A Tale of Two Proofs: On the (Variable) Verbosity of Coq Proofs

Riley Moher

1 Introduction

In my search for a demonstrated application of Coq [1] to prove some theorem, I was drawn to practical scenarios. Along this path, I came across the paper *Formal proofs applied to system models* [1], in which the authors attempt to demonstrate the usage of Coq for verifying properties of system models. In the paper, they give the example of Failure Mode and Effects Analysis method (FMEA) in Nuclear equipment. Specification of these systems can become very complex and long, even when the number of rules involved is relatively small. In these cases, the use of software to verify the system may be desirable, however, software in general, is also complex and may be challenged, as it is not guaranteed to function as designed; one must trust the developer of said software. This motivates the usage of formal methods, like proof assistants, to verify functions and properties of these systems. In the paper, they give an example of a steam generator that may be used in a nuclear plant, and exemplify the usage of Coq by proving properties of the system.

In this paper, I will provide a similar proof of the nuclear steam generator system motivated by that of [1]. Specifically, I will examine their simplified proof utilizing Coq's tautology tactic and demonstrate how one may choose to more explicitly and more meticulously specify this proof by using various proof tactics within Coq.

2 Modelling the System

In [1], the authors wish to demonstrate a simple and desirable property for their example system of a nuclear power plant's steam generator system, that any produced flow is consumed, and any consumed flow is produced. This can correspond with the intuitive desire of such a system to not leak, and to have all fluids accounted for, which is especially important in more potentially dangerous systems like a nuclear reactor.

In order to prove a property such as this, one must have a way of specifying the movement of fluids within the system and between its components. In [1], the authors specify how this could be modelled in a modelling language like SysML or others, and

[1] https://coq.inria.fr/

they then describe their own DSL to automatically translate such a specification into Coq. For the purposes of this paper, we will not focus on the DSL, but rather the resultant Coq encodings. The encoding of this System in Coq is done via inductive types [2] Beginning with the simplest type, we will define a type to aid in uniquely identifying objects, called the identity, which will be an alias for the natural numbers:

```
Definition Id : Type := nat.
```

This can then be used in our first inductive Type, System:

```
Inductive System : Type :=
  | system : Id -> System
  | comp_system : list Component -> System -> System
```

We define this type as inductive through the keyword `Inductive` and declare it as a type, followed by its inductive definition (what follows `:=`). This inductive definition consists of two possible constructors: the simple system constructor, and the component-system constructor. The simple system constructor simply takes an Id and produces a unique system based on it. Using this construct we could define a minimally-defined system, where all we do is uniquely identify it, but do not specify any of its properties, like so:

```
Definition system1 := system 1.
```

Coming to the next constructor, its typing tells us that we are taking a list of components and some existing base system, and combibing them to form a new system. In this way, we could extend our previous minimal specfication of system1 to define a new, more complex system like:

```
Definition system2 := comp_system [comp1, comp2] system 1.
```

However, one may notice that we have not provided any definition for the Component Type. This is because our definition of the System type previously introduced is a truncated one. The full System type also includes definitions for all of its inductive sub-types:

```
Inductive System : Type :=
  | system : Id -> System
  | comp_system : list Component -> System -> System
with Component : Type :=
  | component : Id -> Component
  | comp_component : list Component -> Component -> Component
  | func_component : list Function -> Component -> Component
with Function : Type :=
  | function : Id -> Function
  | dflow_function : DirectedFlow -> Function -> Function
with DirectedFlow : Type :=
  | directedflow : Id -> DirectedFlow
  | pflow_directedflow : PhysicalFlow -> Direction -> DirectedFlow -> DirectedFlow
with PhysicalFlow : Type :=
  | physicalflow : Id -> PhysicalFlow.
```

As well as the inductive type Direction:

[2]Note that I present a slightly simplified inductive type encoding compared to the one originally presented in [1], in line with the scope of this paper to focus more on the proof mechanisms rather than the specific style of encoding systems.

```
Inductive Direction : Set := input | output.
```

One may immediatly see a pattern here of inductive types having similar constructors to that of the truncated System Type definition, they have a simple base constructor which returns an entity given an identifier, and more complex constructors which denote composition of those types from other entities. Using a definition like this, one can model complex nested components and functions within systems.

