How to Start on Formal Methods and Share It

January 27, 2022

You might have heard of formal methods and there is a small chance that you might even be using them. We know that we cannot really rigorously test software and formal methods have the answer. However, it is hard to find anyone in our circle that was actually using them, except for some exotic projects on the web, and our guess was that it is hard to learn and use. Long story short, we decided to learn a thing or two about formal method and try to use it in some way in general software development.

Formal verification

> Testing can show the presence of bugs, not their absence.
>
> Edsger W. Dijkstra

Verification in software development usually means testing, i.e. the running of test subject with some input and checking the output against some expected output. Formal verification is checking the code against some formal proofs. When we explained this to some people, the responses were usually, "Yeah, of course" or "That makes sense". Showing this on a piece of paper or on a white-board is a different story. We found that simple math can illustrate the problem: let's say we have a specification $(x + y)^2$ and a programmer implemented it as $x^2 + 2xy + y^2$ and now we have to verify it.

Two things that we want to point out with this illustration are 1) the "formal proof" is not that hard to understand as it is basic math and 2) as we will show later, computer code can be verified in the same way using formal proof. But why is it called formal proof? The "formality" can be seen as the level of details,

Table 1: Unit test cases

x	y	result
0	0	0
0	1	1
1	0	1
1	1	4

Table 2: Formal proof

Proof step	Goal
	$(x+y)^2 = x^2 + 2xy + y^2$
expansion	$(x+y)(x+y) = x^2 + 2xy + y^2$
distributive	$x(x+y) + y(x+y) = x^2 + 2xy + y^2$
simplify	$x^2 + xy + yx + y^2 = x^2 + 2xy + y^2$
reflexivity	$x^2 + 2xy + y^2 = x^2 + 2xy + y^2$

analogous to formal proof being a programming language and informal as a natural language. In informal proof, one can say "expand the power of 2 then use distributive law". Formal proof is writing down the proof steps. Informal proof requires some knowledge and intuition to comprehend, where formal proof is so detailed it can be processed by a computer.

Prerequisite

There are many tools that can be used for formal verification, together they are called Proof Assistants. We choose Coq because we found that one of the best sources to learn about formal verification is the Software Foundations[1] and it uses Coq. Coq is built on OCaml and a convenient way to install Coq and its dependencies is by using opam, the OCaml package manager. It also allows you to install packages built on Coq later in the journey. We will also use 'make', which hopefully you already have installed

To install opam, please go to https://opam.ocaml.org/doc/Install.html and follow the instruction that is suitable for your OS. After opam is installed, installing Coq is just a matter of doing '$ opam install coq'.

Writing specifications

As our intention is to make this formal verification exercise really applicable to whatever project you might have, we want to write the example specifications inside some existing source code that you might have been working on. If you don't have one, just create an empty one in any language, as long as it supports some kind of doc block notation. We will use PHP here for our example.

Let's say we have a function that is used to decide whether a given transaction is available or not based on a few simple rules and we want to formally verify the implementation.

```
/**
 * returns true when a transaction is:
 * unlocked and committed and a withdrawal or
 * unlocked and it was a reversal or
 * unlocked and committed and undisputed
```

```
 */
function available(bool unlocked, bool disputed,
        bool committed, bool reversal, bool withdrawal): bool
{
        return unlocked && (committed && (!disputed
                || withdrawal) || reversal);
}
```

From the piece of code above we can see the specifications of the function in the comments and the function definition as the implementation. Obviously the example here is crafted to make it straight forward to translate it into Coq. Now, let's translate the rules in the comment block, we start by replacing "and" with "/\" (forward and back slashes) and removing "or".

