CS 591K1 Dependently Typed Automated Systems
Verifying Imperative Programs with Hoare Logic

Kinan Dak Albab

13 February 2019

Acknowledgments

The contents of this lab are mostly taken and heavily inspired by the section on Hoare Logic in the "Formal Reasoning About Programs" MIT course and book by Adam Chlipala: http://adam.chlipala.net/frap/frap_book.pdf
Additionally, the code is mostly taken from the associated code from the FRAP repository:
https://github.com/achlipala/frap.

1 Content

All of the code samples are taken from within lab4/code.
The contents of this lab are as follows:

1. Modeling Imperative Programs.
2. Hoare Logic.
3. Custom Tactics.
4. Termination Proofs.
5. Recommended and Additional References.

2 Modeling Imperative Programs

Unlike previous labs, we will be reasoning about programs written in our own toy imperative language. The first step is to define what this language is in Coq. When reasoning about programs written in Gallina, this step is not needed. Coq already "understands" what Gallina statements are and what its syntax is.

We need to define two things to be able to reason about our toy language (or any custom language) effectively in Coq:

1. Syntax: We need to define what constructs our language supports. We can do this using inductive types, so that we are effectively defining a minimal tree-like structure of our language. Additionally, we can use Coq notation features to define a human-friendly syntax for our language, and transform it into instances of our inductive type.

```
(* Defining a single statement in our language *)
(* exp and bexp are previously defined types of arithmetic and boolean expressions *)
Inductive cmd :=
| Skip
| AssignVar (x : var) (e : exp)
| AssignMem (e1 e2 : exp)
| Seq (c1 c2 : cmd)
| If_ (be : bexp) (then_ else_ : cmd)
| While_ (inv : assertion) (be : bexp) (body : cmd)
(* Think of this more as an annotation
   something we will add to our programs that help us analyze them. *)
| Assert (a : assertion).
```

2. Semantics: We need to tell Coq what the statements in our language "mean". So that we can use Coq to reason about their effects. There are several ways to define and encode semnatics, you can read about here https://en.wikipedia.org/wiki/Semantics_(computer_science)

   Interpreters One intuitive approach is to define an interpreter for our language: a native Coq/Gallina function that takes as parameters a statement/program in our toy language, and an environment (some representation of memory or state), and returns what we specify to be the result of the program in the environment.

   This has certain advantages (simplicity), but has several drawbacks: Our interpreter must be an obviously terminating function, otherwise Coq will not allow us to define it. This restricts our language, since we cannot represent and interpret non-terminating programs, or programs that are not "obviously" terminating. We use interpreters to encode the semantics for portions of our language that we know must terminate, one such example is arithmetic and boolean expressions.

```
(* heap represents a linear memory layout
   valuation assigns values to variables by name *)
Fixpoint eval (e : exp) (h : heap) (v : valuation) : nat :=
match e with
| Const n => n
| Var x => v $! x
| ReadMem e1 => h $! eval e1 h v
| Plus e1 e2 => eval e1 h v + eval e2 h v
| Minus e1 e2 => eval e1 h v - eval e2 h v
| Times e1 e2 => eval e1 h v * eval e2 h v
end.
```

   Small-step Operational Semantics There are other approaches to defining semantics, one notable approach is to use "operational semantics". Operational semantics have different variants, including "big-step" and "small-step" semantics. We will use small-step semantics in this lab.

