CS 591K1 Dependently Typed Automated Systems
Reasoning About Programs in Coq: Coq Syntax and Tactics

Kinan Dak Albab

30 January 2019

1 Content

The contents of this lab are as follows:

1. Binary search trees in Coq.
2. Tactics Reference.
3. Recommended and Additional References.

2 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 *)
```

3 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. simplify: 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.
5. equality: tactic that encpasulates equality and its property (symmetry, transitivity).

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 ($x <> y$).

4 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 3 in Frap https://github.com/achlipala/frap
2. Chapter 3 of the Frap book http://adam.chlipala.net/frap/frap_book.pdf

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