX_MAX ((~(uintmax_t)0) >> 1)
    const intmax_t upper_boundary = ASN1_INTMAX_MAX / 10;
        intmax_t last_digit_max = ASN1_INTMAX_MAX % 10;
#undef  ASN1_INTMAX_MAX

        if(str >= *end) return ASN_STRTOX_ERROR_INVAL;

        switch(*str) { case '-':
                        last_digit_max++;
                        sign = -1;
                        /* FALL THROUGH */
                        case '+':
                          str++;
                          if(str >= *end) {
                            *end = str;
                            return ASN_STRTOX_EXPECT_MORE; } }

        for(value = 0; str < (*end); str++) {
                switch(*str) {
                case 0x30: case 0x31: case 0x32: case 0x33: case 0x34:
                case 0x35: case 0x36: case 0x37: case 0x38: case 0x39: {
                        int d = *str - '0';
                        if(value < upper_boundary) {
                                value = value * 10 + d;
                        } else if(value == upper_boundary) {
                                if(d <= last_digit_max) {
                                        if(sign > 0) { value = value * 10 + d;
                                        } else { sign = 1;
                                                 value = -value * 10 - d; }
                                } else { *end = str;
                                         return ASN_STRTOX_ERROR_RANGE; }
                        } else { *end = str;
                                 return ASN_STRTOX_ERROR_RANGE; } }
                    continue;
                default:
                    *end = str;
                    *intp = sign * value;
                    return ASN_STRTOX_EXTRA_DATA; } }
        *end = str;
        *intp = sign * value;
        return ASN_STRTOX_OK; }
```

Negative range bug example

When we go beyond allowed *int* range, a wrong result is given for some inputs[1]:

input	-128
intmax	127
upper boundary	12
last digit max	7
return	`ASN_STRTOX_ERROR_RANGE`
input	-1281
intmax	127
upper boundary	12
last digit max	7
return	`-127, ASN_STRTOX_OK`

[1] Assume we are working on a 8-bit system and maximal signed int `MAX_INT` is 127

Negative range bug

This happens whenever the input string represents a number smaller than `MIN_INT`, due to the fact that absolute value of `MIN_INT` is greater than `MAX_INT`, thus negative number cannot be treated as $\mathrm{value} \times \mathrm{sign}$ when value is represented as *int*.

Formal proof has to cover all cases, hence this bug became obvious during the proof.

A bug uncovered during verification

The bug (#344) was filed and promptly fixed by developers:

```
--- asn_strtoimax_lim_old.c     2019-09-11 10:18:00.013478144 -0700
+++ asn_strtoimax_lim.c 2019-09-11 10:18:00.013478144 -0700
@@ -33,15 +28,21 @@
                                            if(sign > 0) { value = value * 10 + d;
                                            } else { sign = 1;
                                                     value = -value * 10 - d; }
+                        str += 1;
+                        if(str < *end) {
+                            // If digits continue, we're guaranteed out of range.
+                            *end = str;
+                            if(*str >= 0x30 && *str <= 0x39) { return ASN_STRTOX_ERROR_RANGE;
+                            } else { *intp = sign * value;
+                                      return ASN_STRTOX_EXTRA_DATA; }}
+                        break;
+                    } else { *end = str;
+                             return ASN_STRTOX_ERROR_RANGE; }
                                    } else { *end = str;
                                             return ASN_STRTOX_ERROR_RANGE; }
                            } else { *end = str;
-                                     return ASN_STRTOX_ERROR_RANGE; } }
-                        continue;
-                    default:
-                        *end = str;
                         *intp = sign * value;
-                        return ASN_STRTOX_EXTRA_DATA; } }
+                     return ASN_STRTOX_EXTRA_DATA; }}
            *end = str;
            *intp = sign * value;
            return ASN_STRTOX_OK; }
```

asn_strtoimax_lim **fixed**

```c
for(value = 0; str < (*end); str++) {
    if(*str >= 0x30 && *str <= 0x39) {
        int d = *str - '0';
        if(value < upper_boundary) {
            value = value * 10 + d;
        } else if(value == upper_boundary) {
            if(d <= last_digit_max) {
                if(sign > 0) { value = value * 10 + d;
                } else { sign = 1;
                    value = -value * 10 - d; }
                str += 1;
                if(str < *end) {
                    // If digits continue, we're guaranteed out of range.
                    *end = str;
                    if(*str >= 0x30 && *str <= 0x39) { return ASN_STRTOX_ERROR_RANGE;
                    } else { *intp = sign * value;
                        return ASN_STRTOX_EXTRA_DATA; }}
                break;
            } else { *end = str;
                return ASN_STRTOX_ERROR_RANGE; }
        } else { *end = str;
            return ASN_STRTOX_ERROR_RANGE; }
    } else { *end = str;
        *intp = sign * value;
        return ASN_STRTOX_EXTRA_DATA; }}
*end = str;
*intp = sign * value;
return ASN_STRTOX_OK; }
```

2nd bug uncovered in fixed version

Is this fix OK? Look at this part of the code:

