Introduction to the formal verification

Prepared by Pruvendo

The well established traditional QA process ensures the tremendous quality of software products. However, for some mission-critical applications such as finance, aerospace, transport, medicine etc. the quality requirements are even more strict than those QA can afford.

Fortunately, the formal methods can help here. While developed in the 70th they were never widely adopted due to their complexity and very high entrance threshold. The Pruvendo mission is to make these methods applicable for the regular IT-professionals.

Some basic explanation about formal methods can be found below:

Though every program is definitely designed and in principle can work on a finite set of input data and internal state, the variety of resulting space is great enough to forget about testing (checking) the correctness of a program behavior on each possible value.

Here the math comes. The mathematical logic and correspondent type systems allow to “quantify” (apply a quantifier before) the term, which gives the possibility to “check” the program behavior immediately on every possible input and internal state. We usually write “forall” quantifier as ∀ and mean that if the statement ∀ x, P (x) holds, property (or predicate) P actually holds for every x from some eligible (probably infinite) subset. The possibility to reason and argue about “*forall*” quantified properties on a program lies deep in the theory of programming languages and its interpretation on admissible machines (real or virtual).

Another mathematical abstraction which allows the formal deductive methods of verification to evolve is Curry-Howard correspondence. Briefly it means that in some full enough type theories (for example Calculus of constructions (CoC), which the Coq framework is based on) there is deep mathematical correspondence between programs and proofs, and types and propositions. That is to formulate a proposition one needs to construct the *type*, and a program which has this type as outcome will in the same way *prove* the original proposition. So in deductive verification based on CoC, proofs look like programs, and statements are actually types (eventually including all the logic symbols like quantifiers).

Suggest we write a list sorting algorithm that way (in ML like syntax):

```
Fixpoint insert (n:Z) (ms : zlist) {struct ms} : zlist :=
match ms with
| nil => [n]
| m :: ms' => if (n <=? m) then (n :: ms)
                           else (m :: (insert n ms'))
end.

Fixpoint sort (ms : zlist) : zlist :=
match ms with
| [] => []
| m :: ms' => insert m (sort ms')
end.
```

Then the formulation of the sorting specification can be presented by the theorems:

```
(1) Theorem sort_perm: forall (l: zlist),  Permutation l (sort l).
```

where `Permutation` is an inductive predicate stated that one list is permutation of another.

```
(2) Theorem sort_sorted: forall (l: zlist), sorted (sort l).
```

where sort predicate is formulated as follows:

```
Definition headDefault {X} (a: X) (l: list X) : X :=
match l with
| [] => a
| x :: _ => x
end.

Inductive sorted : zlist -> Prop :=
| sortnil: sorted []
| sortcons: forall (a:Z) m, sorted m -> (a <= (headDefault a m)) ->
sorted (a :: m).
```

stated that the order inside the list is held inductively.

Further we can prove those theorems strictly using Coq proof assistant and Coq checks our proofs so no more need for any human review or analysis of them. The main outcome of this procedure is that we have a trinity: specification (theorems to be held), code itself (written in ML embedded to Coq in example) and the proof which was checked by Coq in its turn. All that means that the sorting algorithm is correct against the given specification.

If everything is so good, why not formally verify every program. Besides a lot of technical still unsolved tasks, there are two bold arguments: it is always very difficult to perform and to realize that it is actually done. Sometimes it is also completely impossible.

On the first point, Poincare and Hadamard established that all more or less formally formulated tasks can be divided into direct and indirect (inverse) simply put - correct and incorrect. Direct task is such where there exists an algorithm to solve it step by step starting from initial data, having the correct answer or proven failure to solve (e.g. number addition, or quadratic equation) at the end. The indirect task is usually an inverse task of finding initial data which corresponds to observed behavior. Often, humans find a way to transform the inverse tasks to direct (quadratic equation still works as example), but more oftenly they fail. Formal verification is the task to find the properties of the given program and prove them in a formal way. No direct way to solve this has been found.