   The main idea behind "small-step" semantics, is to define the semantics at the level of a single statement/command in our language (syntax-driven definition), so that every rule in our semantics only covers a "single step" in a program's execution. This is most obvious when defining the semantics of iteration constructs (such as while loops): "small-step" semantics encode a single iteration of the loop, as opposed to the entire execution of the loop. You can read more about "small-step" semantic and the differences between it and "big-step" semantics here https://cs.stackexchange.com/questions/43294/difference-between-small-and-big-step-operational-semantics

```
(* Small step semantics *)
Inductive step : heap * valuation * cmd -> heap * valuation * cmd -> Prop :=
| StAssign : forall h v x e,
  step (h, v, AssignVar x e) (h, v $+ (x, eval e h v), Skip)

| StWrite : forall h v e1 e2,
  step (h, v, AssignMem e1 e2) (h $+ (eval e1 h v, eval e2 h v), v, Skip)
| StStepSkip : forall h v c,
  step (h, v, Seq Skip c) (h, v, c) (* Skip in a sequence is skipped *)
| StStepRec : forall h1 v1 c1 h2 v2 c1' c2,
  step (h1, v1, c1) (h2, v2, c1')
  -> step (h1, v1, Seq c1 c2) (h2, v2, Seq c1' c2) (* step in the first statement in a sequence *)

| StIfTrue : forall h v b c1 c2,
  beval b h v = true
  -> step (h, v, If_ b c1 c2) (h, v, c1) (* If condition is true *)
| StIfFalse : forall h v b c1 c2,
  beval b h v = false
  -> step (h, v, If_ b c1 c2) (h, v, c2) (* Else *)

| StWhileFalse : forall I h v b c,
  beval b h v = false
  -> step (h, v, While_ I b c) (h, v, Skip) (* Case where loop terminates *)
| StWhileTrue : forall I h v b c,
  beval b h v = true
  -> step (h, v, While_ I b c) (h, v, Seq c (While_ I b c)) (* One iteration of the loop *)

| StAssert : forall h v (a : assertion),
  a h v -> step (h, v, Assert a) (h, v, Skip). (* assertion annotations have no effect *)
```

3 Hoare Logic

Hoare Logic is a powerful logical system for reasoning about imperative programs. The central concept in Hoare Logic is "Hoare Triple", which consists of three components: a precondition, a program, and a post-condition. We will use the syntax $\{\{P\}\}\ c\ \{\{Q\}\}$ to refer to Hoare Triples with precondition P, program c, and post-condition Q.

Intuitively, if the Triple $\{\{P\}\}\ c\ \{\{Q\}\}$ is valid, that means that running the program c on any state and inputs satisfying precondition P will yield a state and output satisfying post-condition Q, provided that c terminates. This is sometimes called "partial correctness", since it does not enforce termination.

Hoare logic consists of a set of syntax-driven rules, which allow you to prove that a "Hoare Triple" is valid. The rules allow you to traverse the program, such that after every rule application, you consume some statement in the program, and acquire.modify the precondition or post-condition for the program up to that statement.

```
Inductive hoare_triple : assertion -> cmd -> assertion -> Prop :=
(* Skip keep the precondition the same, for any precondition *)
| HtSkip : forall P, hoare_triple P Skip P

(* Assignment extends the old precondition with the new value of the variable *)
| HtAssign : forall (P : assertion) x e,
  hoare_triple P (AssignVar x e) (fun h v => exists v', P h v' /\ v = v' $+ (x, eval e h v'))

(* Writing to memory extends the old precondition with the new value in memory *)
| HtWrite : forall (P : assertion) (e1 e2 : exp),
  hoare_triple P (AssignMem e1 e2)
    (fun h v => exists h', P h' v /\ h = h' $+ (eval e1 h' v, eval e2 h' v))

(* Sequencing is very intuitive:
```

```
The precondition of the sequence is that of the first statement.
The post-condition is that of the second.
The post-condition of the first statement must match. *)
| HtSeq : forall (P Q R : assertion) c1 c2,
  hoare_triple P c1 Q
  -> hoare_triple Q c2 R
  -> hoare_triple P (Seq c1 c2) R

(* Post-condition of an if statement is that of its body or else *)
| HtIf : forall (P Q1 Q2 : assertion) b c1 c2,
  hoare_triple (fun h v => P h v /\ beval b h v = true) c1 Q1
  -> hoare_triple (fun h v => P h v /\ beval b h v = false) c2 Q2
  -> hoare_triple P (If_ b c1 c2) (fun h v => Q1 h v \/ Q2 h v)

(* Relies on a "loop invariant" I
read more here: https://stackoverflow.com/questions/3221577/what-is-a-loop-invariant *)
| HtWhile : forall (I P : assertion) b c,
  (forall h v, P h v -> I h v)
  -> hoare_triple (fun h v => I h v /\ beval b h v = true) c I
  -> hoare_triple P (While_ I b c) (fun h v => I h v /\ beval b h v = false)

(* Assert does not change the precondition, but requires that it implies the assertion *)
| HtAssert : forall P I : assertion,
  (forall h v, P h v -> I h v)
  -> hoare_triple P (Assert I) P

(* Frame or Consequence Rule: you can arbitrarily weaken the post-condition or strengthen
the precondition of a valid Hoare Triple *)
| HtConsequence : forall (P Q P' Q' : assertion) c,
  hoare_triple P c Q
  -> (forall h v, P' h v -> P h v)
  -> (forall h v, Q h v -> Q' h v)
  -> hoare_triple P' c Q'.
```

4 Custom Tactics

Coq provides several features for automating proves. In particular, it provides useful chaining and iteration constructs, it supports pattern matching on the goal, and allows for writing custom tactics that can then be used in any proof.