```
if(str < *end) {
  *end = str;
  if(*str >= 0x30 && *str <= 0x39){
    return ASN_STRTOX_ERROR_RANGE;
  } else {
    *intp = sign * value;
    return ASN_STRTOX_EXTRA_DATA;
  }
}
```

Memory store bug explained (1/3)

Let minimal signed int $\texttt{MIN\_INT} = -4775808$

$*\texttt{str}$ = 2d 34 37 37 35 38 30 31 31 31 (stands for "-477580111")

Scenario 1:

Assume that `*end = str + 9` and end $\geq$ `str + 9`.

$$\text{2d 34 37 37 35 38 30 } \overbrace{31}^{\texttt{str + 7}} \text{ 31 31} \ldots \overbrace{\texttt{X}}^{\texttt{end}}$$

Then at `str + 7` we store `*end = (str + 7)`

Let `str + 7 = 21 21 21 26`

$$\text{2d 34 37 37 35 38 30 31 } \overbrace{31}^{\texttt{str + 8}} \text{ 31} \ldots \overbrace{\text{21 21 21 26}}^{\texttt{end}}$$

And since at `str + 8` we read '1'

The output is `ASN_ERROR_RANGE`.

Memory store bug explained (2/3)

Let minimal signed int MIN_INT $= -4775808$

`*str` = 2d 34 37 37 35 38 30 31 31 31 (stands for "-477580111")

Scenario 2:

Assume that `*end = str + 9` and `end = str + 7`:

$\text{2d 34 37 37 35 38 } \overbrace{\text{30 31 31 31}}^{\texttt{end}}$

Then at `str + 7` we store `*end = str + 7`

Let `str + 7 = 21 21 21 26`

$\text{2d 34 37 37 35 38 30 } \overbrace{\text{21 21 21 26}}^{\texttt{end}}$ (stands for "$-$477580!!!&")

And since at `str + 8` we read '!'

The output is `ASN_EXTRA_DATA`.

Memory store: A bug or an implicit restriction?

We have demonstrated that when the value of the `end` pointer is treated as a part of the input data, there is a bug where the resulting error value could be incorrect.

On the other hand, it is hard to think of a legitimate use-case where the pointer would be a part the input data. Under such interpretation, there is an implicit pre-condition in the specification, mandating that:

```
(*end < end) || (end + sizeof(const char *) <= str)
```

Specification question

After addressing the two buges we discovered we were able to successfully verify that the function finally corresponds to the specification we wrote for. However, it was noticed the following behavior:

For input `"a"` it returns `0,ASN_STRTOX_EXTRA_DATA`.

Is it a bug or a feature?

Lessons learned

This code was part of the library for 15 years. The library is covered by extensive unit and randomized tests. It is used in production by multiple users. Yet, the vulnerabilities are there and pose potential problems.

1. The first bug is related to data type ranges and modulo integer arithmetic. These sort of problems are fairly common and require careful coding to be avoided. Formal verification enforces a strict mathematical model of all computer arithmetic and invariably exposes all such bugs.
2. The second problem was related to *pointer aliasing*. These problems are not immediately obvious because C language does not allow us to enforce any memory aliasing restrictions (unlike, say Rust). In formal verification, there is a rigorous model to analyze such kind of problems called *separation logic*.
3. The third issue shows us that your formal verification is only as good as your specification.

Coq mini-intro

Coq mini-intro

We did our proofs in a formal language called *Gallina*, a mechanized version of **Calculus of Inductive Constructions**, which is a very expressive type theory well studied in mathematical logic. We write the specifications, model our programs and do the proofs in this language.

We could do all of the above on paper, but it would quickly get out of hand. Moreover, we want to be sure that there are no mistakes in the proofs. So we use a tool called **proof assistant**: a program that checks that your proof is correct. It also provides an environment to make the construction of proofs easier.

In particular, we will talk about the Coq proof assistant: `https://coq.inria.fr/`.

What Coq does?

In Coq you can:

- define functions and predicates
- state mathematical theorems and software specifications
- interactively develop formal proofs of theorems
- machine-check these proofs by a relatively small trusted kernel
- extract certified programs to languages like OCaml, Haskell or Scheme.

Verifying C code: a detailed example

Factorial example

One can mathematically specify factorial as a recursive equation, for $(0 \leq n)$:

$$fact(0) = 1$$
$$fact(n+1) = fact(n) * (n+1)$$

We can write it in Coq as a fixpoint definition:

