Very cool title goes here

Cibele Freire and Emma Tosch

May 9, 2014

Instructor: Professor Neil Immerman

Abstract

Logic is foundational to computer science. With discrete mathematics, is often the entry point for students to learn to write proofs. Many proof techniques are first presented in an algorithmic way; students are supplied with a set of conditions for which the technique will apply, and a procedure for constructing evidence for their goal. This approach suffices for simple proofs, but fails for proofs having more complex structure (e.g. requiring lemmas) or for proofs with more subtle inductive hypotheses.

Interactive proof assistants, such as Coq, can be used to aid students in formally stating their assumptions, identifying candidate structures for induction, and correctly stating the inductive hypotheses. The partial automation of these assistants elides over trivial cases, allowing students to focus on more difficult aspects of writing proofs. The type-checking baked in to such systems prevents students from generating invalid inductive hypotheses, without going so far as preventing them from generating inductive hypotheses that are not useful.

We propose several ways to use Coq in an introductory Logic course that we believe will help students better grasp the mechanics of writing proofs.

1 Introduction

The goals of the project are:

- Introduce the proof assistant
- Compare with other proof assistants
- Show how to encode propositional logic (first-order logic too?)
- Present it as a tool that can help learning logic concepts and direct proofs (I mean direct proofs in opposition to other tools that implement resolution)

2 Coq - The proof assistant

Coq is an automated proof assistant. While it can execute *proof scripts*, it is primarily used as interactive tool, to build up proofs. Proofs are written using *tactics*, which can be thought of as programs that take an expression and produce evidence for that expression. Tactics have such familiar names as *reflexivity*, *symmetry*, and *induction*. Tactics can be built up from other tactics, and higher order tactics are called *tacticals*. They all encode concepts from informal proofs.

Some of the proof-writing can be automated, but this requires knowing the appropriate tactic for the domain. For example, a series of arithmEtic rewrites that can be expressed as a Presberger arithmetic can be dispatched with the *omega* tactIc. Generally, the proof writer will need to know the high-level direction of the proof in order to effectively use any automation. Automation does not solve the proof for the proof-writer. Instead, it elides some low-level details in order to make the formal proof-writing process more like the informal proof-writing process.

An obvious objection might be, if Coq cannot automate to the point where it's guiding the seaRch, why use an automated proof assistant in the first place? Setting aside any advantages of exported code,[1] the main advantage of using Coq is that the proof-writer can be sure that the proof produced is correct. Coq provides a kind of type-checking on the proof itself, validating each step of the proof. What students may not realize is that this is not how proofs are written in the wild; the type-checker on proofs in the wild are peers who must review this work.

Writing good proofs takes years of experience, and from that experience, intuition. While the basics of proof writing are generally easy to grasp, many students struggle with more nuanced proof techniques (c.f. Pumping Lemma) or with the application of seemingly basic proof techniques to more complex problems (e.g. structural induction).

3 Propositional Logic

In standard fashion, Schoening presents first the syntax and then the semantics of propositional logic. We can model propositional logic in Coq. An important distinction to be made throughout the following exercises is that the language of propositional logic we define is not the same as the language we are writing in. For example, the three basic building blocks of Coq's intuitionistic logic are implication,[2] universal quantification, and inductive definitions [**?**]. Our language of propositional logic will not directly implement implication, nor will it have universal quantification or the ability to directly express inductive definitions. However, will we use all concepts to prove theorems about our language. Thus, our proofs will be written using the full power of Coq, whereas our model of propositional logic will be a restricted language implementing the behavior described in Schoening.

First we define the notion of formula. This is done inductively. The most basic, indivisible unit of a formula is what we will call an atomic formula. Atomic formulas are drawn from the set $\{A_i \mid i \in \mathbb{Z}\}$. The Coq core library has natural numbers built in, so the mapping into Coq simple:

```
Inductive atomic :=
| A : nat → atomic.
```

Now we are ready to define the syntax of a propositional formula:

```
Inductive formula :=
  | Atom : atomic → formula
  | Negation : formula → formula
  | Disjunction : formula → formula → formula.
```

Coq uses *type signatures* to denote the inputs and output of a function. The literals `Atom`, `Negation`, and `Disjunction` are *constructors*[3] for the *type* `formula`.

The type signatures for these constructors should be read as follows : the final consequence in the implication corresponds to the type the constructor is meant to produce. All previous premises correspond to the arguments of the constructor. We say "meant to produce" because it is possible to apply the function to fewer arguments and still have a valid expression in Coq. This is because functions in Coq are *curried*, meaning that any function having arity n can be thought of as a stack of $n-1$ unary functions. Functions applied to fewer arguments than their arity are said to be *partially applied*. This feature will not be used in this tutorial.

