CS 591K1 Dependently Typed Automated Systems
A Primer on Interactive Theorem Proving (in Coq)

Kinan Dak Albab

23 January 2019

1 Content

The contents of this lab are as follows:

2 What is Coq?

Coq is an *interactive* theorem *assistant*. We will focus on three main components (languages) in it:

1. **Gallina:** a functional programming language used for specification, modeling and writing running (dynamic) code.
2. **Ltac:** a domain-specific language for writing proof tactics. A proof tactic is a function or small program, that carries out (several) steps in a proof. It takes a state of the proof as input, and modifies it (hopefully moving the proof in the right direction) to produce a new state of the proof.
3. **A Bridge:** that brings the two together, basically allowing you to write theorem statements and proofs with Ltac, about entities and programs defined in Gallina.

For moderately interesting theorems (i.e. facts you want to prove are valid) and beyond, Coq will not be able to automatically proof the theorem out of the box. Coq relies on (sometimes extensive) human interaction to produce such a proof. The interaction is done in two ways:

- Coq (when used correctly) can take care of many labour-intensive small steps of the proofs (i.e. symbol manipulations), while relying on the human prover to direct the proof in some way. Coq has capabilities to search the proof tree, run certain automatic solvers for some theories, and support proofs by reflection.
- Coq will automatically check the proof to see if it is correct. If the Coq compiler accepts a proof as correct, then it must have been to verify that it follows from first principles (the logical core of Coq), which you learn more about in the foundations portion of this class.

Theorems in Coq are usually specifies as some implication between a set of hypothesis, and a conclusion. A valid Coq proof, is a correct sequence of steps that starts from these hypothesis, and reaches/proves the conclusion, potentially using tactics and external lemmas. A proof step represents a valid transition between two proof state. Each proof state contains several hypothesis formulae (things that are assumed to be true), and several goal formulae (things we want to demonstrate are true given the hypotheses).

Coq relies on type-checking to check the proof. You will see more on how this work in the foundations. Here is a (very) high-level preliminary example showing why this is intuitive: imagine if you have an already proven lemma that you want to apply in some context, since the lemma is proven to be correct, any results from applying it should be correct. However, one may still apply the lemma incorrectly, for example, by applying it even though one of its conditions (i.e. Hypothesis) does not hold in the particular context of application. Coq assigns each lemma (including sub expressions and hypothesis) a type, and ensures that invocations of a lemma must be provided with matching types.

We recommend starting to use coq with the CoqIDE for beginners. This will avoid spending time learning advanced emacs usage and shortcuts, and simplify the interaction with Coq. However, we will be using emacs for all the other systems (since they do not have IDEs), so we recommend moving to emacs after you are familiar with Coq.

3 A Primer on Functional Programming (in Coq)

All of the systems we will look at rely extensively on functional programming. Functional programming differs from imperative programming in several important aspects:

- The notion of state is not central, and is absent from most pure functional programming languages. Variables are just names you assign to values, as such, they are not mutable, you cannot have aliasing, or pass-by-reference.
- Functions are the central pieces of programming, they can be passed around as parameters to other functions. They can return (construct) other functions. They are treated as first-class-citizens. In the systems we are looking at, functions have (very) express headers, that specify the type of the parameters, as well as return values.
- In our systems, types mean more than in popular imperative programming languages. Types can be refined arbitrarily, and can be defined inductively, polymorphically, or using other types. These are all big words, but they will start to mean something as we move along.
- Looping and if-else conditions are not as prominent as they are in imperative programming, instead, functional languages we will see rely on recursion and pattern-matching.
- As a result of these differences, some data-types or operations that are very easy to use in imperative programming become far more cumbersome in functional programming. An obvious example we will use a lot is arrays and lists. Functional programs rely on linked lists, since they can be defined very nicely inductively, and are immutable. Arrays can be very hard to express and use effectively in a functional language.

The following subsections survey common concepts in functional programming in Coq's syntax. The code snippets are taken from **intro_functional.v** under lab1/code.

3.1 Inductive types (in Coq)

The most common example used to introduce functional programming is defining and operating on a list. Let us start by a simple definition, a linked list made out of natural (non-negative integers) numbers.

```
Inductive my_natural_linked_list : Type :=
  | Empty
  | Node (node: nat) (remaining: my_natural_linked_list).
```

The **Inductive** key word specifies that the following definition is inductive. The keyword is followed by the name of what is being defined: **my_natural_linked_list**, followed by the type of what is being defined, in our case we are defining a **Type**.

