Formal Verification of Computer Programs

Silicon Valley Deep Specification Meetup

Nika Pona [2] Vadim Zaliva [1]

[1]Carnegie Mellon University

[2]Digamma.ai

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 of imperative programs using **Coq proof assistant**.

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 functional code

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 $(0 \leq n)$:

$$fact(0) = 1 \qquad (1)$$

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

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*[1].

[1] 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.

Verifying C code

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[2].

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

[2] 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[3] 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

[3] 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

```c
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.

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

Correctness statement

Now we can state the correctness theorem for factorial:

> *For any memory m and local environments le with variables* `input` *assigned n in le, execution of* `f_factorial` *terminates and returns* $fact(n)$ *with resulting memory* $m' = m$.

Theorem
$\forall le\ m,$

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

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

Hence we proved that factorial works correctly on all inputs[4].

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

Verifying C code: a real-life 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 this 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[5]:

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*

[5] 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 $MIN\_INT = -4775808$

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

Scenario 1:

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

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

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

Let $str + 7 = 21\ 21\ 21\ 26$

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

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 $end = str + 7$:

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

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

Let $str + 7 = 21\ 21\ 21\ 26$

2d 34 37 37 35 38 30 $\overbrace{\text{21 21 21 26}}^{end}$ (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.

Verified Software Toolchain

We did the correctness proof using separation logic defined on top of CompCert's operational semantics using Verified Software Toolchain (VST, `https://github.com/PrincetonUniversity/VST`). Let's go back to the `asn_strtoimax_lim` example.

Separation Logic

Given propositions P, Q and a statement c, a separation logic triple

$$P\{c\}Q$$

states that given the precondition P, the execution of the function c terminates with the post-condition Q being true. Separation logic is defined by giving rules for valid triples. Some simple rules:

$$P\{x := a\}P(x := a)$$

$$P\{\texttt{Sskip}\}P$$

$P\{\texttt{Ssequence s1 s2}\}Q$, if $P\{\texttt{s1}\}P'$ and $P'\{\texttt{s2}\}Q$ for some P'

More complex rules include the ones dealing with memory.

VST specification

To show C implementation correctness wrt the executable spec we prove a separation logic triple

$$P\{c\}Q$$

that given the precondition P, the execution of the C light function c terminates with the post-condition Q being true. The post-condition says that c returns the value according to the executable spec.

VST spec, encoder pre- and post-condition

The memory specification uses spatial predicates `data_at` *sh v p* (“at address p there is a value v with premission sh”).

We can combine the predicates using the separating conjuction $*$: each such conjunct is true on a separate sub-heap of the memory, thus guaranteeing non-overlapping of pointers.

The precondition relates the C types to the abstract types of Coq.

In the post-condition, we use the executable specification to state that the correct result is written in memory.

asn_strtoimax_lim: functional specification (abstract)

```
Fixpoint Z_of_string_loop (s : list byte) (val : Z) (b : 𝔹) :=
   match s with
   | [] ⇒ val
   | c :: tl ⇒
     if is_digit c
     then let val' := v * 10 + (Z_of_char c) in
         Z_of_string_loop tl val' b
     else val
   end.

Definition Z_of_string (s : list byte) :=
     match s with
     | [] ⇒ None
     | [c] ⇒ if is_sign c
             then None
             else Z_of_char c
     | c :: tl ⇒ if (c = plus_char)
                then Z_of_string_loop tl 0
                else if (c = minus_char)
                    then - Z_of_string_loop tl 0
                    else Z_of_string_loop s 0
     end.
```

asn_strtoimax_lim: functional specification (with bounds and errors)

```
Fixpoint Z_of_string_loop (s : list byte) (val i : Z) (b : 𝔹) :=
   match s with
   | [] ⇒ {| OK; val; i |}
   | c :: tl ⇒
     if is_digit c
     then let val' := if b then v * 10 + (Z_of_char c)
                         else v * 10 - (Z_of_char c) in
         if (Int64.min_signed <= val' <= Int64.max_signed)
         then Z_of_string_loop tl val' (i + 1) b
         else {| ERROR_RANGE; val'; i; |}
     else {| EXTRA_DATA; val; i; |}
   end.

Definition Z_of_string (s : list byte) :=
     match s with
     | [] ⇒ {| ERROR_INVAL; 0; 0 |}
     | [c] ⇒ if is_sign c
             then {| EXPECT_MORE; 0; 1 |}
             else Z_of_string_loop s 0 0 true
     | c :: tl ⇒ if (c = plus_char || c = minus_char)
                 then Z_of_string_loop tl 0 1 (𝔹_of_sign c)
                 else Z_of_string_loop s 0 0 true
     end.
```

asn_strtoimax_lim: VST precondition

```
Definition asn_strtoimax_lim_vst_spec : ident * funspec :=
WITH ls : list byte, strp endp intp endp' : val
PRE [tptr tschar, tptr (tptr tschar), tptr tlong]
    PROP (readable_share r;
          writable_share wr;
          0 <= str + |ls| < Ptrofs.max)
    LOCAL (temp _str strp;
           temp _end endp;
           temp _intp intp)
    SEP (data_at r (tarray tschar |ls|) ls strp;
         data_at wr (tptr tschar) endp' endp ;
         data_at_ wr (tint) intp)
```

asn_strtoimax_lim: VST post-condition

```
POST [tint]
let r := res (Z_of_string ls) in
let v := value (Z_of_string ls) in
let i := index (Z_of_string ls) in
    PROP()
    LOCAL (temp ret_temp r)
    SEP(data_at r (tarray tschar |ls|) strp;
        match r with
          | OK | EXTRA_DATA ⇒ data_at wr (tlong) v intp
          | _ ⇒ data_at_ wr (tlong) intp
        end;
        if (strp < endp)
        then data_at wr (tptr tschar) (strp + i) endp
        else data_at wr (tptr tschar) endp' endp).
```

VST proof

The proof is done using so-called *forward simulation*. To prove $P\{c\}Q$:

- start assuming the precondition P
- sequentially execute statements of the function c
- each statement generates a post-condition that follows form its execution
- after executing the last statement of c, prove that the post-condition Q holds.

VST provides tactics to do most of these steps automatically. One has to provide joint postconditions for if statements and loop invariant for the loop

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.

So to prove an existing C function correct:

- Write a formal specification of the function (based on informal spec, we had to use the comment and analysis of the function)
- Then produce Clight AST of the function using Clight generator of CompCert
- Prove that the resulting AST evaluates to correct values on all valid inputs. The proof can be done directly using operational semantics or using separation logic, which is defined on top of the operational semantics.

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

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