- unlocked /\ committed /\ a withdrawal
- unlocked /\ a reversal
- unlocked /\ committed /\ undisputed

These rules must apply to all types of transactions, withdrawal and reversal were special types of transactions by the look of it, to keep it simple, we can treat them as statuses like "unlocked" and "committed", so we can drop "a" and add "forall" to these rules because we want these rules to work "for all" cases:

- forall unlocked /\ committed /\ withdrawal
- forall unlocked /\ reversal
- forall unlocked /\ committed /\ undisputed

Specification is one half of proposition, called the antecedent and the implementation code is the second half of proposition, called the consequent. Propositions have the form "antecedent -> consequent". To convert the implementation we just take the one line of code and replace "!", "&&" and "||" with "~", "/\" (aka conjunction) and "\/" (aka disjunction) respectively like so:

unlocked /\ (committed /\ (~disputed \/ withdrawal) \/ reversal)

Now we can write the complete specifications:

- forall unlocked /\ committed /\ withdrawal -> unlocked /\ (committed /\ (~ disputed \/ withdrawal) \/ reversal).
- forall unlocked /\ reversal -> unlocked /\ (committed /\ (~ disputed \/ withdrawal) \/ reversal).
- forall unlocked /\ committed /\ ~disputed -> unlocked /\ (committed /\ (~ disputed \/ withdrawal) \/ reversal).

They look almost ready, we just need to tell Coq what all these variables are, we tell Coq that these are proposition with "(unlocked committed disputed reversal withdrawal: Prop)". Coq also need to be able to tell what each line is, we just call them Property (we could also call them Lemma or Theorem, they are syntactic sugar) and we give each one a name. The finished lines look like this:

- Property withdrawal: forall (unlocked committed disputed reversal withdrawal: Prop), unlocked /\ committed /\ withdrawal -> unlocked /\ (committed /\ (~ disputed \/ withdrawal) \/ reversal).
- Property reversal: forall (unlocked committed disputed reversal withdrawal: Prop), unlocked /\ reversal -> unlocked /\ (committed /\ (~ disputed \/ withdrawal) \/ reversal).
- Property undisputed: forall (unlocked committed disputed reversal withdrawal: Prop), unlocked /\ committed /\ ~disputed -> unlocked /\ (committed /\ (~ disputed \/ withdrawal) \/ reversal).

Before we continue, we want to point out that unlike logical statements, there is no right or wrong in propositions. Proposition can either be proven or cannot be proven. What that means is if a proposition cannot be proven, it may still give true results, but not always, and it may give false results which in real life can be referred to as "edge cases", "dead lock", "race condition", etc.

Develop proof for propositions

Coq is a proof assistant, it does not create the proof for us, so we have to create the proof. A good thing about formal proof is that we can see each step of the proof and what the meaning of the proof steps are although they can be long compared to informal proof, it does not require intuition, especially when the intuition requires long and hard study in logic. Lets try creating proof for "Property withdrawal" inside Coq, start Coq by running 'coqtop', copy the "Property withdrawal" line and paste it at the "Coq < " prompt.

```
Coq < Property withdrawal:forall (unlocked committed disputed \
reversal withdrawal: Prop), unlocked /\ committed /\ \
withdrawal -> unlocked /\ (committed /\ (~ disputed \/ \
withdrawal) \/ reversal).
1 subgoal
     ============================
  forall unlocked committed disputed reversal withdrawal : Prop,
  unlocked /\ committed /\ withdrawal ->
  unlocked /\ (committed /\ (~ disputed \/ withdrawal) \/ reversal)
```

Coq has accepted the property as the context, indicated by the prompt has changed to "withdrawal". Please note that the line continuation character "\" were only added for formatting here. When using coqtop interactively, the property should be entered as one line with "\" removed. We can now start proving by giving it the command "Proof." ('.' is the statement delimiter in Coq).

```
withdrawal < Proof.
1 subgoal
```

```
  ============================
  forall unlocked committed disputed reversal withdrawal : Prop,
  unlocked /\ committed /\ withdrawal ->
  unlocked /\ (committed /\ (~ disputed \/ withdrawal) \/ reversal)
```

This is where the fun begins. The goal that we are trying to prove is basically in the form of "P -> Q". The Q part in the current form is a bit long to read, but if we ignore the parts within the outer parentheses for now, it is a conjunction ("/\") of A & B or "A /\ B". What that means is to prove P -> Q, we have to prove both P -> A and P -> B. We can tell Coq that we will prove both of them separately using the tactic "split".

```
withdrawal < split.
2 subgoals

  unlocked, committed, disputed, reversal, withdrawal : Prop
  H : unlocked /\ committed /\ withdrawal
  ============================
  unlocked

subgoal 2 is:
 committed /\ (~ disputed \/ withdrawal) \/ reversal
```

