CS 591K1 Dependently Typed Automated Systems
Reasoning About Programs in Coq: Certified Programming and Levels of Verification

Kinan Dak Albab

6 February 2019

1 Content

The contents of this lab are as follows:

1. Proving $1 + ... + n = \frac{n(n+1)}{2}$.
2. Specifying and proving correctness of reversing a list.
3. Certified programming alternatives.
4. Recommended and Additional References.

2 Sum 1 to n

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

Coq can be used to reason about programs, as well as mathematical functions and objects! One noteable example in literature is using Coq to verify the four color theorem [1].

We will demonstrate this by proving a well known sum, you probably saw this in the analysis of various algorithms (such as buble sort).

$$1 + ... + n = \frac{n(n+1)}{2}$$

To avoid reasoning about division and remainders, we will modify the theorem to the following equivalent statement.

$$2 * (1 + ... + n) = n(n+1)$$

In order to formally define and prove this theorem, we need to define the sum $1 + ... + n$ formally in Coq. One such approach is to use a recursive functions.

```
Fixpoint sum_to (n: nat): nat :=
  match n with
  | 0 => 0
  | S n' => n + sum_to n'
  end.
```

Our theorem statement therefore becomes:

```
Theorem sum_to_thm : forall (n: nat),
  2 * sum_to n = n * (n+1).
```

The proof for this theorem is a very simple inductive proof, it mimics the traditional proof you might have seen in a course on discrete maths. The proof proceeds by induction on n. The base case simplifies to the trivial goal $0 = 0$, while the inductive case requires a bit of symbols manipulation (distributing multiplication on the first addition inside the recursive function).

Note that the proof relies on the **ring** tactic, as opposed to linear_artihmetic. This is because the theorem statement contains a non linear multiplicative term $n * (n + 1)$.

2.1 Continuation Passing Style

Note that our implementation of sum_to is not tail-recursive. This does not matter in this case, since we never really use sum_to except in proofs. However, if we are reasoning about real programs, we would like to encode the program in an efficient way.

Tail recursion is a specific form of recursion, where the recursive call is the last operation executed in the function. This allows compiles and interpreters to optimize the execution of the program, by reusing the current stack frame for the recursive call, as opposed to creating a new one. This is possible since tail recursion guarantees that the recursive call is the last statement in the function, and hence the intermediate variables and state of the function is never used during or after the recursive call. Read more here https://stackoverflow.com/questions/33923/what-is-tail-recursion

Continuation passing style (CPS) is one general recipe for turning certain recursive function into tail-recursive equivalent functions. CPS is used by certain compilers as an intermediate representation. The main idea is to change the control flow of the recursive function: instead of doing work after the recursive call is completed, that remaining work is packaged as a function (known as contiuation), and passed as a parameter to the recursive call, the recursive call must guarantee that this continuation will be executed on the final value it computes, and return the output of the continuation. Read more here https://en.wikipedia.org/wiki/Continuation-passing_style

Here is a CPS implementation of our sum function:

```
Fixpoint sum_to_CPS (n: nat) (C: nat -> nat): nat :=
  match n with
  | 0 => C 0
  | S n' => sum_to_CPS n' (fun R => C (n + R))
  end.
```

Notice that the function takes an additional continuation function C as a parameter, whose takes a single input parameter of type nat and produces an output of type nat. We make the following observations:

1. In the recursive case, notice how summing n to the result of the recursive call is “lazily” scheduled for execution later as part of the new continuation passed to the recursive call.

2. The new continuation constructed in the recursive case must utilize the continuation passed to the function. Otherwise, sums previously scheduled by previous calls in the recursion will be ignored and not executed, causing an incorrect output.

3. The base case does not output 0 directly as in before, instead it outputs the result of the continuation on 0. Effectively this starts execution of the sums that were scheduled by previous recursive call.

4. For this particular example, this is equivalent to a (more efficient) version that takes in a running sum instead of a continuation, such that the running sum is incremented by n, and passed to the next recursive call, and such that the base case returns the running sum as opposed to 0.

