CS 591K1 - Homework 1: Coq

Kinan Dak Albab

Due: 13 Feb 2019 - 11:55 PM

Instructions

This is a long homework with several purposes. In addition to making you familiar with Coq's syntax and tactics. It also will ensure you get sufficient practice in functional programming, get exposed to the methodology and some of the issues regularly faced when formally verifying and reasoning about programs, as well as bridge some of the content between Foundations and Pragmatics. Start working early and post your questions to Piazza!

The next assignments will be shorter and more focused, but will have less detailed instructions and hints.

Full Credits

This lab consists of 5 problems, the first of which is a Warmup. For full credit, you need to fully solve 4 problems. The warmup and first two problems are required, you can choose between the last two problems. You can solve an extra problem, or extra parts of problems for extra credit.

If you are stuck in any proof, you can skip the particular proof goal/obligation by using admit. Additionally, if you know how to prove something, but need an extra (reasonable) assumption or push to do it (or need to skip a few steps), you can create a lemma describing the assumption, use admit to skip proving it, and use it in the main proof you were stuck in. This will help you receive higher partial credit for that problem.

You can solve part(s) of an extra problem even if you have not solved all the required parts, this will also help you get some partial credit.

Full credit for this Homework is a 100. However, you can score up to 135 total points if you solve all the problems, or up to 175 if you solve all problems plus and the two bonus parts in Problem 4.

Creating and Running Your Solutions

Make sure you run 'make' in the command line / terminal inside every problem folder before you start writing your solutions for that problem.

Each problem has its own separate directory, you will find several files inside. You should not modify any of them except "Solution.v". Put your solutions inside "Solution.v". You can write and prove as many auxiliary lemmas and tactics as you wish. You will also find other files in each directory: "Check.v" this is used for grading, ignore it. "Problem.v" contains the problem statements, and useful explanations and hints, **read it carefully**.

Some problems may contain "Helpers.v" and/or "Modeling.v". The first contains useful helper tactics you can use to simplify the proofs, with instructions on how to use them. The second contains required modeling code to be able to state and prove the theorem, you do not need to modify the modeling code you are given, but you are encouraged to read and understand it, so that you can carry on with the proofs without much pain.

Submission

You will need to submit a single Zip file. The Zip file must contain one folder per problem, each containing only the "Solution.v" file for that problem. You can submit your files at https://goo.gl/forms/j1P6WSh21H8AcTfj2

Warm up (15 points) - Natural Deductions - Required

This problem is a warm up problem to help you familiarize yourself with the Coq environment, and relate some material from the foundations.

You are required to prove two theorems of propositional logic. The first is the second formulation of DeMorgan's Law, and the second is the forth formulation of DeMogran's law. These proofs must be carried out using Coq's encoding of natural deductions. You can find the pen-and-paper version of these proofs at https://www.dropbox.com/s/li86bh9vgztvfzq/natural-deduction-examples-in-IPC.pdf?dl=0

Before starting these proofs, look at the given proof sample at the start of "Solution.v", and try to compare it against the pen and paper proof in the slides above.

You can prove the first theorem using only intuisinistic (constructive) logic rules, for the second one, you will need to use classical logic's Law of Excluded middle, there are instructions inside "Solution.v" for importing and using classical logic in Coq.

1. **Part 1 - 5 points** Prove DeMorgran's Law 2.
2. **Part 2 - 5 points** Prove DeMorgran's Law 1.
3. **Part 3 - 5 points** At the end of "Solution.v", write your explanation for why Coq has two seemingly similar types, Prop and bool. Write your answer using one or two paragraphs in comments. Look online for some resources on this topic, and use what you learned from class and the above proofs to find key differences between the two types. Hint: does Prop support the law of excluded middle? What about bool?

1 Problem 1 (25 points) - Verification of Functional Programs - Required

You are required to implement and prove interesting properties about a simple functional program: factorial.

