

Formalization of the Fu's logic in Coq

Guillaume Claret

August 11, 2010

Abstract

The goal is to give a Coq formalization of **Fu's logic** to verify the soundness of the logic and use it to prove several algorithms.

I will be working with Yanni Kouskoulas. The code is available on a public Mercurial repository : http://bitbucket.org/guillaumeclaret/fu_logic/.

1 Code base and first improvements

This project will follow a previous formalization attempt I stopped three weeks before, so there is already a code base. But I get stuck on many problems we will try to solve here.

1.1 Other separation logic implementations

- <http://staff.aist.go.jp/reynald.affeldt/seplog/> : project to verify pieces of MIPS assembly using separation logic ; very complete, with concrete examples
- **Tactics for Separation Logic** : a paper to present a general way to implement good separation logic tactics

1.2 Syntax trees of assertions

There is the choice of describing the assertions first as a syntax, with an abstract tree and an evaluation function to give the semantics, or directly as Coq assertion on states or traces.

The first possibility allows more automation, since the syntax allows introspection to do substitutions, which can be critical in rules for assignments. For example, if we want to compute the *Weakest Precondition* of P for a store variable assignment :

$$\{WP(P) = P[v := E]\} v = E; \{P\}$$

we need to do a substitution on the variable v in P .

But if we represent predicates as Coq functions of type `state -> Prop`, it is impossible to compute such an explicit form. We can only do a function composition for the general case :

$$WP(P) = P \circ \{(s, h) \mapsto (s[v := \llbracket E \rrbracket_{(s,h)}], h)\}$$

and next we need to apply some tactics, based on rewriting rules, to get an explicit result. What is both more complex and general than the previous form.

However, it is often much easier to manipulate, because syntax and semantics are the same, easier to extend with new predicates. It also allows the usage of Coq operators, like `forall` or `exists`, without having to define the notion of assertion variable, free variable, ... which can be a mess to formalize.

1.3 Logical variables

We often want to use logical variables in assertions to record the value of a program expression at a special point during the execution. For example, with X a logical variable to record the value of E :

$$\overline{\{X = E\} v = E; \{v = X\}}$$

The question is how to define precisely X .

We could first introduce a code annotation mechanism, with a **let** pseudo-instruction. For that, we need to extend the program state with a store for logical variables, ie the state become a triple $(store, heap, logical_variables)$. The semantics of a **let** is then to assign a new logical variable to the value of an expression at a program point. It allows us to deduce the rule :

$$\frac{\{P \wedge X = E\} C \{Q\}}{\{P\} \mathbf{let} X = E \mathbf{in} C \{Q\}}$$

so we can always enrich the hypothesis with the value of an expression.

But this method need to extend artificially the program state. Actually, we do not need it, if we use an external universal quantification instead ; for example, for the assign rule :

$$\overline{\forall X \in int, \{X = E\} v = E; \{v = X\}}$$

This quantification is not part of the assertion language. To remove it, we have this simple rule :

$$\frac{\forall X \in int, \{P \wedge X = E\} C \{Q\}}{\{P\} C \{Q\}}$$

where P and Q do not depend on X .

1.4 Equivalence of assertions

We will have to manipulate assertions all the time doing proofs, doing simplifications or term reorganisation on them. So we need an easy way to claim and to show that two assertions are equivalent :

$$P \sim Q \iff \forall s \in State, P(s) \iff Q(s)$$

This is the natural definition, and we can next show basic results such as :

$$emp * P \sim P$$

$$P * Q \sim Q * P$$

But, since it is just an equivalence class, we would need to show that all our operations using assertions are congruences according to \sim , to be able to do replacements. Fortunately, this can be avoided adding two logical axiom to Coq :

- proposition's extensionality : $P \iff Q \implies P = Q$
- functional extensionality : $\forall x, f(x) = g(x) \implies f = g$

Using this, we can then show :

$$P \sim Q \implies P = Q$$

and we get that all the operations are congruences for free.