Now we have two lines above "==========", the first line lists all the variables and the second line shows the hypothesis "H". We also now have two subgoals, the first and current subgoal is "unlocked". Since we have "unlocked /\ committed /\ withdrawal" as hypothesis, we apply the hypothesis to the goal using "apply H".

```
withdrawal < apply H.
1 subgoal

  unlocked, committed, disputed, reversal, withdrawal : Prop
  H : unlocked /\ committed /\ withdrawal
  ============================
  committed /\ (~ disputed \/ withdrawal) \/ reversal
```

As the first goal has been satisfied, Coq is now showing the next goal. The goal has a disjunction ("\/") which means that the two parts on either side of the disjunction are two separate statements "Left \/ Right". Because they are two separate statements, we do not need to "split" the goal. Just like a logical "or" in programming, it does not really matter which statement is true, as long as one of them is true. Now, which side should we prove? Looking at the subgoal, Right is "reversal", it is something that we do not have in the current hypothesis, if we choose Right, we will not progress our proving. So, let's choose to continue proving on Left, the Coq command is "left".

```
withdrawal < left.
1 subgoal

  unlocked, committed, disputed, reversal, withdrawal : Prop
  H : unlocked /\ committed /\ withdrawal
  ============================
  committed /\ (~ disputed \/ withdrawal)
```

Here we have the "A /\ B" again, so let's split it.

```
withdrawal < split.
2 subgoals

  unlocked, committed, disputed, reversal, withdrawal : Prop
  H : unlocked /\ committed /\ withdrawal
  ============================
  committed

subgoal 2 is:
 ~ disputed \/ withdrawal
```

We have subgoal 1 (committed) in the hypothesis, so we can apply the hypothesis.

```
withdrawal < apply H.
1 subgoal

  unlocked, committed, disputed, reversal, withdrawal : Prop
  H : unlocked /\ committed /\ withdrawal
  ============================
  ~ disputed \/ withdrawal
```

We have to choose between left (~disputed) or right (withdrawal). We can see withdrawal in the hypothesis so let's go with Right.

```
withdrawal < right.
1 subgoal

  unlocked, committed, disputed, reversal, withdrawal : Prop
  H : unlocked /\ committed /\ withdrawal
  ============================
  withdrawal
```

And now we can apply the hypothesis.

```
withdrawal < apply H.
No more subgoals.
```

That is it! We close the proof with "Qed".

```
withdrawal < Qed.
Proof.
split.
 apply H.

 left.
 split.
  apply H.

  right.
  apply H.

Qed.
withdrawal is defined
```

Only two more rules to prove! Hint: the second rule is even easier to prove. When you are done, the command to quit Coq is "Quit."

Let's recap what we have learnt so far, we have shown how to write proof using three tactics: split, left/right and apply. There are many more tactics available in Coq, the highlight here is that we can start with a handful of tactics and write a working proof. Looking back at the original comment block and the function code, we have extracted the logic from the rules and the implementation code and make the connection between them in the form of proof. Having the proof available relieves programmers or code reviewers from mentally making that connection and gives assurance of the correctness.

Putting all together

We now know how to write propositions and formal proofs. The next goal that we want to achieve is to apply it to our project. One way to do it is to create a sub-project and write as much as we can in Coq then integrate it back to the main project. This is a common approach, there are many projects done this way. The downside is they become separated from the main projects, whatever that main projects are, and isolate the techniques and knowledge used in that project to the few people who are working on it. This is fine for domain specific knowledge, but formal methods are applicable across domains. We want people to practice formal method like they practice test driven or behaviour driven development. So, we add a subgoal: we want to apply formal methods to our main project, in other words, we will share it so other people working on the same project can see it too.

