GIST A10

BY ANDREW SIKOWITZ
OVERVIEW FOR A10

• Prove data structures, logic theorems, and numerical theorems correct, using Coq
  • Implement a queue with a single list (simple), and prove numerous theorems about it
  • Implement a queue with two lists (more complicated), and prove the same theorems about it
  • Prove several logical statements by creating a Coq program with the corresponding type
  • Prove a numerical theorem using induction and the built-in \texttt{[ring]} tactic
  • Prove a couple theorems about a defined “backwards list” using induction

• Some theorems will be much harder than others and some will be trivial

• Individual assignment
  • You are allowed to collaborate with others, but acknowledge them and do not share code
A10 DELIVERABLES

• [a10.v], your Coq file with all your Definitions and Theorems

• Make sure this program compiles!
  • Complete all Definitions
  • If you attempt a theorem but do not finish it, your program will not compile
    • In that case, put “Admitted.” instead of “Qed.” to finish the proof
  • Run `make check` before you submit to make sure everything is okay

• If you prove all theorems (no admits) and do not use disallowed tactics, you get full credit
  • Disallowed tactics are anything with ‘auto’ in it: [auto], [tauto], [eauto], etc.
CORE TACTICS: SOLVING GOALS
(See Coq Cheatsheet for more and better explanations)

- Basic Coq Theorem environment:
  - Have context, your hypotheses / assumptions
  - Have goal(s) that you must solve / prove

- reflexivity: Solve goal if it can be (easily) simplified to the form x = x.

- discriminate: Solve goal if it can be simplified to the form x <> y, where x and y are different.
  - Or, solve goal if there is a false hypothesis of the form x = y, (after simplification) where x and y are different

- assumption: Solve goal if it is one of your hypotheses

- trivial: Solve simple goals, such as those solved by reflexivity and assumption.
  - Not necessary to complete this assignment

- contradiction: Solve a goal if your context contains False or contradictory hypotheses
CORE TACTICS: TRANSFORMING GOALS
(See Coq Cheatsheet for more and better explanations)

• **intros**: Whenever you have givens / assumptions, use intros to put them in the context
  
  • Start your proofs with intros, if there’s anything to introduce

• **simpl**: Simplify your (sub)goal, often by evaluating expressions as much as possible

• **destruct**: Split an inductive type (e.g. list) into its different cases, getting goals for each case
  
  • Use – to handle one case at a time. If you already used -, use + or * or --

• **rewrite**: If you have a hypothesis a = b, replace expressions a with b or vice versa, in hypotheses or goals

• **apply**: If you have a hypothesis A -> B, replace goal B with goal A

• **contradict <hypothesis>**: Rather than prove your goal, prove <hypothesis> is False
  
  • Not in the notes or cheat sheet and not necessary for this assignment
PART 1: SIMPLE QUEUE

• Implement a Queue, as specified in [a10.v] documentation, in Coq
  • Should be very simple once you get the hang of Coq syntax

• Prove the theorems about it (eqn1, eqn2, …, eqn8) below
  • Replace “Admitted.” with “Proof.” and “Qed.”, putting your proof between those commands
  • Step through the program (with control-alt-N / control-command-N), proving one at a time

• Proofs should be quite short and relatively simple as well
  • Introductory part of the assignment to get you used to writing Coq proofs
PART 2: TWO-LIST QUEUE

• Implement a two-list queue, as specified in [a10.v] documentation, in Coq
  • Should still be relatively simple, but pay attention to the AF and RI specified
  • Try to implement the functions in a way that makes proofs easier

• Prove the subsequent theorems, once again
  • Some should be relatively simple still
  • If you can’t evaluate with `Simpl` (e.g. run into match) try destructing on what is being matched
  • [eqn8_equiv] is the hardest proof so far. Use the strategy above and take the [++] hint seriously
    • Consider making your own lemmas with “Lemma”
    • Recommendation: do this part last. We are covering related material Tuesday, 11/20/18

• Also implement CounterEx, a module representing a counter example to a false theorem
  • If `simpl` is not evaluating a function, use `unfold <function_name>`
PART 3: LOGIC + PROOFS AS PROGRAMS

• Prove several logical statements by creating a Coq program with the corresponding type

• See lecture 22 for information on this part of the assignment

• Quick Summary:
  • Logical statement $A \rightarrow B$ represented as function $A \rightarrow B$ (argument type $A$, return type $B$)
  • Logical statement $A \land B$ represented as $\text{conj} A \ B$ (inductive type with one case)
  • Logical statement $A \lor B$ represented as $\text{or_introl} A \ | \ \text{or_intror} B$ (inductive type, two cases)
  • Logical statement $\neg A$ represented as function $A \rightarrow \text{False}$
    • $\neg \neg A$ is $(A \rightarrow \text{False}) \rightarrow \text{False}$
PART 3: LOGIC + PROOFS AS PROGRAMS

• Basic format:
  • Replace “Conjecture” with “Definition”
  • Replace the period at the end with := fun <args> => <body>

• Example:

Definition logic0 : forall (P Q : Prop),
(P \ Q) -> P
:= fun (P Q : Prop) p_and_q =>
    match p_and_q with
    | conj p q => p
    end.
PART 4: INDUCTION

• Some more proofs, this time using the induction tactic
  • When doing induction, try to get the answer in a form in which you can use the IH

• When doing numerical proofs (involving addition / multiplication), use ring tactic
  • ring can do a lot, but not induction
  • Examples of things ring can solve:
    • $x + 0 = x$ [additive identity]
    • $x \cdot (y + z) = x \cdot y + x \cdot z$ [distributivity of multiplication over addition]
    • $(a + b) + c = a + (b + c)$ [associativity of addition]
    • $x \cdot y = y \cdot z$ [commutativity of multiplication]
PART 4: INDUCTION – BACKWARDS LIST

• Looks confusing, but that’s partially due to the backwards names

• Basic idea: Store at the “head” of the list the conceptually last element in the list
  • Adding an element to the “end” of the list is efficient, but not to the “front” of the list
  • (snoc t h) can be interpreted as putting h at the end of the list t
  • This is mainly a conceptual change: notice the constructor is equivalent to the normal list one
    • [length] and [app] implementations are similar to their list counterparts as well
    • Do not let the odd data structure prevent you from reasoning about proofs like you normally do

• You may have to use ring here as well