Now this type is inductive, because it can be built from the bottom-up inductively. The type has two constructors (i.e. functions that produces instances of that type). The first is **Empty** which returns an empty list. The second is **Node** which takes in a natural number named **node** and another list of our type called **remaining**, and chains them together to form a new list of our type.

We can use these constructors to create lists as we see fit. Note the lisp-style parenthesizing used by Coq, parenthesis surround both function name and parameters.

```
Check Empty.
Check (Node 0 Empty).
Check Node 1 (Node 0 Empty).
Check my_natural_linked_list.
```

The type definition can be made polymorphic, so that we do not have to re-define lists for each content type.

```
Set Implicit Arguments. (* Ask coq to implicitly infer types from the parameters passed to functions *)
Inductive my_list (A: Type) :=
  | my_nil
  (* in functional lingo, the element and list constructing a list are called head and tail *)
  | my_cons (head: A) (tail: my_list A).
Arguments my_nil [A]. (* Ask coq to deduce my_nil's type argument from the context *)
```

Turns out these inductive types are pretty helpful, Coq allows us to perform induction on them during proofs. They are very easy to operate on using recursive functions, and pattern matching. Here are two more interesting definitions:

```
Inductive nat :=
  | zero (* base natural number is zero *)
  | S (n: nat). (* Successor function, our representation of + 1 *)
(* This is called the unary representation, Coq uses this internally *)

(* A "nullable" generic type, allows us to specify that it can be
   either None (null), or Some given value "val". *)
Inductive option (A : Type) : Type :=
  | None
  | Some: (val : A).

(* An alternate style of definition clarifies the functional nature
   of constructors *)
Inductive option (A : Type) : Type :=
  (* None takes no parameters, and immediately constructs an object of type "option A") *)
  | None : option A
  (* Some requires that you pass some object of type A as parameter, in order
     for it to return (construct) an object of type A *)
  | Some : A -> option A.
```

3.2 Recursion and Pattern Matching

Let us see an example of how we can operate on some inductive type. Let us define a function that computes the size of objects of type my_list, by recursively unchaining its head, and adding one to the size of the remaining tail.

```
Fixpoint my_list_size (A: Type) (list: my_list A) : nat :=
  match list with
  | my_nil => 0
  | my_cons element list' => 1 + (my_list_size list')
  end.

Compute my_list_size (my_cons 3 (my_cons 2 (my_cons 1 my_nil))). (* Gives 3 *)
```

The **Fixpoint** keyword specifies that we are defining a **recursive** function. Gallina (Coq's specification language) only allows you to write obviously terminating recursive function. Otherwise, you will have to prove to Gallina that your function terminates. This is important in Coq, since these functions can be used within proofs, and having non-terminating functions can make things complicated.

The **match ... with ... end** syntax above is used for pattern matching (some what similar to a switch case statement). The object being pattern-matched is specified right after the match keyword, and the matching patterns options are specified after with separated by a bar. In our case, my_list has two potential patterns (constructs), either my_nil or my_cons.

Here is another example that sums the element of a list of natural numbers.

```
Fixpoint my_list_sum (list: my_list nat) : nat :=
  match list with
  | my_nil => 0
  | my_cons element list' => element + (my_list_sum list')
  end.

Compute my_list_sum (my_cons 3 (my_cons 2 (my_cons 1 my_nil))). (* Gives 6 *)
```

3.3 Helpful Coq Features

Fortunatly, Coq has a large library modules, usually organized into **theories**. A theory contains modeling or specification code (like our list definition), useful operations on these definitions (like our functions above), as well as helpful proven lemmas and facts about these definitions and operations, and tactics that simplify dealing with them in proofs.

Coq allows you to import and use modules, and provides auxiliary commands for checking the type of some defined entity, printing its definition, locating where it is defined, and searching for entities by patterns.

```
Require Import List. (* Import the List module *)
Require Import Frap. (* Import the Frap library: contains many useful helpers and tactics
                        In order for this import to work, you must specify the path to
                        this library correctly, either in the Makefile for command line
                        use, or in a _CoqProject file for editing in emacs and CoqIDE.
                        Check code/lab1 to see how it is done. *)

Print list. (* Shows you the entire definition of list, very similar to ours *)
Check length. (* Shows you just the header of function length *)
Search (list _ -> list _ -> list _). (* Search for anything with the given type expression
                                         in its header. This will find several functions with
                                         a matching type signature, including app: a function
                                         that takes two lists, and returns a third one made
                                         from appending the second to the first.
                                         underscores are matched with any expression. *)
Locate app. (* finds where the function app is defined *)
```

3.4 Overview of Some Issues in Reasoning About Programs

At a high level, reasoning about programs using theorem assistants involves several important parts:

1. Modeling of the program to reason about.
2. Modeling of the context and execution of the program.
3. Specifying the desired properties to reason with about the program.
4. Proving that the desired properties hold in our model.

