

# Formal Verification of Computer Programs

## A Primer

---

Vadim Zaliva <sup>1</sup>   Nika Pona <sup>2</sup>

<sup>1</sup>Carnegie Mellon University

<sup>2</sup>Digamma.ai

# Table of contents

1. What are formal methods?
2. Verifying C code: a motivating example
3. Coq mini-intro
4. How does Coq work?
5. Verifying C code: a detailed example
6. Other languages

**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 verification using **Coq proof assistant**.

## **Verifying C code: a motivating example**

---

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

```
1
2 enum asn_strtox_result_e
3 asn_strtoimax_lim(const char *str, const char **end, intmax_t *intp) {
4     int sign = 1;
5     intmax_t value;
6
7     #define ASN1JINTMAX_MAX ((~(uintmax_t)0) >> 1)
8     const intmax_t upper_boundary = ASN1JINTMAX_MAX / 10;
9     intmax_t last_digit_max = ASN1JINTMAX_MAX % 10;
10    #undef ASN1JINTMAX_MAX
11
12    if(str >= *end) return ASN_STRTOX_ERROR_INVALID;
13
14    switch(*str) { case '-':
15        last_digit_max++;
16        sign = -1;
17        /* FALL THROUGH */
18        case '+':
19            str++;
20            if(str >= *end) {
21                *end = str;
22                return ASN_STRTOX_EXPECT_MORE; } }
23
24    for(value = 0; str < (*end); str++) {
25        switch(*str) {
26            case 0x30: case 0x31: case 0x32: case 0x33: case 0x34:
27            case 0x35: case 0x36: case 0x37: case 0x38: case 0x39: {
28                int d = *str - '0';
29                if(value < upper_boundary) {
30                    value = value * 10 + d;
31                } else if(value == upper_boundary) {
32                    if(d <= last_digit_max) {
33                        if(sign > 0) { value = value * 10 + d;
34                        } else { sign = 1;
35                        value = -value * 10 - d; }
36                    } else { *end = str;
37                        return ASN_STRTOX_ERROR_RANGE; }
38                    } else { *end = str;
39                        return ASN_STRTOX_ERROR_RANGE; } }
40        continue;
41        default:
42            *end = str;
43            *intp = sign * value;
44            return ASN_STRTOX_EXTRA_DATA; } }
45    *end = str;
46    *intp = sign * value;
47    return ASN_STRTOX_OK; }
```

During correctness proof, some proof obligations about the for loop were produced that could not be met, because of a bug in the function. Can you see it?

<https://github.com/vlm/asn1c/issues/344>



When we go beyond allowed int range, a false result is given on some inputs:

input: -9223372036854775809  
intmax: 9223372036854775807  
upper boundary: 922337203685477580  
last digit max: 7  
ASN\_STRTOX\_ERROR\_RANGE

input: -9223372036854775810  
intmax: 9223372036854775807  
upper boundary: 922337203685477580  
last digit max: 7  
output: 70  
ASN\_STRTOX\_OK