[1] Definitions in Coq can be exported to OCaml. This allows a programmer to create code that is proven correct.

[2] In Coq, implication is overloaded to also refer to type annotation. This is in keeping with the Curry-Howard isomorphism, which asserts that proofs are programs and vice versa.

[3] These are not constructors in the sense of object-oriented programming. The Coq language, Gallina, is implemented OCaml and although OCaml has an object system attached, the above are not analogous to the OO paradigm. If you have not used a language like OCaml before, it would be helpful to forget anything you know about OO programming terminology. :)

Note that nullary functions are permitted. If, for example, we decided to elide the enumeration of atomic formulas, we could have defined `Atom : formula`, so it would take no arguments. We will need to differentiate atoms when we get to the semantics of propositional logic, but want to note that this sort of elision is what permits program verification to use SAT solvers to transform expressions of state from predicate logic into propositional logic.

We can now write a function to check whether an input is a well-formed formula:

```
Definition WFF (f : formula) : boolean := true.
```

What's going on here? Our type definition for `formula` specifies the definition of a well-formed formula. If we try to either provide an argument to `WFF` that is not a `formula` or to construct an expression using our defined constructors for `formula` that does not obey their type signatures, the Coq type-checker will throw an error. Since we get this syntactic check "for free," we know that any input that satisfies the type-checker will be well formed, so the definition of `WFF` is trivial. Students should test well-formedness for themselves with the following commands and observe what happens:

```
Eval simpl in WFF true.
Eval simpl in WFF (Negation (Atom (A 1)) (Atom (A 2))).
```

Now that we have defined the syntax of propositional logic, we can define its semantics. Colloquially, we think of semantics as "what we really mean." For example, "Please pass me the coffee" and "Por favor, me passe o café" are semantically equivalent; they have different syntaxes, since they are in different languages, but we can think of them as evaluating to the same meaning. Statements in natural language can evaluate to infinite meanings, but all formulas propositional logic can only evaluate to two : entities we'll call "true" and "false," known collectively as *truth values*.

We want to avoid any intuitive notion of truth and falsehood here, since it turns out that neither are as simple as they might first appear. In the context, we'd rather think of them as two distinct tokens whose meaning is defined solely by their interaction with the logical connectives we've defined. We could just as easily have called them "foo" and "bar." We chose the boolean values because there already exists a library of predefined functions over booleans in Coq, some of which will be useful later. We could just as easily have defined the truth values as being drawn from the set $\{0, 1\}$. The two conventions are equivalent; we just need to ensure that any functions we use in the context of evaluating truth values have equivalent formulations over $\{0, 1\}$ and vice versa. For example, we might be interested in the `min` function over $\{0, 1\}$. We can define a `min` function for booleans as follows:

```
Definition min (a b : bool) : bool :=
  match a, b with
    | _, false | false, _ ⇒ false
    | _, _ ⇒ true
  end.
```

This function establishes an ordering on the set $\{true, false\}$. Note the underscores in the match statement – these simply denote that we can match anything on that position. We could use a token instead, but since we do not need to use that token, we can just use the underscore as a wildcard. Also note that we can match two patterns on the same line if their output is equivalent.

Before we can evaluate arbitrary formulas, we will need to know how to evaluate our atomic formulas. We define an assignment to be a map from atomic formulas to booleans. We can express this is as a list of tuples:

```
Definition assignment := list (atomic * bool).
```

We now need a way to look up the truth for a particular atomic formula. The definition for `beq_atomic` is provided in `prop.v`, but is not relevant to understanding our model of propositional logic in Coq. Throughout

this tutorial there will be functions and theorems that are used, but not defined here in this document. We will periodically enumerate them as part of the student's "toolbox" for writing proofs.

```
Fixpoint find_assignment (a : atomic) (assignments : assignment) : option bool :=
  match assignments with
    | nil ⇒ None
    | (b, truth_value)::tail ⇒ if beq_atomic a b
                               then Some truth_value
                               else find_assignment a tail
  end.
```

Note that the above definition does not return type `bool`, but instead type `option bool`. `option` is a *parameterized type* that takes another type as its argument. This means that `option nat`, `option atomic`, and `option formula` are all meaningful types. The `option` type has two constructors : a nullary constructor `None` and a unary constructor `Some`.

