1: Maps

Strings:

Before we can talk about maps we need to define strings and ids. These will the keys for our maps. We will also need to define comparison functions for strings so we can test their equality. This equality function is defined as follows.

```
Definition beq_string x y :=
    if string_dec x y then true else false.
```

Total Maps:

Now that we have keys defined for the maps, we can start talking about maps. The first of these are ***Total Maps***. Total Maps are defined as follows:

```
Definition total_map (A:Type) := string -> A.
```

Essentially it maps string to the type that was passed in upon creation. There are other definitions such as t_empty and t_update. T_empty creates an empty total map that has a default value. This means that if a key is not found then it will give back the default value. T_update adds another function to the map which maps a new key and value pair. A good way to think about maps is just a stack of functions. When looking for a key it will see if the first function holds, if not then it will go to the next one and so on an so forth until it reaches the end of the map, in which case the default value will just be returned.
When creating maps you can define them as follows:

```
Definition examplemap := t_update (t_update (t_empty false) "foo" true)
       "bar" true.
```

This is pretty straightforward and even getting values from a map is just as straightforward. Since maps are just stacks of functions then we pass the key that we want into the map and it will return the value from the map. For example:

```
Example update_example1 : examplemap "foo" = true.
    Proof. reflexivity. Qed.
```

Partial Maps:

Having partial maps is the main goal of this section. Partial maps are different from total maps in that a Partial Map with elements of type A corresponds to a total map with elements of type *option* A and a default value of *None*. We can define partial maps as follows:

```
Definition partial_map (A:Type) := total_map (option A).
```

We can then define our empty and update functions as follows:

```
Definition empty {A:Type} : partial_map A :=
        t_empty None.

Definition update {A:Type} (m : partial_map A) (x : string) (v : A) :=
        m & t_update x (Some v).
```

These are essentially the exact same as the total map's versions but they are now working with optional values of the type instead of the actual type.

2: Imp

Now, we will be discussing the proofs and structures in `Imp.v`. This program contains an interpreter for a language called Imp, that can handle basic arithmetic (both boolean and algebraic), and can also contain some basic commands.The language is simple indeed, but it is powerful enough to handle non-terminating programs and simple arithmetic. Along with that, it is also Turing complete. To begin we need to discuss the underlying structures:

Aexp:

To begin our discussion about `Imp.v`, we should first talk about the structure `aexp`, an inductive type defined in `Imp.v` as:

```
Inductive aexp :  Type :=
  | ANum :  nat -> aexp
  | APlus :  aexp -> aexp -> aexp
  | AMinus :  aexp -> aexp -> aexp
  | AMult :  aexp -> aexp -> aexp.
```

As you can see, the Inductive type has 4 possible base expressions:

1. `Anum:` This is an aexp -¿ aexp method that is essentially just meant to represent some constant.
2. `APlus, AMult,` and `AMinus:` A mapping of an aexp-¿aexp-¿aexp, meaning it will require two values of type aexp to function and will return an aexp (conisisting of a+b, a*b, or a-b, respectively)

So, to begin, what we basically have here is an inductive type that simply maps out what types are expected and the type that is returned. In order to define this we would need to add something like this:

```
Fixpoint aeval (a :  aexp) :  nat :=
 match a with
  | ANum n => n
  | APlus a1 a2 => (aeval a1) + (aeval a2)
  | AMinus a1 a2 => (aeval a1) - (aeval a2)
```

```
  | AMult a1 a2 => (aeval a1) * (aeval a2)
 end.
```

That function now maps an aexp type and evaluates it to it's natural number counterpart.

Bexp:

Next, we have boolean expressions that follow the same format as aexp, let's take a look:

```
Inductive bexp :  Type :=
  | BTrue :  bexp
  | BFalse :  bexp
  | BEq :  aexp -> aexp -> bexp
  | BLe :  aexp -> aexp -> bexp
  | BNot :  bexp -> bexp
  | BAnd :  bexp -> bexp -> bexp.
```

As you can see, this Inductive type has 6 possible base expressions:

1. `BTrue` and `BFalse`: These are the true and false values that will be the direct result of our boolean evaluations
2. `BEq` and `BLe`: A mapping of two values of type aexp that determine whether they are equal or not.
3. `BNot` and `BAnd`: These are bexp functions that accept one or two bexp values as input and returns a value of type bexp.