# asn\_strtoimax\_lim fixed

```
1  enum asn_strtox_result_e
2  asn_strtoimax_lim(const char *str, const char **end, intmax_t *intp) {
3      int sign = 1;
4      intmax_t value;
5
6      const intmax_t asnl_intmax_max = ((~(uintmax_t)0) >> 1);
7      const intmax_t upper_boundary = asnl_intmax_max / 10;
8      intmax_t last_digit_max = asnl_intmax_max % 10;
9
10     if(str >= *end) return ASN_STRTOX_ERROR_INVALID;
11
12     switch(*str) { case '-':
13         last_digit_max++;
14         sign = -1;
15         /* FALL THROUGH */
16     case '+':
17         str++;
18         if(str >= *end) { *end = str;
19             return ASN_STRTOX_EXPECT_MORE; }}
20
21     for(value = 0; str < (*end); str++) {
22         if(*str >= 0x30 && *str <= 0x39) {
23             int d = *str - '0';
24             if(value < upper_boundary) {
25                 value = value * 10 + d;
26             } else if(value == upper_boundary) {
27                 if(d <= last_digit_max) {
28                     if(sign > 0) { value = value * 10 + d;
29                         } else { sign = 1;
30                             value = -value * 10 - d; }
31                     str += 1;
32                     if(str < *end) {
33                         // If digits continue, we're guaranteed out of range.
34                         *end = str;
35                         if(*str >= 0x30 && *str <= 0x39) { return ASN_STRTOX_ERROR_RANGE;
36                             } else { *intp = sign * value;
37                                 return ASN_STRTOX_EXTRA_DATA; }}
38                     break;
39                 } else { *end = str;
40                     return ASN_STRTOX_ERROR_RANGE; }
41             } else { *end = str;
42                 return ASN_STRTOX_ERROR_RANGE; }
43         } else { *end = str;
44             *intp = sign * value;
45             return ASN_STRTOX_EXTRA_DATA; }}
46     *end = str;
47     *intp = sign * value;
48     return ASN_STRTOX_OK; }
```

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

```
1  if(str < *end) {  
2      *end = str;  
3      if(*str >= 0x30 && *str <= 0x39){  
4          return ASN_STRTOX_ERROR_RANGE;  
5      } else {  
6          *intp = sign * value;  
7          return ASN_STRTOX_EXTRA_DATA;  
8      }  
9  }
```

# Memory store bug

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` points outside of `*str`.

`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`.

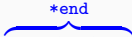
# Memory store bug

Let minimal signed int `MIN_INT = -4775808`

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

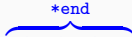
## Scenario 2:

Assume that `*end = *(str + 9)` and end points inside of `*str`:

`2d 34 37 37 35 38`  `30 31 31 31`

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

Let `(str + 7) = 21 21 21 26`

`2d 34 37 37 35 38 30`  `21 21 21 26` (stands for “- 477580!!!&”)

And since at `*(str + 8)` we read “!”

The output is `ASN_EXTRA_DATA`.

This code has been extensively tested and used for 15 years. Formal verification guarantees absence of these kind of bugs.

# Coq mini-intro

---

We did our proofs in a formal language we choose *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 certification kernel
- extract certified programs to languages like OCaml, Haskell or Scheme.

# Inductive definitions

In Coq everything is either a *term* or a *type*. You can define basic inductive types using the `Inductive` command.

```
1   Inductive B : Set :=  
2     | true : B  
3     | false : B.
```

The boolean type has two **constructors**: `true` and `false`.

```
1   Inductive N : Set :=  
2     | 0 : N  
3     | S : N → N.
```

Natural numbers are defined by two constructors as well: `0` (zero) and `S` (successor function). Any *term* of type `N` is constructed using these two. E.g., `S 0 : N`, `S (S 0) : N`, `S (S (S 0)) : N ...`

# Recursive definitions

You can define recursive functions using the `Fixpoint` command.

Since we know that by construction any term of type  $\mathbb{N}$  is either `0` or `S n` for some  $n : \mathbb{N}$ , we can use *pattern-matching*.

```
1  Fixpoint plus (n m :  $\mathbb{N}$ ) :  $\mathbb{N}$  :=  
2  match n with  
3  | 0  $\Rightarrow$  m  
4  | S p  $\Rightarrow$  S (plus p m)  
5  end
```

Note: Coq only accepts definitions that terminate<sup>1</sup>. Here we are recursing on a direct subterm of  $n$  thus we are guaranteed to terminate and Coq is able to automatically ensure this. Sometimes you have to do a termination proof by hand.

---

<sup>1</sup>This limitation is needed to ensure consistency of the system, as well as decidability of type-checking.

Finally, you can state and prove theorems about the objects you defined.  
Let  $+$  be notation for plus.