Here is an interview of some of the common chaining features:

- **[tactic 1]; [tactic 2]; ... [tactic n]** using semicolon ; we can chain applications of tactics together. Chaining here means that every goal produced by the [tactic 1] is passed to the second tactic, which may try to prove them, make progress on them, add new goals, or fail on some goals. If it did not fail, all the remaining goals are passed to the next tactic, and so on.
- **try [tactic]** try is similar to a try-catch statement in some imperative languages. It is useful for use during chaining. If [tactic] proves some goal(s), this try statement will too. However, if [tactic] fails on some goals, this try statement will unroll the changes made by that tactic, and proceed with the proof without failing.
  Note: try can be chained in two different ways:
  1. **try ([tactic 1]; [tactic 2]) ...**: tries to apply both tactics to all goals, if either failed on some goal, the changes made by both tactics on that goal is unrolled and passed forward.

2. **try [tactic 1]; [tactic 2] ...**: tries to apply [tactic 1] to all goals, if it fails, its changes are unrolled. All remaining goals are passed to [tactic 2], failures in [tactic 2] will not be caught by try. This is equivalent to writing: (try [tactic1]); [tactic 2]

- **[tactic 1]** || **[tactic 2]** || **...** || **[tactic n]** this will try to apply the current goal(s) to each of these tactic from left to right. If one of the tactics failed on some goal(s), that goal is forwarded to the next tactic in the —— chain. Otherwise, if a tactic made progress on some goal(s) but did not prove it, or created some new goal(s), that goal is **not passed** to any of the following tactics in the —— chain.
- **repeat** [**tactic**] repeats applying the tactic to all goals. For every goal, if the tactic proved the goal, that goal is proven and discarded. If the tactic made progress on the goal, it is passed to the tactic again (repeated). If the tactic failed, or did not make any progress, the goal is retained beyond the repeat statement. When all goals have been proven, or stopped making progress, the repeat statement is terminated. Repeat is associative to the right, similar to try.
- **fail** causes a failure. This is useful to use inside try and repeat chains, to force termination or to roll back effect of previous tactics.
- **idtac** the identity tactic, has no effect.
- **all:** [**tactic**] applies the given tactic to all remaining goals (not just the current goal(s)). Goals that are out of focus (due to use of +, -, ++ etc) are not affected.

Coq allows you to perform pattern matching on the goal, the syntax for doing so is on this form:

```
(* match the goal with one of the following patterns from top to bottom,
   if a pattern is matched, the corresponding tactic is executed, and
   the match statement is terminated. *)
match goal with
    (* the next pattern looks for an assumption matching <pattern>, H will be matched with the
       name of the assumption, you can use H within the resulting tactic *)
    (* If more than a single assumption matches the pattern only the first (top one) is used. *)
    | [ H: <pattern> |- _ ] => [tactic <possibly uses H>]
    (* similar to the above, but can match to any assumption that has some sub expression
       that matches the pattern, without the entire assumption matching the pattern. *)
    | [ H: context[ <pattern> ] |- _ ] => [tactic <possibly uses H]
    (* matches the conclusion to be proved with a pattern (can use context) *)
    | [ |- <pattern> ] => [tactic <possibly uses H]

    (* Examples *)
    (* looks for an assumption that exactly matches p \/ q *)
    | [ H: p \/ q |- _ ] => destruct H
    (* looks for an assumption that matches p \/ <anything> *)
    | [ H: p \/ _ |- _ ] => destruct H
    (* looks for an assumption that matches p \/ <anything>, and apply
       some lemma with what <anything> was matched with *)
    | [ H: p \/ ?q |- _] => apply <something> with q
    (* looks for an assumption that has p \/ <anything> inside it, and
       apply some lemma with what <anything> was matched with *)
    | [ H: context[p \/ ?q] |- _ ] => apply <something> with q
    (* matches the conclusion of the goal with p /\ <anything> *)
    | |- p /\ _ ] => propositional
    (* looks for a variable that appears inside a conjunction in the
       conclusion, and inside a disjunction in some assumption *)
    | [ H: context[?p \/ _] |- context[?p /\ _ ] ] => propositional
end. (* if no pattern was matched, a failure is produced *)
```