5. Initially, when sum_to_CPS is first call by an external user, the user must provide the identity function as a continuation, to get the correct result.

We can prove that the CPS implementation is equivalent to the original implementation.

```
Theorem CPS_correct: forall (n: nat) (f: nat -> nat),
  f (sum_to n) = sum_to_CPS n f.
```

The prove here proceeds by induction on n. The crucial step is to specialize/apply the inductive hypothesis with the correct continuation function. In particular, we cannot just use the same f from the theorem statement, instead we must use (fun R =¿ f (n+1 + R)).

Proving the theorem by induction requires that we state the theorem for all continuation, since the continuation function changes during recursive call. You can try to prove the simpler (weaker) correctness theorem that only applies to the identity continuation, but the inductive step will not go through. Proving a weaker desired theorem by proving a stronger theorem (one that implies it) is very common. Depending on the context, this may be referred to as strengthening the invariant (as in the case of Hoare Logic or reasoning about transition systems), or strengthening the inductive hypothesis (as in this inductive case).

3 Levels of Specification

One common theme we will see throughout this course is the trade-off between ease-of-proof (or efficiency of proof in case of automation) and the level of specification in our theorem statements. Here, we will explore a classic example demonstrating this trade-off.

Consider this list reversing function:

```
(* ++ is Coq-provided notation for appending two lists *)
Fixpoint my_reverse (A: Type) (l: list A): list A :=
  match l with
  | nil => nil
  | cons head tail => (my_reverse A tail) ++ (cons head nil)
  end.
```

We want to convince ourselves that this function is correct. In other words, that this function abides by some desired specification. In many cases, we may be happy proving that the function abides by some specific property, that may not imply what we may see intuitively as full correctness, but still provides us with a strong enough guarantee, is sufficient to the context in which we use the function, or can be combined with other properties - that may be established by other formal proofs, testing, inspecting the code, etc - to give us the desired correctness.

In this case, we can formulate several properties that the reverse function must satisify. Each is meaningful on its own, but each is weaker than the next one.

We start by providing a property on the shape of the returned list, namely that it must be of the same length as the input list:

```
Theorem my_reverse_correct1: forall (A: Type) (l: list A),
  length l = length (my_reverse l).
```

This does not guarantee correctness. For example, the reverse function could have produced a list consisting of the same element, duplicated as many times as the length of the input length, and it will satisfy this property, regardless of what the input list is. We may be ok with this, since we can manually inspect the reverse function's implementation, and establish that the function never adds any element that did not exist in the original list to it. However, this is error-prone, especially for large functions.

We can improve the property by requiring that any element in one list is an element of the other:

```
(* In is a function that coq gives us for checking if an element is in a list *)
Theorem my_reverse_correct2: forall (A: Type) (l: list A),
  length l = length (my_reverse l) /\
  (forall (e: A), (In e l) -> In e (my_reverse l)) /\
  (forall (e: A), (In e (my_reverse l)) -> In e l).
```

This still does not guarantee full correctness. The function may return any random shuffle of the original list, and that will satisfy this property. The property does not enforce that duplicates have the same count in both input and output lists.

Another improvement is to specify the property inductively. In other words, assume that the function is correct for smaller lists, and show that it combines correctly for larger lists.

```
Theorem my_reverse_correct3: forall (A: Type) (l: list A),
  length l = length (my_reverse l) /\
  (forall (a: A), my_reverse [a] = [a]) /\
  forall (l1 l2: list A) (a1 a2: A),
    (my_reverse (l1 ++ [a1; a2] ++ l2)) = (my_reverse l2) ++ [a2; a1] ++ (my_reverse l1).

(* Alternatively *)
Theorem my_reverse_correct3': forall (A: Type) (l: list A),
  length l = length (my_reverse l) /\
  forall (l1 l2: list A) (a: A),
    (my_reverse (l1 ++ [a] ++ l2)) = (my_reverse l2) ++ [a] ++ (my_reverse l1).
```

We do not need to consider the case of an empty list, since the length preservation ensures that the output would also be empty in that case. This formulation turns out to be equivalent to the next one (it implies our intuitive notion of full correctness). However, that equivalence itself contains enough details that it may merit its own proof. In the general case, it may not be clear whether such inductive formulation really expresses our intuitive notion of full correctness.

