Formal Verification of Computer Programs

A Primer

Vadim Zaliva [1] Nika Pona [2]

[1] Carnegie Mellon University

[2] Digamma.ai

Outline

1. What are formal methods?
2. Proof assistants (Coq) mini-intro
3. Reasoning about programs (semantics, specification, etc)
4. Motivating example (strtoimax_lim)
5. Detailed example (strlen of similar)
6. Other languages? (Javascript, LLVM, etc.)
7. Other approaches (exraction)
8. Conclusions

Table of contents

1. What are formal methods?

2. Motivating Example

3. Detailed example

4. Other languages

What are formal methods?

Formal verification

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 and how do we do it?

1. We start by writing a *specification* in a formal language which strictly defines how our program should behave.
2. Then we define 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.

What Coq does?

In Coq you can:

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

Logic in Coq

`Prop` stands for Proposition: this is the type used for logical definitions.

```
Inductive False : Prop := .

Inductive True : Prop :=
| I : True.

Theorem impl_to_and : (∀ A B : Prop, A → B → A ∧ B).
```

The theorem states that `impl_to_and` is a proof (to be constructed) of $\forall A\ B : \texttt{Prop}, A \to B \to A \wedge B$ or equivalently that $\forall A\ B : \texttt{Prop}, A \to B \to A \wedge B$ is the type of `impl_to_and`.

Basic Types

You can define basic inductive types using `Inductive` command:

```
Inductive 𝔹 : Set :=
  | true : 𝔹
  | false : 𝔹.
```

```
Inductive ℕ : Set :=
  | O : ℕ
  | S : ℕ → ℕ.
```

And state and prove theorems about them:

```
Theorem zero_lt_Sn : (∀ n : ℕ, 0 < S n).
```

```
Theorem 𝔹_dec : (∀ b : 𝔹, b = true ∨ b = false).
```

Recursive functions

You can define recursive functions and use pattern-matching on inductive types.

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

Note that you can only write terminating functions in Coq. (Hence, you often get the termination proof "for free").

How does Coq work?

Programs, properties and proofs can be formalized in the same language due to the so-called Curry-Howard isomorphism (correspondence between programs and terms, proofs and types respectively). All logical judgments in Coq are typing judgments and thus checking the correctness of proofs amounts to type checking. We will see how Coq is used for veryfing programs in a bit. But first a motivating example.

Motivating Example

asn_strtoimax_lim

In this talk we are concentrating 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 real-life function. 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).*

The code: `https://github.com/vlm/asn1c/blob/9f470d60faf6b994de6b9d6c0dbfb9279d9bb48d/skeletons/INTEGER.c#L1033`

asn_strtoimax_lim

While doing the correctness proof we found a bug, can you see it?

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

asn_strtoimax_lim

Fixed version: `https://github.com/vlm/asn1c/blob/b361194599b46d97a398195e6a18f1d581f7fab9/skeletons/INTEGER.c#L1033`

Is this fix OK?

asn_strtoimax_lim

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

Detailed example

Factorial example

Mathematical specification of factorial is 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 (recursive) definition:

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

Note that this definition is also a functional program.

Factorial example: verifying a functional program

We can write a more efficient 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.
```

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[1] 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

[1] C light is a subset of C

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

C light AST corresponding to the factorial function:

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

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 **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

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

- 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 recorded.

Go to factorial tutorial:

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

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

Other languages

`http://www.jscert.org/` JSCert: Certified JavaScript

References I