The option type exists to make a computation total. Consider the case where the assignment list does not contain the atomic formula a. What should the computation return? In some sense, it should be undefined when a is not in *assignments*. However, that would make `find_assignment` as partial function, which cannot be evaluated in Coq. Instead, we use the `option` type as a stand-in for this case[4].

Now we can define a procedure for evaluating an arbitrary formula and an assignment to its truth value:

```
Fixpoint eval_formula (phi : formula) (a : assignment) : option bool :=
match phi with
| Atom foo ⇒ find_assignment foo a
| Negation foo ⇒ match (eval_formula foo a) with
                   | None ⇒ None
                   | Some x ⇒ Some (negb x)
                 end
| Disjunction foo bar ⇒ match (eval_formula foo a) with
                          | None ⇒ None
                          | Some x ⇒ match (eval_formula bar a) with
                                       | None ⇒ None
                                       | Some y ⇒ Some (orb x y)
                                     end
                        end
end.
```

The use of the `option` type allows us to define the notion of a *suitable* assignment for a formula:

```
Definition suitable (f : formula) (a : assignment) := eval_formula f a <> None.
```

An assignment is suitable if all the atoms of f have assignments in a. Let's generate some assignments we know to suitable for some formulas to see how things work:

```
Eval simpl in eval_formula (Negation (Disjunction (Atom (A 1)) (Atom (A 2))))
                           (((A 1), true)::((A 2), true)::nil).
Eval simpl in eval_formula (Negation (Disjunction (Atom (A 1)) (Atom (A 2))))
                           (((A 1), true)::((A 2), false)::nil).
Eval simpl in eval_formula (Negation (Disjunction (Atom (A 1)) (Atom (A 2))))
                           (((A 1), false)::((A 3), false)::nil).
```

[4]An alternative approach would be to restrict the domain of assignments allowed as arguments to `find_assignment`. We would do this by defining another function `contains_assignment` that would search through an assignment to test whether a was in *assignments*. Although this definition is much closer to one we would use in class and in informal proofs, it requires defining the input types as dependent types. That is, the type of *assignments* would depend on the value of a. We plan on updating this, now that we better understand how to program with dependent types, but since the current incarnation of this project was done with options, we're sticking with that for now.

Now we have sufficient abstractions to solve Homework 1, Problem 1 in Coq. However, we have not yet covered the basics of proving theorems in Coq.

3.1 A Basic Proof Example – Introducing Tactics

In order to prove anything in Coq about a system such as propositional logic, we need to iterate between modeling the system and proving theorems about the system. Thus far, we have focused on modeling. The model is the set of definitions of propositions and types whose structure we will use in our proofs.

In order to prove anything of value about our system, we are going to need tactics. Tactics are small programs that are used to manipulate proofs.

Consider the examples shown above. The `Eval simpl` command reduced the first expression down to `Some false`, using its limited understanding of the function `eval_formula`. `simpl` is a kind of *tactic*. What it actually does is not very simple at all, but at a high level, it reduces a term by evaluating expressions that have computations in them. This capacity is limited however – the interested reader should try to define two functions that they know are equivalent, but whose order of operations differ and see how `simpl` handles each definition. Spoiler alert : reductions (what we just called "computations"); function application is one kind of reduction[5].

Let's start by stating a theorem. In Coq, the `Theorem`, `Lemma`, and `Example` keywords are just synonyms for objects that have proofs associated with them. We can what we learned from evaluating the expression to state that the expression is equal to the thing it evaluated to:

```
Example simple_proof :
  eval_formula (Negation (Disjunction (Atom (A 1)) (Atom (A 2))))
                         (((A 1), true)::((A 2), true)::nil) = Some false.
```

When we tell Coq to evaluate the above expression (i.e. execute the "proof-assert-next-command-interactive" function, which is bound to Control-h in the setup guide we provide), we see something like the following:

```
1 subgoals, subgoal 1 (ID 57)

  ============================
   eval_formula (Negation (Disjunction (Atom (A 1)) (Atom (A 2))))
     ((A 1, true) :: (A 2, true) :: nil) = Some false
```

Every proof we write will have something like the above displayed when proving interactively. Everything listed above the double line is something assumed, already proven, or otherwise available to prove whatever appears below the line. We can think of the contents above the line as a problem-specific toolbox for proofs. If something appears there, we can *apply* it below. We move contents below the double line to above the double line using a combination of this problem-specific toolbox and "global" theorems already proven, glued together with tactics.

Moving back over to the proof script we're writing, we prove the theorem by writing the following directly below our statement of the proof:

```
Proof.
  simpl. reflexivity.
Qed.
```

[5] We should also note that reductions here are not reductions in the sense of 601. These reductions include β reductions, from the lambda calculus. The interested reader can look at the `simpl` documentation for more information.