These parts interact heavily with each other, and thus, the specific details about how one part is implemented may affect all the other parts. We will see that there are several levels of modeling, reasoning, and verifying programs, ranging from the very high level, to the very concrete.

3.5 Levels of Modeling

At the highest level, correctness proofs in algorithm textbooks can be considered a form of program reasoning. They model the programs/algorithms using some high level pseudo-code, and ignore or abstract a lot of the execution details you would encounter implementing and running these algorithms on a real machine. These proofs are very important, standard algorithms are important building blocks of Computer Science, it is critical for them to be accurate. However, they only really reflect algorithms at the design level. Implementations of these algorithms can be buggy and erroneous, either due to human error in implementation, or due to unexpected side-effect of running on real machines (e.g. hyper-threading, timing, caching)

At the lowest level, one can think about verifying compiled programs at the machine instruction level, and including in the modeling all the side-effects and details of the underlying Computer Architecture, Operating System, and other complex interactions, while having a very complex and detailed specifications describing the desired property. Such a setting would be extremely complex, and the proofs and reasoning will require a lot of work. Additionally, since the modeling and the specifications are very involved, the chances of making errors while encoding them is increased. Finally, such proofs are unlikely to generalize, in the sense that proving a similar property about the same program, but compiled or run on a different architecture, or with different assumption, may require a radically different proof such that the time and effort needed to write the new proof is similar to what's needed for the original proof.

In our labs and homework, we will walk this fine line, and try to remain somewhere in between, such that the picture is not very complex and time consuming, but at the same time, the exercises remain practical and exciting. We will see some of these trade-offs in greater detail as we move along.

3.6 Functional vs Imperative Programs

One important issue that we will start running into immediately is verification of functional vs imperative programs. As you will see in the later parts in the foundations, functional programming has been the traditional vessel in which automated systems have been developed and used. Functional programming lends itself well to the kinds of formalism we will be seeing through out the course. Since all of the systems we are going to study are by default geared towards functional programming, you will find it much easier to reason about functional programming, while reasoning about imperative programs will require a good amount of plumbing.

In particular, if you write the program you are interested in reasoning about in Gallina. Coq theorems revolving around this program become easier to prove. In particular, because Coq's logical core is compatible with Gallina, so that manipulating proof terms with Gallina statements in them is easy, and going through the program during the proof is in some ways "built-in". One down side of this is practicality, you must write and run your programs using Gallina, as rewriting them in your language of choice may introduce bugs. Coq allows you to extract runnable OCaml programs from Coq implementations, but this remains insufficient for real use. Additionally, pure functional programming as a paradigm suffers from certain well-known inefficiencies (such as relying on linked lists instead of arrays), and attempting to alleviate these issues using some post-compilation or run-time environment may introduce bugs, and adds burden on the provers and modelers.

Instead, what we would like to do, is reason about programs written in a language of choice (e.g. Python or Java), using the proof tool-set that Coq gives us. This is possible to do, Coq allows you to define other language's syntax as inductive types, and parse it within Coq. However, Coq does not "understand" such custom modelling out-of-the-box the way it does Gallina. Thus, you will have to add modeling code that specifies the meaning of the statements written in the target language in a way Coq can understand (this is called formal semantics), and add helpful tactics and tools to help you simplify reasoning about these programs that contain additional features beyond what Coq easily supports out of the box (e.g. aliasing, shared memory, concurrency, inheritance, etc).

All this can be done, and we will see different ways of doing it for small examples through out the course. One intuitive way is to provide some translation layer that describe the meaning of statements in the target language, by reducing them to statements in Gallina, this is a form of embedding that we will see later on.

3.7 Who will guard the guards?

One last issue worth mentioning is about the correctness of your specifications and modeling. Coq ensures that theorems are valid within the modeling. But what if the modeling is not reflective of real-life, or that the theorem statement (specification) does not really mean what we thought it meant. This is a very difficult problem, and cannot be solved by more of the same style of reasoning, since that will just create a new level of modeling and specifications.