Some code already has comments to explain what it was doing and they should also come with test files. In code review, we read them and check if they make sense. We think we could do better, if we put the formal specification and the formal proof together with the code, then all we need to check is that the code was translated correctly into the specification and let machine do the rest. Here is the original code with specification and proof in the comment:

```
/**
* returns true when:
* unlocked and committed and a withdrawal or
* unlocked and it was a reversal or
* unlocked and committed and undisputed
*)
Property withdrawal:
forall (unlocked committed disputed reversal \
  withdrawal: Prop),
unlocked /\ committed /\ withdrawal -> unlocked \
  /\ (committed /\ (~ disputed \/ withdrawal) \/ \
  reversal).
Proof. split. apply H. left. split. apply H. right. \
  apply H. Qed.

Property reversal:
forall (unlocked committed disputed reversal \
  withdrawal: Prop),
unlocked /\ reversal -> unlocked /\ (committed /\ \
  (~ disputed \/ withdrawal) \/ reversal).
Proof. Admitted.

Property undisputed:
forall (unlocked committed disputed reversal \
  withdrawal: Prop),
unlocked /\ committed /\ ~ disputed -> unlocked /\ \
  (committed /\ (~ disputed \/ withdrawal) \/ reversal).
Proof. Admitted.

(*
*/
function available(bool unlocked, bool disputed,
        bool committed, bool reversal, bool withdrawal): bool
{
        return unlocked && (committed && (!disputed ||
                withdrawal) || reversal);
}
```

If we know how to separate the Coq from the rest of the code, then it should be straight forward to check the proofs. Here is what our Makefile looks like:

```
proof: test.vo

%.v: %.php
@cat $^| sed -ne '/\/\*/,/\*\//p' | sed -n '/\*)/,/(\*/p' \
| grep -v '(\*' | grep -v '\*)' > $@
```

```
%.vo: %.v
coqc $^

clean:
$(RM) *.v *.vo *.glob
```

We named the php containing the code "test.php" which 'make' will look for to build "test.vo" file, which is the output of Coq compiler. When we run 'make', it will extract Coq code from 'test.php' into 'test.v' and run 'coqc' on the file. The key to make that works is to use Coq's block comment delimiter "(* *)". In the PHP code above we have Coq code between "*)" and "(*", this is intentional because the Coq code lives inside a PHP comment block. When the file is read as Coq code, the PHP code would be the comment block, hence the inverse commenting "*) ... (*".

As usual, if everything was Ok, make will build "test.vo" file and if not then error will be thrown. In fact, you should try removing one of the proof step and see what happens, 'make' should throw an error. At this point, we believe it is straight forward to make this as part of a project or help integrating it into a build pipeline.

Closing

That is pretty much it! From here, it should just be like learning a new tool or programming language. There will be challenges and shortcuts along the way. Some of them are:

- Coqtop does not have the best user interface for development. Coq has an IDE, which is called CoqIDE, and there are also text editor plug-ins, like Proof General for Emacs.
- The example shown was for proving proposition A -> B. We actually have to prove that A <-> B and the way to do it is to "split" it into A -> B and B -> A. With multiple rules as in the example, you can write them all as one proposition like A_1 \/ A_2\/ A_3 <-> B.
- Sometimes Coq can do the job for you because someone has written smart tactics in the library. E.g. try using "tauto" in the proof, as in "Proof. tauto. Qed".

As people get more comfortable with Coq and if you write functional programs, you might even go the whole length and write your types and functions in Coq. This will give even more return on your investment because you will be able to use types and functions as part of formal proof. Coq can then "extract" the code into Haskell or OCaml.

There are other alternatives to Coq, one of them Isabelle/HOL which is used in seL4 microkernel project[2]. Operating systems are common targets for formal methods because it sits at the bottom of the stack so it should give most

bang for the buck. CertiKOS is another project that is formally verified, this time with Coq[3].

The example used here was based on real experience. When we review code, it is not hard to check some piece of code which has two or three boolean variables, by mentally doing a truth table. When it gets to four or more variables it will get out of hand. Having proof assistant to check some logical expressions is like having a calculator. Sure we can do math with pen and paper but when machine can do it, usually we can trust the result. Most programmers already understand that features in a softwares come from many small functions. We hope that by exposing formal proof of some propositions in a common project will lead to better understanding of, let's say how an application can be proven or certified to be secure. Or at least it will start a discussion.

References

[1] https://softwarefoundations.cis.upenn.edu/lf-current/toc.html

[2] http://sel4.systems

[3] https://www.usenix.org/system/files/conference/osdi16/osdi16-gu.pdf