You can use any of these features inside proofs, or to define custom tactics. A custom tactic is defined using the command **Ltac**. It must be given a name, and can be defined to take several parameters, the types of parameters are dropped. Here are a few examples:

```
(* transforms any assumption or goal of the form
    (x =? <anything>) = true    to    x = <anything>) *)
(* the actual value of x depends on the parameter passed to this tactic when used. *)
Ltac bool_to_prop x :=
  repeat (match goal with
    | [ H: Nat.eqb x _ = true |- _ ] => apply Nat.eqb_eq in H
    | [ |- Nat.eqb x _ = true] => apply Nat.eqb_eq
  end).

(* Helper tactic for traversing through a program and applying Hoare Logic rules *)
(* tries to apply all constructors of hoare_triple defined above in order. *)
Ltac ht1 := apply HtSkip || apply HtAssign || apply HtWrite || eapply HtSeq
            || eapply HtIf || eapply HtWhile || eapply HtAssert
            || eapply HtStrengthenPost.
```

5 Termination and Hoare Logic

The rules of Hoare logic as defined above only provide partial correctness: if the program terminates, the final state is guaranteed to satisfy the post condition. This, in addition to proving termination, is what constitutes total correctness.

All constructs in our programming language and rules in Hoare logic have no issue with termination, with the exception of while loops. The invariant by itself is not enough to guarantee termination, since an infinite loop may satisfy the invariant indefinitely (think of an empty while true loop).

The while rule of Hoare logic can be strengthened with a variant: traditionally, this is defined as a quantity that is guaranteed to decrease after each iteration, until it reaches some lower bound, and is guaranteed to never be lower than that bound. Usually, a variant is some natural number or arithmetic expression, that decreases with every iteration, and never goes below zero, for example, the different between the length of an array and the loop counter.

The while rule of Hoare logic must be re-written to require two additional premises, where S is the condition of the while loop:

1. For every number n: $\{\{I \wedge S = \mathit{true} \wedge \mathit{variant} = n\}\}$ c $\{\{I \wedge \mathit{variant} < n\}\}$:
   This will replace the inductive premise in our existing while rule. It requires that both the invariant is preserved, and the variant is decreased, after any execution of the body of the while loop.
2. $\{\{I \wedge S = \mathit{true} \wedge \rightarrow \mathit{variant} \geq 0\}\}$ Guarantee that whenever the invariant the condition of the loop are true, the variant must be greater than or equal to 0.

6 Recommended and Additional References:

Recommended references:

1. Frap Code for Semantics via Interpreters: https://github.com/achlipala/frap/blob/master/Interpreters.v
2. Frap Book chapter on Hoare logic: Chapter 12
   http://adam.chlipala.net/frap/frap_book.pdf

3. Frap Code for Hoare Logic:
   https://github.com/achlipala/frap/blob/master/HoareLogic.v

Additional references:

1. Frap Book chapter and code on Operational semantics: Chapter 7
   http://adam.chlipala.net/frap/frap_book.pdf
   https://github.com/achlipala/frap/blob/master/OperationalSemantics.v

2. Termination and total correctness of Hoare logic:
   https://www.cl.cam.ac.uk/archive/mjcg/HoareLogic/Lectures/Oct17.pdf

Bonus references for advanced extensions:

1. Small Step semantics for non-deterministic programs: Frap book, Section 7.3 and 7.4

2. Separation logic: Hoare logic extension for aliasing and dynamic memory allocation:
   Primer: http://www0.cs.ucl.ac.uk/staff/p.ohearn/papers/Marktoberdorf11LectureNotes.pdf
   Frap Book and code: Chapter 14.

3. Separation logic for concurrent shared-memory programs: Frap book and code Chapters 17 and 18.

4. Probabilistic Hoare logic: useful for extending to probabilistic and quantum languages
   Lectures from EasyCrypt school at UPenn:
   https://www.easycrypt.info/trac/raw-attachment/wiki/SchoolUPen2013/lecture3.pdf
   https://www.easycrypt.info/trac/raw-attachment/wiki/SchoolUPen2013/lecture6.pdf