```
Fixpoint fact (n : ℕ) : ℕ :=
  match n with
  | O ⇒ 1
  | S n' ⇒ n * fact n'
  end.
```

Note that this definition is also a functional program. This corresponds to the idea of verifying a program wrt reference implementation. The program evidently corresponds to our mathematical spec on paper, so we can use this approach here. But for more complex program we will want to write a specification in a more declarative fashion.

Factorial example

We can also use an inductive definition of factorial. For $(0 \leq n)$:

$$fact(0) = 1 \tag{1}$$

$$\text{If } fact(n) = m \text{ then } fact(n+1) = m * (n+1) \tag{2}$$

In Coq it corresponds to an inductive type or a predicate on natural numbers:

```
Inductive factorial : ℕ → ℕ→ Prop :=
| FactZero : factorial 0 1
| FactSucc : ∀ n m, factorial n m → factorial (S n) ((S n)*m).
```

Think of `FactZero` and `FactSucc` as axioms or rules that define what factorial is.

More declarative specs

Another simple example: sorting. We can write an inductive spec as before:

```
Inductive sorted: list ℕ → Prop :=
| sorted_nil: sorted nil
| sorted_1: ∀ x, sorted (x :: nil)
| sorted_cons: ∀ x y l, x ≤ y → sorted (y::l) → sorted (x::y :: l ).
```

Alternatively:

```
Definition sorted (al: list ℕ) :=
 ∀ i j, i < j < length al → al[i] ≤ al[j].
```

Then we can go on and prove that your favorite sorting algorithm's output is `sorted`[2].

[2] and a permutation of the input

Factorial example: verifying a functional program

We can write a tail-recursive functional program to compute factorial

```
Fixpoint fact_acc (n : ℕ) (acc : ℕ) :=
  match n with
  | 0 ⇒ acc
  | S k ⇒ fact_acc k (n * acc)
  end.

Definition fact' (n : ℕ) :=
  fact_acc n 1.
```

Now we want to show that it actually computes factorial. To do this we can show in Coq that:

```
Theorem fact'_correct : ∀ n, fact' n = fact n.
```

```
Theorem fact'_correct_R : ∀ n, factorial n (fact' n).
```

Factorial example: verifying a functional program

Now using Coq's extraction mechanism we can automatically extract an OCaml or Haskell function that is provably correct.

Alternatively, one could easily embed a functional language into Coq and reason about the existing implementation in a similar fashion.

But what if you want to verify code written in imperative language? Things get *slightly* more complicated.

Factorial example: verifying a C program

To be able to state theorems about C programs in Coq we need to somehow represent C functions in Coq. This means to model C syntax (as abstract syntax trees) and semantics (execution of programs) in Coq[3].

Luckily, this has already been done in the project called CompCert.

[3] For formalization of C semantics see [Blazy and Leroy, 2009]

CompCert

CompCert is a verified compiler for C, almost entirely written in Coq and proved to work according to its 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)

Verifying imperative programs

We use CompCert and use the following approach to verifying C programs:

- Parse C code into a Coq abstract syntax tree using C light[4] generator of CompCert
- Write a functional specification in Coq, using CompCert's memory model and integer representations
- Prove properties about the generated AST using semantics of C (light) defined in CompCert

[4] C light is a subset of C

Unsupported features in C light

- `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

Factorial example: verifying a C program

Factorial C implementation that we want to verify

```
unsigned int factorial (unsigned int input) {
  unsigned int output = 1;
  while (input) {
      output = output*input ;
      input = input - 1 ;
    }
  return output ;
}
```

The specification stays the same.

Syntax of C programs in Coq

Our C function can be represented in Coq as an abstract syntax tree:

```
(Ssequence
  (* int output = 1 *)
  (Sset _output (Econst_int (Int.repr 1) tuint))
  (Ssequence
    (* while (input) *)
    (Swhile (Etempvar _input tuint)
      (Ssequence
      (* output = output*input *)
        (Sset _output
          (Ebinop Omul (Etempvar _output tuint)
          (Etempvar _input tuint) tuint))
          (* input = input - 1 *)
         (Sset _input (Ebinop Osub (Etempvar _input tuint)
                     (Econst_int (Int.repr 1) tuint) tuint))))
            (* return output *)
            (Sreturn (Some (Etempvar _output tuint))))).
```

Clight Expressions: Examples

Expressions are annotated with types:

```
(* constant 0 of type int *)
(* 0 *)
(Econst_int (Int.repr 0) tint)

(* binary operration add applied to constants 0 and 1 *)
(* 0 + 1 *)
(Ebinop Oadd (Econst_int (Int.repr 0) tint)
(Econst_int (Int.repr 1) tint) (tint))