## Theorem

$$2 + 3 = 5.$$

How would you prove this, if you were to justify each step of the proof?

```
1 Theorem plus_2_3 : (S (S 0)) + (S (S (S 0))) = (S (S (S (S (S 0))))).
2 Proof.
3   unfold plus. (* apply definition of plus *)
4   reflexivity. (* apply definition of equality *)
5   Qed.
```

In Coq you constructs a proof using so-called tactics. E.g., `simpl` is a tactic that performs basic application of definitions, `reflexivity` proves equality between two syntactically equal terms (modulo some reductions).

## More proofs

### Theorem

$\forall n (0 + n = n).$

To prove this we also use the definition of addition: since the recursion is on the first term, we are in the base case.

```
1  Fixpoint plus (n m : ℕ) : ℕ :=  
2  match n with  
3  | 0 ⇒ m (* base case *)  
4  | S p ⇒ S (plus p m)  
5  end
```

Proof in Coq:

```
1  Theorem plus_0_n : (∀ n, 0 + n = n).  
2  Proof.  
3    intros n. (* take any n *)  
4    simpl. (* apply definition of plus *)  
5    reflexivity. (* get equal terms *)  
6  Qed.
```

## More proofs

Most proofs would need to go by induction (Why the previous proof won't work below? See how plus is defined.)

```
1      Theorem plus_n_0 : ( $\forall$  n, n + 0 = n).
2      Proof.
3          (* proof by induction on n *)
4          induction n; simpl.
5          (* base case *)
6          — reflexivity.
7          (* inductive step *)
8          — rewrite IHn. (* apply induction hypothesis *)
9             reflexivity.
10     Qed.
```

In Coq you have to justify every step of the proof and the `Qed` command only succeeds on correct proofs. But you don't have to do every proof from scratch, since there are extensive libraries covering lemmas about basic mathematics as well as several decision procedures that automatize proof search.

- Using SMT and SAT solvers: <https://smtcoq.github.io/>
- First-order decision procedures (CoqHammer):  
<https://github.com/lukaszcz/coqhammer>
- Tactics for solving arithmetic goals over ordered rings (Micromega)
- And much more, cf. <https://coq.inria.fr/opam/www/>

## How does Coq work?

---



# How does Coq work?

We can formalize programs, properties and proofs in the same language, as well as efficiently and reliably check proof correctness due to the so-called **Curry-Howard isomorphism**.

Curry-Howard isomorphism is a correspondence between programs and terms on one hand and proofs and types on the other. All logical statements in Coq are typing judgments and thus checking the correctness of proofs amounts to type checking. Let's see what it means precisely on a small example.

# Curry-Howard Isomorphism (CHI)

We said that “*Curry-Howard isomorphism is a correspondence between programs and terms on one hand and proofs and types on the other*”. To make this precise, we need to specify what kind of programs we have on one side and what kind of proofs (that is, what kind of logic) we have on the other side.

Various **lambda calculi** were designed to describe programs, that is, computation: *untyped lambda-calculus*, *simply-typed lambda calculus*, *polymorphic lambda-calculus*, Gödel’s system **T**, system **F** etc<sup>2</sup>. If you never heard of them, think of functional programming languages such as **OCaml** and **Haskell**, they implement certain lambda calculi.

---

<sup>2</sup>cf. Lectures on Curry-Howard Isomorphism [?]

# The simplest CHI: simply typed lambda calculus

Consider the simplest example of typed lambda calculus: simply typed lambda calculus **ST**. Terms of **ST** are:

**Var** variables

**Abs**  $(\lambda x.M)$ , if  $x$  a variable and  $M$  is a term

**App**  $(MN)$ , if  $M$  and  $N$  are terms

Think of  $(\lambda x.M)$  as a function with argument  $x$  and body  $M$  and of  $(MN)$  as applying function  $M$  to the argument  $N$ .

# Simply typed lambda calculus

In lambda calculus one could encode basic data types such as natural numbers and lists, as well as functions on them and logical constants and operations. **ST** could be extended with a fixpoint operator to allow for recursive functions.