There are several approaches one may take to mitigate this. One can rely on existing modelling of common scenarios, and rely on wildly used specifications for popular properties, with the hope that any mistake made in them would have been noticed by one of the many people who used it. However, I think there is a deeper bright spot here. Specifications and modeling are a lot simpler than the actual program they accompany. A good program is not only correct, but efficient, easy-to-use, fault-tolerant, etc. All these desired properties increase the complexity and size of the program. Specifications, proofs, theorems, and modeling, are different. They are static objects that are used at compile time to ensure correctness, and are thrown away afterwards. They are never really run, and therefore can afford to implement things in simple but inefficient way. Finally, they can be expressed descriptively, as opposed to constructively. Programs must produce or construct outputs from inputs, but specifications need only judge whether a property is true, or alternatively, describe what makes a property true.

An example demonstrating these two points is as follows: imagine you have a complicated SAT solver that you would like to verify, the SAT solver is a program that takes as input a Boolean formula made out of variables and logical connectives (ands, ors, and nots), and produces as an output either a satisfying formula that assigns values (true or false) to the variables, such that the input formula evaluates to true, or, produces some special value (null or false) if no such evaluation exists (the formula is always false no matter what values you give to variables). The SAT solver can be extremely complex, with several heuristics and optimizations implemented to improve its efficiency. On the other hand the specification can be very simple, descriptive, and without any limitation on complexity. For example, one possible way to specify correctness of this SAT solver, is to require that for every input formula, these two properties are true: (1) if the SAT solver returns some assignment, then plugging that assignment in should make the formula true, (2) if the SAT solver returns null, then every possible assignment must make the formula false. Implementing this specification is a lot easier than a real SAT Solver, the first part can be implemented with a function that replaces variables with their assigned values, then evaluate the resulting Boolean expression (a bunch of ands and ors). The second part can be implemented with a (very inefficient) function that goes through every possible assignment, and requires that the formula evaluates to false on that assignment.

4 Verifying Binary Search Trees in Coq

The code here is taken from **binary_tree.v**.

First let us define what a binary tree is using Coq's inductive type. We will restrict our attention to trees of natural numbers of now, since we need the data type to be comparable to be able to implement binary search.

```
Inductive tree :=
| nil (* Empty tree *)
| cons (left: tree) (root: nat) (right: tree). (* left tree / root \ right tree *)
```

We can define simple linear search on the tree (traversing all the nodes looking for some given attribute as follows).

```
Fixpoint search (t: tree) (n: nat) : bool :=
  match t with
  | nil => false (* no element is contained in an empty tree *)
  | cons l e r => (* could be equal to the root, in the left tree, or right tree *)
      let r1 := (Nat.eqb e n) in (* eqb: equal boolean, can also use =? *)
      let r2 := (search l n) in
      let r3 := (search r n) in
        orb r1 (orb r2 r3)
  end.
```

Alternatively, we can define binary search, also as a recursive function.

```
Fixpoint binary_search (t: tree) (n: nat) : bool :=
  match t with
  | nil => false
  | cons l e r =>
      if (Nat.eqb e n) then true (* is the root *)
      else if (Nat.leb e n) then binary_search r n (* must go right *)
      else binary_search l n (* must go left *)
  end.
```

Keep in mind that binary_search requires the tree be a binary search tree to work correctly. A search tree corresponds to a sorted array in some ways. All elements on the left of some element must be smaller than or equal to it, while elements on the right must be greater than or equal to it.

One possible way to specify what it means for our binary_search algorithm above to be correct, is to require that it produces the same output as search on binary search tree. In coq, we can write this as a theorem as follows.

```
Theorem binary_search_correct:
 forall (t: tree) (n: nat), (is_binary_search t) = true -> (binary_search t n) = (search t n).
 (* for every tree and natural number, if that tree is a search tree, then binary searching
 or regular linear searching for the natural number in the tree must produce the same answer.
    Alternatively, one can say: for every binary search tree and natural number,
    running binary search or linear search on both must give the same output *)
```

This is very interesting in several ways. First, this is a demonstration of the last point in the previous session, we are less likely to make a mistake implementing linear search than we are implementing binary search since it is less complicated. Also, although we are using linear search during the proof, the implementation (and thus runtime) of binary search remains unmodified. Finally, since Coq must have been able to determine that Gallina programs terminate in order to accept them, we do not need to worry about cases where one or both programs do not terminate.