Our final formulation is equivalent to the previous one, but is a direct encoding of our intuitive correctness criteria for reversing a list. An element of the input list located at index i would be located at index $n-i-1$ in the output list:

```
(* nth_error <list> <index> is a Coq-provided function that returns an option type:
   1. None: if the index is out of bounds
   2. Some <element>: if the index is in bounds, and matches <element>.
*)
Theorem my_reverse_really_correct: forall (A: Type) (l: list A),
    length l = length (my_reverse l) /\
    forall (i: nat), i < length l -> nth_error l i = nth_error (my_reverse l) (length l - i - 1).

(* Alternatively *)
Theorem my_reverse_really_correct': forall (A: Type) (l: list A),
    forall (i: nat), nth_error l i = nth_error (my_reverse l) (length l - i - 1).
```

Note that even this formulation requires some thought. If we restrict i with $i < length l$, we must require that the length of the two lists is equal, otherwise, we may accept lists that contain the reverse as a prefix, and other elements is a suffix.

The proofs for all these theorem proceeds by induction on the list (for the third formulation, it actually performs induction on two lists l1 and l2). The proofs become slightly more complicated as the specification becomes richer. We use several Coq-provided lemmas and facts about ++, In, and nth_error in our proofs to keep them short. In particular, associativity of ++ and cons, and how they distributes over In and nth_error.

4 Certified Programming

An alternative approach to verifying programs is to write the program with enriched (dependent) input/output (and intermediate) types, such that the types themselves encode our correctness and preconditions on the input. Coq allows us to write such Certified Programs. The Program module and command simplifies reasoning about certified programs, and proving that they really abide by the type. For more information on “Program”, look at https://coq.inria.fr/refman/addendum/program.html.

The Program command will automatically generate and attempt to solve proof obligations that guarantee that the program type checks against the specified dependent types. Program will likely proof base cases and simple goals automatically, while failing and requiring user intervention on complicated inductive cases. You can help Program proof things automatically, by providing richer intermediate types, splitting your function into helpers, and providing hints to the Coq hint engine. We will see examples of how this style of certified programming can be automatically checked later in ATS and Agda.

Here are two certified implementations of both the sum_to and reverse functions from the previous sections:

```
Program Fixpoint sum_to (n: nat): {r: nat | 2 * r = n * (n+1) } :=
  (* Notice that the return type looks like a set with some predicate function that depends on input va
  match n with
  | 0 => 0
  | S n' => n + sum_to n'
  end.

Check sum_to_obligation_1. (* Base case related obligation is already proven *)
Obligation 2 of sum_to. (* must proof inductive step related obligation *)
    <proof>
Defined sum_to. (* This will only be accepted if all obligations are proven *)

Program Fixpoint my_reverse (A: Type) (l: list A): {l': list A | length l = length l'} :=
  match l with
  | nil => nil
  | cons head tail => (my_reverse A tail) ++ [head]
  end.
```

5 Tactics Reference

The new standard library tactics we used are:

1. pose: for adding an already proven theorem/lemma as a premise.
2. destruct: destructing an inductive type in a premise into its possible constructors, this will spit out several goals, each corresponding to a single constructor, very useful for proving things with a disjunction/or as a premise.
3. ring: similar to linear_arithmetic but supports multiplication, arithmetic with multiplication is undecidable, this tactic is not complete and almost always needs help.

You can find their exact effect and syntax (including any optional parameters) at https://coq.inria.fr/refman/coq-tacindex.html

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 subset dependent types in Frap
   https://github.com/achlipala/frap/blob/master/SubsetTypes.v

Additional references:

1. Subset and Sigma Types http://adam.chlipala.net/cpdt/html/Subset.html
2. Code associated with dependent inductive types in Frap
   https://github.com/achlipala/frap/blob/master/DependentInductiveTypes.v

References

[1] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.