Examples of terms:

$\lambda x.x$  (identity function)

$\lambda x.\lambda y.x$  (true<sup>3</sup>)

$\lambda x.\lambda y.y$  (false or zero)

$\lambda n.\lambda y.\lambda x.y(nyx)$  (successor function)

---

<sup>3</sup>By convention: you could define constants in a different way as well.

# Simply typed lambda-calculus

We are interested in a **typed** lambda calculus. The types of **ST** are:

**Var** type variables

**Arr**  $\tau \rightarrow \sigma$ , if  $\tau$  and  $\sigma$  are types

Types are assigned to terms according to certain rules. Let  $\Gamma$  be a set of simply-typed terms (also called context).

$$\Gamma, x : \tau \vdash x : \tau \text{ (Ax)}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x. M) : \sigma \rightarrow \tau} \text{ (Abs)}$$

$$\frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau} \text{ (App)}$$

# Simply typed lambda-calculus

Here is a simple derivation that shows that the term  $\lambda x.x$  has type  $\sigma \rightarrow \sigma$ , with  $\sigma$  being any type of **ST**:

$$\frac{x : \sigma \vdash x : \sigma \text{ (Ax)}}{\vdash (\lambda x.x) : \sigma \rightarrow \sigma} \text{ (Abs)}$$

A more complex example:

$$\frac{\frac{\frac{y : \tau, x : \tau \rightarrow \sigma \vdash x : \tau \rightarrow \sigma \text{ (Ax)}}{y : \tau, x : \tau \rightarrow \sigma \vdash (xy) : \sigma} \text{ (Abs)}}{y : \tau \vdash \lambda x.(xy) : (\tau \rightarrow \sigma) \rightarrow \sigma} \text{ (Abs)}}{\vdash \lambda y.\lambda x.(xy) : \tau \rightarrow (\tau \rightarrow \sigma) \rightarrow \sigma} \text{ (App)}$$

We made one side of the Curry-Howard correspondence more precise: the simply typed lambda calculus, which corresponds to some rudimentary functional programming language. Now: what does it correspond to and how?

In general programs (or computations) will correspond to certain kind of logics. Namely, **intuitionistic logics**. These logics were created to formalize what is called constructive mathematics: that is, mathematics in which we are only interested in objects that can be effectively constructed.

# Implicational propositional logic

Consider the simplest intuitionistic logic **Impl**. The formulae of **Impl** are the following:

**Var** propositional variables

**Impl**  $\tau \rightarrow \sigma$ , for  $\tau, \sigma$  propositional formulae

Examples of formulae:  $p \rightarrow p$ ,  $p \rightarrow (q \rightarrow p)$ .

Let  $\Gamma$  be a set of formulae. The theorems of **Impl** are proved according to the following rules of inference:

$$\Gamma, \phi \vdash \phi \text{ (Ax)}$$

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \rightarrow \psi} (\rightarrow\text{-I})$$

$$\frac{\Gamma \vdash \phi \rightarrow \psi \quad \Gamma \vdash \phi}{\Gamma \vdash \psi} (\rightarrow\text{-E})$$



# Curry-Howard Isomorphism

You may notice a parallelism between the rules of inference of **Impl** and typing rules of simply typed lambda-calculus, this is the core of Curry-Howard Isomorphism.

$$\Gamma, \phi \vdash \phi \text{ (Ax)}$$

$$\Gamma, x : \tau \vdash x : \tau \text{ (Ax)}$$

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \rightarrow \psi} (\rightarrow\text{-I})$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x. M) : \sigma \rightarrow \tau} \text{ (Abs)}$$

$$\frac{\Gamma \vdash \phi \rightarrow \psi \quad \Gamma \vdash \phi}{\Gamma \vdash \psi} (\rightarrow\text{-E})$$

$$\frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau} \text{ (App)}$$

Consider **ST** with type variables being propositional variables; then types are formulae of **Impl**. We can show that:

## **Theorem (CHI)**

*A simple lambda term  $M$  has a type  $\tau$  in **ST** if and only if  $\tau$  is provable in **Impl**.*

# Curry-Howard Isomorphism

You've seen the simplest type theory and propositional logic and how Curry-Howard Isomorphism works.