3 Explaining and Comparing The Proofs

Having explained the types involved in this formal modelling of systems, I now focus on the proof of the property for an example system of a nuclear power plant's steam generator system, that any produced flow is consumed, and any consumed flow is produced. Firstly, let us construct our example system:

```
Definition steam := physicalflow 0.

Definition water := physicalflow 1.

Definition circuit := func_component
[
dflow_function (pflow_directedflow steam input (directedflow 0)) (function 0);
dflow_function (pflow_directedflow water output (directedflow 0)) (function 1)
]
(component 0).

Definition drying_block := func_component
[
dflow_function (pflow_directedflow steam output (directedflow 0)) (function 2)
]
(component 1).

Definition inlet := func_component
[
dflow_function (pflow_directedflow water input (directedflow 0)) (function 3)
]
(component 2).

Definition steam_generator := comp_system
[
circuit;
drying_block;
inlet
]
(system 0).
```

Here, we first construct the necessary elements of our system, firstly defining relevant fluids of water and steam, then defining the components which handle them, and finally constructing the system consisting of those components. Intuitively, we can see that the desired property of our system is true by observing the fact that the circuit's inflow of steam has a corresponding outflow via the drying block, and the circuit's outflow of water has a corresponding inflow via the inlet. However, since we are interested in formally proving this property of our system, an intuitive explanation is not enough.

To construct our proof, we must show that:

- Any produced flow in the system is consumed in the system.
- Any consumed flow in the system is produced in the system.

These form our two goals, which can be explicitly constructed via definitions, and later applied to specific instances of systems to be verified, like our `steam_generator`. These definitions are constructed as:

```
Definition ConsumedFlowsAllProduced (s : System) : Prop :=
  forall (p: PhysicalFlow),
  containsDirectedFlow_List p input (get_directed_flows_system s)
  -> containsDirectedFlow_List p output (get_directed_flows_system s).
```

```
Definition ProducedFlowsAllConsumed (s : System) : Prop :=
  forall (p: PhysicalFlow),
  containsDirectedFlow_List p output (get_directed_flows_system s) ->
  containsDirectedFlow_List p input (get_directed_flows_system s).
```

Each of these definitions is a proposition which may hold for some System `s`, with their structure clearly mirroring each other (via the switch input/output) in their implication. [3]. I will present the proofs for each of these goals, beginning with the more verbose proof of the first, followed by the less verbose proof of the second.

3.1 A verbose proof

The goal that all produced flows are consumed in our steam generator system is written via the application of our definition to `steam_generator`, along with the Goal keyword:

```
Goal (ConsumedFlowsAllProduced steam_generator).
```

After stating the goal, we use the Coq keyword Proof to begin our proof:

```
Goal (ConsumedFlowsAllProduced steam_generator).
Proof.
...
```

At this point, our goal remains very high-level, and we proceed by utilizing our first proof tactic, `intros`, to introduce a new variable into the context, such that our proof view now looks like:

```
p: PhysicalFlow
-------------------
1/1
containsDirectedFlow_List p input
  (get_directed_flows_system steam_generator) ->
containsDirectedFlow_List p output
  (get_directed_flows_system steam_generator)
```

This first presents us with our context, containing the variables over which our goal is universally quantified, and the goal, a singular goal in this case (denoted by the 1/1).

Next, we will utilize our function definitions to call the functions defined in our Goal and simply our goal into a more concrete form. This is done via the `simpl` proof tactic. This now leaves us with the proof view:

[3] the full code, including helper functions which have been omitted here for brevity, can be found at githubhere

```
p: PhysicalFlow
-------------------
1/1
p = steam /\ input = input \/
(p = water /\ input = output \/
 (p = steam /\ input = output \/
  (p = water /\ input = input \/ False \/ False) \/ False) \/
 False) \/ False ->
p = steam /\ output = input \/
(p = water /\ output = output \/
 (p = steam /\ output = output \/
  (p = water /\ output = input \/ False \/ False) \/ False) \/
 False) \/ False
```

Through all the disjunctions and conjunctions, it may be difficult to parse this goal and determine what it is we really need to prove. However, this goal can be proven by a simple tautology: it does not matter what value p takes on, this goal is now equivalent to a universal truth. However, how we reach that tautology is where the power of variable verbosity in Coq becomes evident. In the simple case, our proof can be finished by calling the tautology tactic `tauto`. However, what if we wish to know or exemplify what identities or lemmas were applied, and how? For example, we may wish to take advantage of the fact that:

$$\forall P, P \vee \bot \equiv P$$

And we could model this same property of propositional logic in Coq as the assertion:

```
assert (lor_false : forall p : Prop, (p \/ False) <-> p).
```