There is no need to show bexp, as it is essentially the same as aexp, simply containing the boolean functions (like `leb`) we had defined earlier in the class.

Reasoning on Bexp and Aexp:

Though it may not seem like much, we can actually do quite a bit of arithmetic with what we currently have in `Imp.v`. Though it may not be much, this is enough for us to have a very legitimate arithmetic language, and, we can make some pretty powerful assertions with this, and even begin to start to optimize operations and prove that every optimization are sound. For example:

```
Theorem optimize0plussound:  forall a,
   aeval (optimize0plus a) = aeval a.
Proof.
   intros a.  induction a.
    - (* ANum *) reflexivity.
    - (* APlus *) destruct a1.
     + (* a1 = ANum n *) destruct n.
       * (* n = 0 *) simpl.  apply IHa2.
```

```
      * (* n <> 0 *) simpl.  rewrite IHa2.  reflexivity.
    + (* a1 = APlus a11 a12 *)
     simpl.  simpl in IHa1.  rewrite IHa1.
     rewrite IHa2.  reflexivity.
    + (* a1 = AMinus a11 a12 *)
     simpl.  simpl in IHa1.  rewrite IHa1.
     rewrite IHa2.  reflexivity.
    + (* a1 = AMult a11 a12 *)
     simpl.  simpl in IHa1.  rewrite IHa1.
     rewrite IHa2.  reflexivity.
  - (* AMinus *)
   simpl.  rewrite IHa1.  rewrite IHa2.  reflexivity.
  - (* AMult *)
   simpl.  rewrite IHa1.  rewrite IHa2.  reflexivity.  Qed.
```

So, what does this proof do and how does it work? Well, this optimizes out any values of type aexp that are a (a natural number) and zero to just equate to the literal value of a. Thus after just defining arithmetic expressions and how we evaluate them, we can already optimize the evaluations we make themselves.

Note: We need to perform induction on these proofs because we cannot prove that the types of aexp (AMinus, APlus, etc.) will consistently behave in the same way. So we have to perform induction and prove it for all the types assigned to Aexp.

Let's see another example here:

```
Theorem aeval-iff-aevalR' : forall a n,
   (a \\ n) <-> aeval a = n.
Proof.
  split.
   - (* -> *)
    intros H; induction H; subst; reflexivity.
   - (* <- *)
    generalize dependent n.
    induction a; simpl; intros; subst; constructor;
     try apply IHa1; try apply IHa2; reflexivity.
Qed.
```

Here, we can see that Aevaln (when defined as 'a \\ b') can be proved through that same method as defined above, which would make sense, as these proofs utilize inference rules defined in the type, see our last section to read up on inference rules.

Constructed Logic

So, Coq runs off of constructed logic. In which we construct and form propositions and use those constructions to complete our goals. We have discussed propositions in full detail in our previous

2 continued on next page...

notes, yet they are crucial to understanding how reasoning in Coq is formed and created. So, in order to complete logic in Coq, all assertions and variables must be constructed, and those logical constructions exist in a list of hypotheses that we can use to assert our goal.

Variables and States

Now, we move onto to defining variables, in a very simple sense, variables are just keys for a map (which represents our memory) with strings (variable names) as keys pointing to some values. The map that holds those variables and their values is called the **state** and it is absolutely crucial for being able to reason on imperative programming languages.
Note: When using a map, you would have to define some default value for an uninitialized variable, the easiest way to fix this is by using a partial map, so that `None` will be the default value.