Now we can expand **ST** to include more terms and types (you can allow sum and product types, polymorphism, recursors - which allows you to formulate more functions) or alternatively expand the logic to be more expressible and complex (add other connectives, quantifiers, inductive definitions etc).

But the principle stays the same and you have Curry-Howard Isomorphism for very complex type systems, such as Calculus of Inductive Constructions, on which Coq is based.

# Curry-Howard Isomorphism

logic	lambda-calculus
formula	type
proof	term
propositional variable	type variable
implication	function space
conjunction	product
disjunction	disjoint sum
absurdity	empty type
normalization	reduction
provability	type inhabitation

# Proofs as functional programs

Let's return to our first proofs:

```
1      Theorem plus_0_n : ( $\forall$  n, 0 + n = n).  
2      Proof.  
3          intros n.  
4          simpl.  
5          reflexivity.  
6      Qed.
```

By Curry-Howard Isomorphism, the proof of this statement corresponds to constructing a term, that is, to writing a functional program.

```
1      plus_0_n =  $\lambda$  n :  $\mathbb{N} \Rightarrow$  eq_refl  
2      :  $\forall$  n :  $\mathbb{N}$ , 0 + n = n
```

with `eq_refl` being a term of type `x = x`.

# Proofs as functional programs

The following proof has a more complex term:

```
1  Theorem plus_n_0 : ( $\forall$  n, n + 0 = n).
2  Proof.
3    induction n; simpl.
4    (* base case *)
5    - reflexivity.
6    (* inductive step *)
7    - rewrite IHn. (* here we apply induction hypothesis *)
8    reflexivity.
9  Qed.

1  plus_n_0 =  $\lambda$  n :  $\mathbb{N} \Rightarrow$ 
2     $\mathbb{N}_{\text{ind}}$  ( $\lambda$  n0 :  $\mathbb{N} \Rightarrow$  n0 + 0 = n0) eq_refl
3    ( $\lambda$  (n0 :  $\mathbb{N}$ ) (IHn : n0 + 0 = n0)  $\Rightarrow$ 
4      eq_ind_r ( $\lambda$  n1 :  $\mathbb{N} \Rightarrow$  S n1 = S n0) eq_refl IHn) n
5  :  $\forall$  n :  $\mathbb{N}$ , n + 0 = n
```

where `nat_ind` is a term with type that corresponds to induction on natural numbers.

Since the logic we are dealing with is more complex than **Impl**, the functional programs corresponding to these types are also way more complex. However, checking whether a given term  $t$  has a given type  $\sigma$  is a decidable problem even for Calculus of Inductive Constructions. By Curry-Howard Isomorphism, this also yields a procedure for checking the correctness of proofs written in Coq's logic. This algorithm constitutes Coq's trusted **kernel**.

Now we will see how Coq is used for verifying programs in a bit.



## Verifying C code: a detailed example

---

## Factorial example

Mathematical specification of factorial is a recursive equation, for  $(0 \leq n)$ :

$$\begin{aligned} \text{fact}(0) &= 1 \\ \text{fact}(n+1) &= \text{fact}(n) * (n+1) \end{aligned}$$

We can write it in Coq as a fixpoint definition:

```
1  Fixpoint fact (n : ℕ) : ℕ :=  
2    match n with  
3      | 0 ⇒ 1  
4      | S n' ⇒ n * fact n'  
5    end.
```

Note that this definition is also a functional program<sup>4</sup>.

<sup>4</sup>This corresponds to the idea of verifying a program wrt reference implementation. For more complex program we will want to write a specification in a more declarative fashion.