1. **Part 1 - 5 points** Implement factorial and prove the freeby theorem:

```
fact (n: nat): nat := ....

Theorem fact_thm1:
    fact 5 = 120.
```

2. **Part 2 - 5 points** Prove that the factorial of any number greater than 1 is even. Hint: induction!

```
Theorem fact_thm2: forall (n: nat),
    n > 1 -> (exists (k: nat), fact n = 2 * k).
```

3. **Part 3 - 10 points** Implement factorial using continuation passing style (CPS), and prove that it is equivalent to regular factorial.

```
Fixpoint fact_CPS (n: nat) (C: nat -> nat): nat := ... (* C is the continuation *)

Theorem CPS_correct: forall (n: nat) (f: nat -> nat),
 fact_CPS n f = f (fact n).
```

Continuation passing style is an important concept in functional programming, it is used during compilation of functional programs as an intermediate level, and it provides a general recipe for transforming functional programs to tail-recursion for efficiency.

Resources: Read more about tail recursion here https://en.wikipedia.org/wiki/Continuation-passing_style. Here is a hands-on overview of continuation passing style in JavaScript, using factorial as an example! http://matt.might.net/articles/by-example-continuation-passing-style/

4. **Part 4 - 5 points** Prove that the factorial of any number greater than 1, computed using CPS, is even! Hint: use the two theorems above and put them together!

```
Theorem fact_CPS_thm2: forall (n: nat),
  n > 1 -> (exists (k: nat), fact_CPS n (fun R => R) = 2 * k).
```

2 Problem 2 (35 points) - Verification of Imperative Programs Using Interpreters - Required

In this problem, you are required to verify correctness of a simple imperative Fibonacci program.

Coq does not understand imperative programs defined in any language, it only understands Gallina! We need a way to explain to Coq what each construct in our language does (this is called "semantics"), so that we can use Coq to reason about programs in our language. One way to achieve this is using an interpreter.

An interpreter is a function/program, that takes as a parameter a program in some language and some representation of memory (e.g. a mapping between variable names and values), and other run time information. The interpreter returns the result of executing the given program in the given environment.

Gallina programs must terminated, any recursive function cannot have infinite / arbitrary recursion. Coq enforces this by checking that every recursive call decreases at least one parameter, while all other parameters remain the same or are decreased as well. This means that our interpreter cannot execute infinite programs, and restricts the expressiveness of our language. This is way our language only uses a very rudimentary notion of iteration: repeat with a fixed number of iterations.

Our toy language and the interpreter for it is defined in "Modeling.v", useful hints and explanations are provided in "Problem.v".

1. **Part 1 - 5 points** Implement a recursive functional fibonacci program in Gallina.

```
Fixpoint fib (n: nat) : nat := ...
```

2. **Part 2 - 30 points** Prove that the interpreted imperative fibonacci program, and the functional fibonacci program are equivalent for all inputs. Hint: You are given a set of lemmas that will help you prove the final theorem, you are not required to prove them, but if you, it will simplify your task considerably! Hint2: look at the hints and explanations in "Problem.v".

```
Theorem fib_iterative_correct: forall (n: nat),
  interpret (fib_iterative n) = fib n.
```

3 Problem 3 - (Very Simple) Compiler and Code Transformation Correctness - 25 points

You are given a very simple model of an arithmetic expression language, and two Gallina functions that transform given expressions to simpler ones. You must prove that these transformations are correct, in other words, for every expression, the original and transformed expressions evaluate to the same thing.

1. **Part 1 - 10 points** Show that the "useless" transformation is correct. The "useless' transformation adds (0 + ¡subexp¿) to every sub-expression in a given expression.

```
(* eval_nat_exp is an interpreter for our expression language
   it is defined in ``Modeling.v'' *)
Theorem useless_is_uselsss: forall (e: natExp) (v: valuation),
  eval_nat_exp e v = eval_nat_exp (uselessTransformation e) v.
```

2. **Part 2 - 15 points** Prove that the constant rewriter/reducer is correct. The constant reducer compiles a given expression into a more efficient equivalent expression, by pre-evaluating any sub-expression whose terms are all known constants.

```
Theorem constants_reducer_correct: forall (e: natExp) (v: valuation),
  eval_nat_exp e v = eval_nat_exp (constantsReducer e) v.
```

4 Problem 4 - Verifying Imperative Programs Using Small-Step Semantics and Hoare Logic - 35 points

You are given a model of an imperative language, with arbitrary while loops, and if statements. The semantics for this language cannot rely on an interpreter. Instead, we use "Small-Step" semantics, which describe the meaning of the language one statement at a time in isolation. The syntax and semantics are provided in "Modeling.v".