To begin, we need variable declaration (and assignment) to our existing aexp types.

```
Inductive aexp :  Type :=
  | ANum :  nat -> aexp
  | AId :  string -> aexp (* <----- NEW *)
  | APlus :  aexp -> aexp -> aexp
  | AMinus :  aexp -> aexp -> aexp
  | AMult :  aexp -> aexp -> aexp.
```

As you can see, we now have variable declaration as mapping of a string to an arithmetic expression, in `aeval`, it gets expanded to this:

```
AId x => st x
```

So what `AId` does is it retrieves the value (an expression in this case) that the key (x) represents. Since `AId` does not initialized or assign values to variables, we have to do that ourselves using `t_update` to update the values on our map, otherwise we will get our default value (see note regarding partial maps). In our version of `Imp.v`, it is defined as this:

```
(** Note, we are using a total map, so there is more than just this, you can see
all of them in Imp.v:  *)

Notation " a --> x ; b --> y " :=
 (t_update ( a --> x ) b y) (at level 0).
```

Since there is really nothing to prove in regards to variables and states that we haven't proved when working with maps, there are really no special proofs on states alone. However, comparing states is absolutely crucial in determining program equivalence and reasoning on robust programs.

2 continued on next page...

Commands

Now, we can move onto commands, these are statements that alter the functionality of our program. An easy example of a command is a loop. Here are the commands defined so far in Imp:

1. **While loops:** These are loops and are defined as `'WHILE' b 'DO' c 'END'`. This loop takes in a boolean expression which relates to something in the state, and it will then perform the command dictated by `c`. Infinite loops are allowed in Imp, and this poses an interesting problem, we will discuss more about it in the next section.
2. **SKIP:** `SKIP` is essentially Imp's version of a No Op command, this does nothing (literally, it's purpose is to do nothing).
3. **Variable Assignment:** `CAss` is our command that does variable assignment, as discussed in the previous section. It has the notation `x '::=' a`.
4. **Sequencing Commands:** Simply, `CSeq` is just a command that puts two commands into a sequence, the command looks like this: `c1 ;; c2`.
5. **If statements:** This, like loops, takes in a boolean expression and two commands, and executes one of the two commands based on the result of evaluating the given boolean expression. The command is written like so: `'IFB' c1 'THEN' c2 'ELSE' c3 'FI'`

Interesting Proofs on Imp (now with commands!)

Now, since we have commands, all those important things about states are coming into fruition. We now can create the function ceval, which is an inductive type, in Imp, it is defined as:

```
Inductive ceval : com -> state -> state -> Prop :=
  | E_Skip : forall st,
      SKIP / st \\ st
  | E_Ass  : forall st a1 n x,
      aeval st a1 = n ->
      (x ::= a1) / st \\ st & { x --> n }
  | E_Seq : forall c1 c2 st st' st'',
      c1 / st  \\ st' ->
      c2 / st' \\ st'' ->
      (c1 ;; c2) / st \\ st''
  | E_IfTrue : forall st st' b c1 c2,
      beval st b = true ->
      c1 / st \\ st' ->
      (IFB b THEN c1 ELSE c2 FI) / st \\ st'
  | E_IfFalse : forall st st' b c1 c2,
      beval st b = false ->
      c2 / st \\ st' ->
```

2 continued on next page...

```
      (IFB b THEN c1 ELSE c2 FI) / st \\ st'
  | E_WhileFalse : forall b st c,
      beval st b = false ->
      (WHILE b DO c END) / st \\ st
  | E_WhileTrue : forall st st' st'' b c,
      beval st b = true ->
      c / st \\ st' ->
      (WHILE b DO c END) / st' \\ st'' ->
      (WHILE b DO c END) / st \\ st''

  where "c1 '/' st '\\' st'" := (ceval c1 st st').
```

This inductive type, shows how we would evaluate all the commands passed into it. With that given to us, we are now able to expand an insanely long list of command/program evaluations. For example:

```
Z ::= 3;;
Y ::= 1;;
WHILE ! (Z = 0) DO
  Y ::= Y * Z;;
  Z ::= Z - 1;
END.
```