(* temporary variable of integer pointer type *)
(* int *p *)
(Etempvar _p (tptr tint))

(* dereferencing integer pointer *)
(* (*p) *)
(Ederef (Etempvar _p (tptr tint)) tint)
```

Clight Statements: Examples

```
(* int s = 1 *)
(Sset _s (Econst_int (Int.repr 1) tint))

(* return s *)
(Sreturn (Some (Etempvar _s tint)))

(* int s = 1 ;
   int t = 0 ; *)
(Ssequence
  (Sset _s (Econst_int (Int.repr 1) tint))
  (Sset _t (Econst_int (Int.repr 0) tint)))

(* while (s) { s = s - 1 } *)
(Swhile (Etempvar _s tint)
(Sset _s (Ebinop Osub (Etempvar _input tint)
           (Econst_int (Int.repr 1) tint) tint))))
```

Operational Semantics

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

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

We use big-step operational semantics used for all intermediate languages of CompCert.

Operational Semantics

The idea is to assign primitive values to constants and then compositionally compute values of expressions and results of execution of statements.

The evaluation of expressions and execution of statements is done in the context of global and local environments and memory state.

- Expressions are mapped to memory locations or values (integers, bool etc).
- The execution of statements produces **outcomes** (break, normal, return), an updated memory and local environment.

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 :=
      {| intval: Z;
         intrange: -1 < intval < modulus |}.
```

C memory model

In order to verify a program written in C, one has to have a good model of variable environments, integer and pointer arithmetic and memory model.

A **memory model** is a specification of memory states and operations over memory.

In CompCert, 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 ranging from allowing all operations (read, write etc.) to allowing no operation.

C memory model

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

- **load** : read memory at a given address;
- **store** : store memory at a given address;
- **alloc** : allocate a fresh memory block;
- **free** : invalidate a memory block.

These operations are to satisfy some basic properties like: `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.

Examples

Expression (`Econst_int Int.zero tint`) is evaluated to value 0 in any local environment and memory.

$$le, m, (\texttt{Econst\_int Int.zero tint}) \Rightarrow 0, le, m$$

Evaluation of statement (`Sset _s (Econst_int Int.zero tint)`) in local environment *le* and memory *m* produces new local environment *le*′ with `_s` mapped to value 0 and a **normal** outcome.

$$le, m, (\texttt{Sset \_s (Econst\_int Int.zero tint)}) \Rightarrow le\{\_s = 0\}, m, \texttt{normal}$$

Statement (`Sreturn (Some (Etempvar _s tint))`) evaluates to a **return**(`s`) outcome and leaves *le* and memory unchanged.

$$le, m, (\texttt{Sreturn (Some (Etempvar \_s tint))}) \Rightarrow le, m, \texttt{return(s)}$$

Correctness statement

Now we can state the correctness theorem for factorial:

For any memory m and local environments le with variables $\mathtt{input}$ assigned n in le, execution of $\mathtt{f\_factorial}$ terminates and returns $fact(n)$ with resulting memory $m' = m$.

Theorem
$\forall le\ m,$

$le\{\mathtt{input\ =\ n}\} \rightarrow$

$\exists le', le, m, \mathtt{factorial} \Rightarrow le', m, \mathbf{return}(\mathtt{fact\ \ n})$

Hence we proved that factorial works correctly on all inputs[5].

[5] For simplicity, here we also assume that (fact n) doesn't overflow

Conclusion

Coq can be used to prove correctness of imperative programs, as well as functional ones. However, the former requires an additional step of embedding C syntax and semantics in Coq.

Going back to our first example of `asn_strtoimax_lim`:

- We wrote a formal specification of the function (based on the comment and analysis of the function)
- Then we produced Clight AST of the function using Clight generator of CompCert
- And proved that the resulting AST evaluates to correct values on all valid inputs using operational semantics.

Moreover, using CompCert's C memory model we can state properties about correct memory usage and heap and stack bounds.

Other languages

Other languages

JSCert : certified JavaScript.

RustBelt : formal (and machine-checked) safety proof for a language representing a realistic subset of Rust.

Vellvm : a framework for reasoning about programs expressed in LLVM's intermediate representation and transformations that operate on it.

CakeML : is a functional programming language and an ecosystem of proofs and tools built around the language. The ecosystem includes a proven-correct compiler that can bootstrap itself.

Questions?

Examples from this presentation:

`https://github.com/digamma-ai/formal-verification-intro`

Contact:

- Vadim Zaliva, ✉ vzaliva@cmu.edu, @vzaliva
- Nika Pona, ✉ npona@digamma.ai

References i

Blazy, S. and Leroy, X. (2009).
Mechanized Semantics for the Clight Subset of the C Language.