Small-step semantics have many advantages: they allow expressing and reasoning about non-terminating programs, they can be extended to reason about non-determinism, randomization, and concurrency.

Additionally, you are given proof rules for "Hoare Logic", a program logic dedicated to reasoning about imperative iterative programs. Hoare logic will allow us to easily reason about complicated iterative programs with state and memory. The proof rules are given in "Hoare.v". The rules can be proven to be consistent with the semantics (i.e. sound) within Coq, to ensure that they are bug-free, but we skip that here.

Finally, we provide tactics for automatically applying and stepping through Hoare Logic proof rules. Careful use of these tactics will ensure that most of the tedious steps in the proof are automated. These tactics and instructions on using them are provided in "Helpers.v". **Read the comments and relevant pieces of the code in Modeling.v, Hoare.v, Helpers.v, and Problem.v for useful hints and explanations**.

Resources: Our modeling, Hoare logic, and helper tactics are (almost literal) adaption of the modeling in the Frap library/book. You can read more about these concepts in Chapter 12 (6 pages) of the Frap book http://adam.chlipala.net/frap/frap_book.pdf

For examples on using the tactics and modeling of Hoare logic, check out the Frap github repository https://github.com/achlipala/frap/blob/master/HoareLogic.v This is a large file with several components. You may skip the soundness proofs for the purposes of this Homework. Pay closer attention to the examples at the end of the file, in particular, the proof that selection sort is correct.

1. **Part 1 - 5 points** Prove the Warmup lemma, this lemma will be useful in your final proof. The lemma states that any sub array of a sorted array is also sorted.

   ```
   Lemma is_sorted_smaller: forall (arr: heap) (startI endI endI': nat),
     is_sorted arr startI endI -> endI' <= endI -> is_sorted arr startI endI'.
   ```

2. **Part 2 - 10 points** Define the loop invariant for merging two sorted arrays. The merge program is defined in "Problem.v". Make sure that the invariant is (1) true before and after the start of every iteration of the loop. (2) Strong enough to imply the final post-condition.

   ```
   Definition merge_invariant (len1 len2: nat): assertion :=
     fun (h: heap) (v: valuation) => ...
   ```

3. **Part 3 - 20 points** Prove that merging is (almost) correct. Given any two sorted arrays of any length, the merge program produces a third array that is sorted, and of equal length to the sum of the two arrays.

   ```
   Theorem merge_correct: forall (len1 len2: nat),
     {{ fun h v => is_sorted h 0 len1 /\ is_sorted h len1 (len1 + len2) }}
     merge_program len1 len2 merge_invariant
     {{ fun h v => (is_sorted h (len1 + len2) (len1 + len2 + len1 + len2)) }}.
   ```

   This does not really guarantee correctness, since incorrect programs can still satisfy it, for example, a program that produces an array of the right size, made out of zeros! This is because the theorem

statement does not relate the content of the two arrays with the content of the file array, instead it only relates their lengths (shapes).

This can be acceptable in certain cases as a simplification. For example, in this program, we can manually inspect the code and find that the resulting array is only ever assigned values from one of the two arrays directly, and never any external values, therefore, it cannot produce an array with new elements.

4. **Part 4 - Bonus - 10 points** Define a different theorem, whose statement you believe gives you full correctness. Your theorem statement must use Hoare Triples, and must relate the content of the input and output arrays. You may start with assuming no duplicates accross the two arrays exist, and refine your statement to take care of duplicates after wards.

5. **Part 5 - Bonus - 30 points** Define a new appropriate loop invariant for your new theorem defined in part 4. Prove that your theorem holds for the given merge function. You are allowed to modify the merge function if that makes this part simpler. **This is significantly harder than part 3, and you must write a lot of the helper functions, lemmas, and definitions yourself. Do not attempt to solve this until you have solved enough problems for full credit first!**