Something like that would equate out to this:

```
({True} \\ {Z = 3, Y = 1} \\ {True} \\ {Y = 3, Z = 2} \\ {True} \\ {Y = 6, Z = 1} \\
    {True} \\ {Y = 6, Z = 0} \\ {False}})
```

What this means is that we started out with a true condition (**Note:** We don't really care about our starting state), which then led to:

- A state in which Z = 3 and Y = 1, because we initialized those values to be so.
- A true state, which was derived from evaluating the loop condition (and it being true)
- Y = 3 and Z = 2 after we reassigned the values to them in the loop.
- Another true state because the loop condition it true
- ...
- Finally, we get the false state, that exists the loop and ends the program

So, as you can see, by the end of a list of commands, we will essentially be left with a sequence of states. This is how we can reason on them. Let's look at an example proof:

2 continued on next page...

```
Theorem ceval_deterministic: forall c st st1 st2,
     c / st \\ st1  ->
     c / st \\ st2 ->
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2.
  induction E1;
           intros st2 E2; inversion E2; subst.
  - (* E_Skip *) reflexivity.
  - (* E_Ass *) reflexivity.
  - (* E_Seq *)
    assert (st' = st'0) as EQ1.
    { (* Proof of assertion *) apply IHE1_1; assumption. }
    subst st'0.
    apply IHE1_2. assumption.
  - (* E_IfTrue, b1 evaluates to true *)
      apply IHE1. assumption.
  - (* E_IfTrue,  b1 evaluates to false (contradiction) *)
      rewrite H in H5. inversion H5.
  - (* E_IfFalse, b1 evaluates to true (contradiction) *)
    rewrite H in H5. inversion H5.
  - (* E_IfFalse, b1 evaluates to false *)
      apply IHE1. assumption.
  - (* E_WhileFalse, b1 evaluates to false *)
    reflexivity.
  - (* E_WhileFalse, b1 evaluates to true (contradiction) *)
    rewrite H in H2. inversion H2.
  - (* E_WhileTrue, b1 evaluates to false (contradiction) *)
    rewrite H in H4. inversion H4.
  - (* E_WhileTrue, b1 evaluates to true *)
      assert (st' = st'0) as EQ1.
      { (* Proof of assertion *) apply IHE1_1; assumption. }
      subst st'0.
      apply IHE1_2. assumption.  Qed.
```

We can see, that we can start to discuss program equivalence, by checking the states and comparing them. We can show that if two programs are equivalent when given the same starting states, then the final states must be equivalent, and again, due to the inductive nature of the command types, we need to use induction.

The Infinite Loop Conundrum

In regards to Imp, we know two things:

1. It is evaluated in Coq, and all Coq evaluations **MUST** terminate
2. Infinite loops are valid syntax in Imp.

So, we see the issue: we can write an infinite loop, which never terminates, but everything must terminate. So, we see the problem: but how do we go about fixing it? The obvious answer, is of course, to not run, or return a false state for any infinite loops. But, in order to do this, we need a way to prove that a loop is an infinite loop. Here would be a proof:

```
Theorem loop_never_stops : forall st st',
  ~(loop / st \\ st').
Proof.
  intros st st' contra. unfold loop in contra.
  remember (WHILE true DO SKIP END) as loopdef
           eqn:Heqloopdef.

  (** Proceed by induction on the assumed derivation showing that
      [loopdef] terminates.  Most of the cases are immediately
      contradictory (and so can be solved in one step with
      [inversion]). *)

  induction contra; try (inversion Heqloopdef).
  - rewrite -> H1 in H. inversion H.
    (* Where do these hpotheses come from, it basically comes from
        the fact that the while loop programs are inherently false.**)
  - apply IHcontra2. subst. reflexivity.
Qed.
```

So, as we can see here. By doing induction on the contradiction of our initial hypothesis, we can show that a loop will be non terminating.So a loop will simply return a false state if it is infinite.