However, note that this is an *assertion*, and if we wish to apply this assertion in our proof to simplify our goal, we must prove this assertion as well. Since this is a very simple Goal to prove, our proof can be a very simple one:

```
Proof.
intros. tauto.
Qed.
```

Now, we can apply this result to our current goal to simplify it, via the `rewrite` tactic:

```
Proof.
...
rewrite lor_false.
...
Qed.
```

and we may further recursively apply this result to our goal if we wish by invoking the `repeat` tactic:

```
Proof.
...
repeat rewrite lor_false.
...
Qed.
```

In this way, we can tautology proofs that are more human-readable to build up to our final result. For instance, if we assert and prove several logical identities and apply them to our previously simplified goal, the goal may be reduced to:

```
(p = steam) \/ (p = water) -> (p = water) \/ (p = steam)
```

Which is much more convincingly proven by tautology. Furthermore, we can also introduce variables to simply expressions to be even more human readable, such that this goal reads:

```
P \/ Q -> Q \/ P
```

When applying assertions in this way, the proof view of Coq becomes much more human readable, providing us with all the proven assertions and variables in the context, followed by the simplified goal. Our full proof view in this case is now:

```
impl_ident: forall p q : Prop, (p -> q) <-> ~ p \/ q
neg_disj_ident: forall p q : Prop,
                ~ (p \/ q) <-> ~ p /\ ~ q
excluded_middle_1: forall p : Prop, p \/ ~ p <-> True
excluded_middle_2: forall p : Prop, ~ p \/ p <-> True
lor_refl: forall p q : Prop, p \/ q <-> q \/ p
lor_false: forall p : Prop, p \/ False <-> p
false_lor: forall p : Prop, False \/ p <-> p
lor_true: forall p : Prop, p \/ True <-> True
true_lor: forall p : Prop, True \/ p <-> True
and_false: forall p : Prop, p /\ False <-> False
false_and: forall p : Prop, False /\ p <-> False
and_true: forall p : Prop, p /\ True <-> p
true_and: forall p : Prop, True /\ p <-> p
input_is_input: input = input <-> True
input_not_output: input = output <-> False
output_not_input: output = input <-> False
output_is_output: output = output <-> True
p: PhysicalFlow
P: Prop
HeqP: P = (p = steam)
Q: Prop
HeqQ: Q = (p = water)
_____________________________________________________
1/1
P \/ Q -> Q \/ P
```

Notice that we could simply conclude the proof with that `tauto` tactic, or continue in this very explicit manner by invoking an assertion (and proof) of the reflexivity of the logical OR, such that the final part of our proof reads:

```
Proof.
...
intros Antecedent.
apply lor_reflexivity.
apply Antecdent.
Qed.
```

3.2 A not-so-verbose proof

Having provided the steps to very explicitly prove all produced flows are consumed in our steam generator system, I now produce the much less verbose proof that all consumed flows are produced in our steam generator system[4]:

[4]For those interested in comparing the full text of each proof, please refer to the full Coq file in githubhere

```
Goal (ProducedFlowsAllConsumed steam_generator).
Proof. intro. simpl. tauto. Qed.
```

4 Learnings & Future Directions

For someone coming from an applied ontology background, familiar with automated theorem provers, the level of verbosity that Coq is capable of may seem extreme. Through this paper, I have demonstrated that while Coq can indeed be very verbose, it is not always necessary. This flexibility, in my view, can be seen as an advantage in terms of how human readable one wishes their proofs to be. In this paper, we demonstrated the proof on a relatively simple system with very few components and functions. However, one could imagine extremely complex systems and correspondingly deep inductive type definitions for such systems. In these cases, proving properties of the systems may very well be achievable via some tautologies, but knowing that Coq affords the user a great deal of flexibility in how those tautologies are broken down and structured is very encouraging. An intriguing avenue for future exploration here is in the application and extensions of a DSL, much like that mentioned in [1]. The distinction here being that the DSL does not need apply only to the encoding of systems, but could also be extended to the methods used in proving properties of the system. Because of the very structured and organized nature of Coq's proof tactics and library of lemmas, one could envision a DSL to take other other existing system verification methods and translate them into the formal methods of Coq.

References

[1] Contejean, É., Samokish, A.: Formal proofs applied to system models. In: JFLA 2023-34èmes Journées Francophones des Langages Applicatifs, pp. 121–133 (2023)