The `Proof` command is not strictly necessary, but does help visually. The use of `simple` happens to be extraneous here; the `reflexivity` tactic will suffice. The `reflexivity` tactic is one of a few tactics that will be used to finish the proof. We can type `Qed` once we're notified that there are no more subgoals, and the proof is complete.

So when should one use `simpl` and `reflexivity`? `simpl` can be used at any time, without producing an error. Sometimes it will do nothing. Sometimes it will produce an unreadable reduction. `reflexivity` should generally be used when the two sides of an equality are syntactically equal. It will attempt to reduce one side if they are not syntactically equal, as in the above example. It absolutely cannot be used if two expressions are syntactically not equal, by semantically equivalent. That is, if we have two expressions that cannot be "evaluated down" to the same syntax, but have equal output for equal input, we will need to do more than just call `reflexivity`.

3.2 Homework 1, Problem 1

The problem can be stated as:

```
Example hw1_prob1 :
    forall (a : assignment) (F G : formula),
        let f := (Negation (Disjunction (Negation F) (Negation G)))
        in suitable f a → eval_formula f a = min (eval_formula F a) (eval_formula G a).
```

The complete solution can be found in $\underline{\texttt{hw1}_{prob1.v}}$. In addition to `simpl` and `reflexivity`, we'd like to highlight the following tactics:

unfold The exercise uses functions `eval_formula` and the variable `f`. `f` can be rewritten using its definition by calling `unfold f`. The student may want to replace a function like `eval_formula` with its definition. If so, they may call `unfold eval_formula`. If the student wishes to replace the definition of `eval_formula` with its name, they may call `fold eval_formula`.

destruct The student will not need to use induction to prove the exercise. We encourage the student to take a moment to think about why this is. Since many expressions in the exercise are simply evaluated to one of `Some true`, `Some false`, or `None`, the student can use destruct to consider each possible outcome of the evaluation. `destruct` may be used over variables or complex expressions. For example, the student will reach a point where the goal contains an expression `Some b`. In this case, the student should call `destruct b`. The student will also see expressions such as `eval_formula F a`; this expression can also be destructed with the call `destruct (eval_formula F a)`.

`eval_formula` and our definition of `assignment` describe the semantics of propositional logic. The semantics provide the meaning of whole given the parts. We defined what a formula is and what is the meaning of the atoms (assignments). In `eval_formula`, we describe how the parts are combined in order to get the meaning of the formula. In a meta context, we'd like to note that semantics are used primarily[6] to make assertions about equality. If two propositional formulas have different surface strings, but evaluate to the same truth value, then they are equivalent. This definition makes sense; when we say that two functions are equal, what we mean is that equivalent inputs will produce equivalent outputs. There are contexts where the definition of equality also considers factors such as the mechanics of a computation, but we will not consider those contexts here.

Since we do not have an inherent notion of equality, we must define it:

```
Definition form_equiv : forall F G a b,
                        eval_formula F a = b ↔ eval_formula G a = b.
```

[6] Are they really used for anything else?

3.3 Homework 1, Problem 2

Definition $\text{some}_a ssignments : assignment := ((A1), false) :: ((A2), true) :: nil.$
Example $\text{suitable}_e x : suitable1(Negation(Atom(A1)))some_a ssignments.Proof.compute; introsH; inversionH.Qed.$
Example $\text{models}_e x : models(Negation(Atom(A1)))some_a ssignments.Proof.compute; reflexivity.Qed.$
Example $\text{sat}_e x : satisfiable(Negation(Atom(A1))).Proof.unfoldsatisfiable; existssome_a ssignments; compute; reflexivity$
Example $\text{unsat}_e x : unsatisfiable(Negation(Disjunction(Atom(A1))(Negation(Atom(A1))))).Proof.unfoldunsatisfiable.$
$False- > None = Somefalse). + intros.inversionH0. + applyH1.applyH.reflexivity.Qed.$
Lemma $\text{empty}_a ssignment_n ot_s uitable : forallF, eval_f ormulaFnil = None- > suitable1Fnil.Proof.inductionFas[|F'IHF|$
Lemma $\text{suitable}_i nvariant_n egation : forallFa, suitable1Fa < - > suitable1(NegationF)a.$

4 First-order Logic

5 To Do

a. Coq tactic cheatsheet

b. Tutorial cheatsheet (the "toolbox")

c. change option types over to dependent types

d. get notation for conjunction working

e. installation instructions (should have automated script. windows users may be screwed. maybe we should have a web server to test and submit work?)