We have not defined what a binary search tree is. There are several ways to do this, for example, using inductive relationships, or by extending the inductive type with additional information to ensure all instances of it are search trees by constructions. However, we will start with something simpler, we can write a Gallina functions, that takes as input a tree, and determine if it is a binary search tree or not, by checking that all elements on the left are less than or greater than every root, and symmetrically to the right. Our implementation here is not efficient, the function is never run, instead it is used statically in proofs, we care more about simplicity here than efficiency.

```
Fixpoint is_binary_search (t: tree) : bool :=
  match t with
  | nil => true
  | cons l e r =>
    let r1 := is_binary_search l in
    let r2 := is_binary_search r in
    let r3 := (all_less l e) in (* helper function defined in binary_tree.v *)
    let r4 := (all_greater r e) in (* helper function defined in binary_tree.v *)
      andb r1 (andb r2 (andb r3 r4))
  end.
```

Finally, the proof can be carried out in several ways. However, the main intuition of it boils down to this: if binary search tree decided to go left, then the element could not have existed in the right sub tree. This is true because, (1) the algorithm only decides to go left if the element we are searching for is less than the root. (2) all elements on the right are greater or equal to the root. The symmetric statement is true for going right as well.

We organize our proof by first proving these two statements above as a lemma, each having exactly the same proof, by induction on the structure of the tree. The proof of the main theorem is also by induction on the tree, the base case is obvious, but for the inductive step, we perform case analysis on the result of comparing the element we are looking for and the root. We have three cases, in the first, the root and element are equal, and the algorithm returns true, this is trivial. In the second, the algorithm goes left, and in the third the algorithm goes right. We used the above lemma to show that running linear search on the right and left subtrees respectively is guaranteed to return false, then rely on the inductive hypothesis to finish the proof. The exact proof can be found in binary_tree.v

```
Lemma not_in_less:
  forall (t: tree) (e n: nat),
    (all_less t e = true) /\ n > e -> search t n = false.
(* for all trees and naturals, if everything in a tree is less or equal to an element,
   and that element is less than a number, that number cannot exist in the tree *)
```

5 Tactics Reference

In our proofs, we used three kinds of tactics: some from Coq's standard library, some from the Frap library, and some custom tactics developed specifically for this proof (found in binary_tree_helpers.v).

The standard library tactics we used are: (1) trivial (2) specialize (3) apply (4) clear (5) unfold (6) cases (7) rewrite (8) assert (9) symmetry. Most of these tactics have descriptive names, but you can find their exact effect and syntax (including any optional parameters) at https://coq.inria.fr/refman/coq-tacindex.html

Tactics we used from Frap library include:

1. simplifiy: automatic rewriting and unfolding of complex expressions, splitting up of premises and conclusions, and other obvious symbols manipulation and simplification.
2. induct: similar to Coq's standard induction, but enhanced to support other forms of induction.
3. linear_arithmetic: attempts to prove the goal from any combination of hypothesis by using the rules of linear arithmetic (naturals with +, ¡, = no multiplications).

4. propositional: attempts to prove the goal from any combination of hypothesis by using inference rules of (intuitionistic) propositional logic, also has the side effect of rewritting/simplifying boolean formulas.

Finally, the last tactics we used are custom tactics defined in binary_tree_helpers.v:

1. expand_ands: expands hypothesis on the form ($x \&\& y \&\& z$ ... = true) to proposition hypothesis on the form (x = true, y = true, z = true). Here: && is (dynamic) Boolean and, a dynamic operation that is run inside Gallina function to perform boolean and, and all of x, y, z have type boolean as opposed to prop.

2. comparison_bool_to_prop: transforms any dynamic natural comparison to its static proposition equivalent: for example ($x >=? y$ = true) is transformed to ($x >= y$), and ($x =? y$ = false) is transformed to ($x <> y$).

6 Recommended and Additional References

Recommended references will help you catch up on things you missed, or get added explanations about what was covered. They will help you replicate this work on your own, as well as work on your labs homework and pragmatic-focused projects.

Recommended references:

1. Code associated with Chapters 2 to 3 in Frap https://github.com/achlipala/frap

2. First 9 pages of the Frap book http://adam.chlipala.net/frap/frap_book.pdf, or up to the end of Chapter 3 for additional detail.

Additional references:

1. Notes from UPenn on the Basics of Coq https://softwarefoundations.cis.upenn.edu/lf-current/Basics.html

2. Old tutorial by Jean-Christophe Filliatre http://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/filliatre_sl1.pdf

3. A Gentle Introduction to Functional Programming in ML https://www.cs.nmsu.edu/ rth/cs/cs471/sml.html