But no less critical as we said earlier is that sometimes the whole task is completely impossible. As programs are not scientific objects, their properties have to be realizable by developers and users. Mathematical proof is not of common sense, and even an excellent programming engineer might not realize all the properties the written program satisfies. Moreover the set of program properties is almost always infinite and there is no known general way of reducing that infinite set to a “finite minimal normal form” with an exception of trivial programs like a list sort. Another example is arithmetics. Widely known that number theory is not fully developed and may never be. Kurt Gödel broke people’s hopes on fully closed mathematics by stating his theorem of incompleteness. So if a program behavior is based on some number properties which are still unknown (unproven) or very difficult (Fermat theorem), to prove program behavior, correspondent theorems have to be involved. However it seems very simple to write a program which depends on the Fermat theorem’s truth. Easy to realize that understanding the properties of such a program will be close to incomprehensible for an ordinary observer.

The latest point is commonly referred to as a main incapability of formal methods. It lies in the so-called halting problem originally formulated for abstract Turing machines. Informally speaking it means that there is no universal program which can reason about any other input program if the latter uses wide enough grammar. Simpler, one cannot write a program which can prove that any given other program halts or runs infinitely.

As a result we see deep incompleteness of any formal method we can use to reason about programs. However the internal contradiction is the following: we have the powerful mechanism to speak about program properties, much more powerful than any testing suite can even imagine but it is limited by the very nature of mathematics. There are many engineering approaches to mix the methods, verifying core parts and testing a less sensitive environment, modeling core program behavior or underlying protocols in simplified sandboxes and many others.

However we think that the root of the problem is that humans search for a more expressive way to write programs where the more expressive the meaning is - the less it is understandable by other programs.

Having the given problems the keys to look for a solution are quite simple:

- every programming engineer knows or believes why her program halts or works properly and she can answer the direct question why it is so - otherwise there is a good motive to rewrite or refactor;
- Nobody really wants to write a program with uncertain properties for production use.

Instead of analyzing a program's correctness by external tools, we might give a programmer tools where she can explain why her program should work like this. Moreover in most of the cases it is quite self explanatory.

The key approach applied by Pruvendo for the good of formal methods and making contracts secure by construction:
We are not going to limit engineers to express their ideas in the most suitable programming language;

- "Good lenses": the tool to look into the state on the stage of programming, not running or testing. This requires adequate blockchain model and REPL driven development where every step is depicted in the nearby and gives the correct feedback;
- "Good configurator": the tool to express the willingness of both engineer and designer. This tool is an ultimate part - the better we can be self-explanatory, the better all the next parts will work.
- "Good programming language". We actually have it (choose any). We need to add some "steroids" for further steps. Ideally to have a language where we can express a trinity: a specification, a program itself and a proof (which actually Coq does but in a too purist way).

- This language can be embedded to the prover bidirectionally. If one likes to verify the already written program she can translate it, embed it in the prover and verify in a friendly environment. Conversely, if one likes to write a more strict program, having the immediate possibility to prove its behavior - do it inside the prover having minimum

differences to “normal” programming process and after satisfaction with proofs - translate it to compilable language, build and deploy.

- The “steroids” mentioned are the possibility to do it instantly, not dividing by phases. The ideal configuration is “write a bit of spec -> write a bit of code -> write a bit of proof -> feedback -> loop”. Humanity can do the last three steps more or less acceptable, but without specification one still is unaware of what she really codes.
- “Good prover”. There are a lot of them, no actual need to have a new one. The problem is to embed the specification and program into.
- “Good automation”. We actually need a solver which helps us to think on the level of abstraction we are used to, solving all easier tasks in more or less invisible mode (in background). That is purely an engineering task: the more code of automation - the more actual automation. Good practice here is to cover most common cases: find the most common pattern of propositions, automate reasoning about it, see what remains. Good heuristics and machine learning will not harm. This part is not so time sensitive.

Distributed programming (such as smart contracts) greatly increases the capability to do formal verification of programs simply due to the fact they become smaller and less complicated. Also it significantly increases reuse of formally verified code within the ecosystem.