## Factorial example: verifying a functional program

We can write a more efficient (tail-recursive) functional program to compute factorial

```
1  Fixpoint fact_acc (n : ℕ) (acc : ℕ) :=
2    match n with
3    | 0 ⇒ acc
4    | S k ⇒ fact_acc k (n * acc)
5  end.
6
7  Definition fact' (n : ℕ) :=
8    fact_acc n 1.
```

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

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

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 you need to embed the language into Coq, meaning model its syntax (as abstract syntax trees) and semantics (modelling execution of programs) in Coq. Luckily, this has been already done in the project called CompCert, 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

So using CompCert our approach is as follows:

- Parse C code into an abstract syntax tree using C light<sup>5</sup> generator of CompCert
- Write a functional specification using CompCert's model of C light
- Reason about the C light program using operational semantics defined in CompCert

---

<sup>5</sup>C light is a subset of C

## Factorial example: verifying a C program

Factorial C implementation that we want to verify

```
1 unsigned int factorial (unsigned int input) {  
2     unsigned int output = 1;  
3     while (input){  
4         output = output*input ;  
5         input = input - 1 ;  
6     }  
7     return output ;  
8 }
```

The specification stays the same.

# Syntax of C programs in Coq

C function embedded in Coq as an abstract syntax tree:

```
1 (Ssequence
2   ((* int output = 1 *))
3   (Sset _output (Econst_int (Int.repr 1) tuint))
4   (Ssequence
5     (Swhile
6       (Etempvar _input tuint) ((* while (input) *))
7       (Ssequence
8         (Sset _output
9           (Ebinop Omul (Etempvar _output tuint)
10            ((* output = output*input *))
11            (Etempvar _input tuint) tuint))
12         (Sset _input
13           (Ebinop Osub (Etempvar _input tuint)
14            ((* input = input - 1 *))
15            (Econst_int (Int.repr 1) tuint) tuint))))
16     ((* return output *))
17     (Sreturn (Some (Etempvar _output tuint))))).
```



## C light Expressions: Examples

```
1      (* 0 *)
2      (Econst_int Int.zero tint)
3
4      (* 0 + 1 *)
5      (Ebinop Oadd (Econst_int Int.zero tint)
6      (Econst_int (Int.repr 1) tint) (tint))
7
8      (* int *p *)
9      (Etempvar _p (tptr tint))
10
11     (* (*p) *)
12     (Ederef (Etempvar _p (tptr tint)) tint)
```

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 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.

We assign primitive values to constants and then compositionally compute values of expressions and outcomes of statements. The evaluation is done in the context of local environment and memory state.

- Each syntactic element is related to the intended result of executing this element (new local environment, memory, outcome or value).
- 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 recorded.

## Examples

- Expression `(Econst_int Int.zero tint)` is evaluated to value 0 in any local environment and memory.
- 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.
- Statement `(Sreturn (Some (Etempvar _s tint)))` evaluates to a return outcome and leaves  $le$  and memory unchanged.

As you can see, in order to verify a program written in C, one has to have a good model of variable environments, integer and pointer arithmetics 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 (current and maximal): `Freeable`, `Writable`, `Readable`, `Nonempty`, `Empty`, ranging from allowing all operations to allowing no operation respectively.

# C 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.

Hence the correctness theorem for factorial would be of the following form:

*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  $m' = m$ .*

Hence we proved that factorial works correctly on all inputs<sup>6</sup>.

---

<sup>6</sup>That do not lead to overflow.

Go to factorial tutorial:

`~/asn1verification/doc/tutorial/MiscExamples/factorial`



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:

We wrote a formal specification of the function (based on the comment and analysis of the function), produced C light AST of the function using C light 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

---

- <http://www.jscert.org/> JSCert: Certified JavaScript
- RustBelt, Vellvm, CakeML
- Krakatoa: A Java code certification tool that uses both Coq and Why to verify the soundness of implementations with regards to the specifications.