3: Program Equivalence

Arithmetic and Boolean

Before we talk about Program Equivalence we need to cover both arithmetic and boolean equivalence. We have chosen to define these equivalences as follows:

```
Definition aequiv (a1 a2 : aexp) : Prop :=
    forall (st:state),
        aeval st a1 = aeval st a2.

Definition bequiv (b1 b2 : bexp) : Prop :=
    forall (st:state),
        beval st b1 = beval st b2.
```

Basically these functions work by setting up a proposition for the two expressions that were passed in and then for every state possible if the evaluation of the two expressions with the given state is are equal then we know that the two expressions are equal in it of themselves.

Commands

Command equivalence has to be a little more involved than this though. It cannot be just the given command and the previous state, we also need to consider the execution of the command and the resulting state, because we can potentially have two commands that are equal but lead to unequal ending states. As a result, we define command equivalence as follows:

```
    Definition cequiv (c1 c2 : com) : Prop :=
        forall (st st' : state),
                (c1 / st \\ st') <-> (c2 / st \\ st').
```

Some proofs to start out with to get a feel for how the equivalence relation works are ones involving the SKIP command, such as the following:

```
Theorem skip_left:  c,
  cequiv (SKIP;; c) c.
Proof.
  intros c st st'.
  split; intros H.
  - (* -> *)
    inversion H; subst.
    inversion H2. subst.
    assumption.
  - (* <- *)
    apply E_Seq with st.
    apply E_Skip.
    assumption.
Qed.
```

Then we can get into more interesting proofs which relate to boolean guards for the if and while statements. These will be more complicated because they will have commands, expressions and booleans all wrapped up together. We do not have to worry about this in general, instead we just have to worry about their equivalence to true or false. Some example proofs for these are as follows:

```
Theorem IFB_true: forall b c1 c2,
     bequiv b BTrue  ->
     cequiv
       (IFB b THEN c1 ELSE c2 FI)
       c1.
Proof.
  intros b c1 c2 Hb.
  split; intros H.
```

```
  - (* -> *)
    inversion H; subst.
    + (* b evaluates to true *)
      assumption.
    + (* b evaluates to false (contradiction) *)
      unfold bequiv in Hb. simpl in Hb.
      rewrite Hb in H5.
      inversion H5.
  - (* <- *)
    apply E_IfTrue; try assumption.
    unfold bequiv in Hb. simpl in Hb.
    rewrite Hb. reflexivity.
Qed.

Theorem WHILE_false : forall b c,
  bequiv b BFalse ->
  cequiv
    (WHILE b DO c END)
    SKIP.
Proof.
  intros b c Hb. split; intros H.
  - (* -> *)
    inversion H; subst.
    + (* E_WhileFalse *)
      apply E_Skip.
    + (* E_WhileTrue *)
      rewrite Hb in H2. inversion H2.
  - (* <- *)
    inversion H; subst.
    apply E_WhileFalse.
    rewrite Hb.
    reflexivity.
Qed.
```

These theorems are useful to us because when we go to prove hard theorems about if and while statements, we have ways to break it down based on what the boolean gaurd comes out to be. One of the most interesting things about this chapter is that we can prove things about non-terminating functions in coq when coq requires termination. This means that we will need to use inversion to cancel out a lot of the rules. Here is an example of proving a non-terminating while true loop.

```
Lemma WHILE_true_nonterm : forall b c st st',
  bequiv b BTrue ->
  ~( (WHILE b DO c END) / st \\ st' ).
Proof.
  intros b c st st' Hb.
```

```
  intros H.
  remember (WHILE b DO c END) as cw eqn:Heqcw.
  induction H;
  (* Most rules don't apply; we rule them out by inversion: *)
  inversion Heqcw; subst; clear Heqcw.
  (* The two interesting cases are the ones for WHILE loops: *)
  - (* E_WhileFalse *) (* contradictory -- b is always true! *)
    unfold bequiv in Hb.
    (* [rewrite] is able to instantiate the quantifier in [st] *)
    rewrite Hb in H. inversion H.
  - (* E_WhileTrue *) (* immediate from the IH *)
    apply IHceval2. reflexivity.
Qed.
```

Program Equivalence:

At this point in time we have arithmetic, boolean and command equivalence set up but now we want to prove that they are truly equivalences, meaning that they need to be reflexive, symmetric, and transitive. These proofs are all straightforward and simple to prove. The less obvious proof comes from the fact that these commands and programs still need to be equivalent even when embedded in larger programs. We can also show program in-equivalence. The main way for proving in-equivalence is to reach a contradiction, i.e. you assume that everything is the same and then at some point in your proof you should reach two things that are not equal. This is exactly what happened when we went over this example:

```
Fixpoint subst_aexp (i : string) (u : aexp) (a : aexp) : aexp :=
  match a with
  | ANum n       =>
      ANum n
  | AId i'       =>
      if eqb_string i i' then u else AId i'
  | APlus a1 a2  =>
      APlus (subst_aexp i u a1) (subst_aexp i u a2)
  | AMinus a1 a2 =>
      AMinus (subst_aexp i u a1) (subst_aexp i u a2)
  | AMult a1 a2  =>
      AMult (subst_aexp i u a1) (subst_aexp i u a2)
  end.

Definition subst_equiv_property := forall i1 i2 a1 a2,
  cequiv (i1 ::= a1;; i2 ::= a2)
         (i1 ::= a1;; i2 ::= subst_aexp i1 a1 a2).
```

This fixpoint and proof are trying to say that if I have two commands where they both assign values to X and Y, where Y has a term of X in it and the second command simply replaces the X

with the value of X, are they the same. If X is just constants then they are the same, however if X includes X in the definition then it starts to break down. We can derive states where Y can take on the value of X before or after X is altered. This would be a contradiction because these two states cannot be equal.

4: Miscellaneous Notes

New Tactics

1. **Specialize:** This tactic allows you to instantiate any terms in a global hypothesis. It works similar to generalize.
2. **Clear:** This tactic clears all or certain hypotheses, which can be very useful in complicated proofs for removing unnecessary hypotheses.
3. **Rename x into y:** This simply renames x as y in the current context, however, it does not change the name for the entire proof (unless the context is the entire proof).
4. **Remember:** This remembers some value, and stores it along with our hypotheses so we can use that in different contexts.
5. **eqn:(name) tag** In certain tactics, you are allowed to add the tag eqn to certain tactics, and it saves the result of that Tactic as a certain name given after the colon in that tag.
6. **Contradiction:** This is a tactic that tries to look for any false hypotheses in the given goal, and if there are any, it will complete the current goal. Just like calling inversion on a false hypothesis.
7. **Constructor:** This is mainly for when you are dealing with inductive logic in which you have constructed hypotheses. You can call constructor and it will call a piece of constructed logic to complete the current goal.
8. **Using semicolons with induction or destruct:** This is an interesting thing, but if you use semicolons after an induction or destruct statement, it will automatically perform those tactics after the semicolon on all of those cases. It's suggested you do this with the try tactic, so that there is no failure that will cause your proof to fail if there exists a case that cannot be solved using the method you provided. If said case does exist, after using this and try, it will leave behind the cases that were not solved for you to solve them manually.

Inference Rule Notation

Inference rules are a pretty important part of understanding programming languages, as it is the main method of reasoning and proving assertions on inductive types. To go over this, let's see a brief example, let's say we have an inductive type to define even numbers:

```
Inductive even: nat -> Prop :=
  | even_0: even 0
  | even_2n: forall n, even n -> even (S (S n)).
```

4 continued on next page...

This means we now have two base cases that we an use to reason work with to prove a number is even:

1. **even_0:** This says that 0 is even
2. **even_2n:** This states that if a number is even, then 2+n is also even.

The question the is, how do we use this to prove a number is even, lets use an example like 6:

```
        True
------------------even_0
        even 0
------------------even_2n
        even 2
------------------even_2n
        even 4
------------------even_2n
        even 6
```

So, what this does is show our result (or final assertion), and a rule we can use to validate that result. So as we can see here, we can work backward to show this:

```
(((even 0) -> even 2) -> even 4) -> even 6
```

We are essentially setting up implications and working backwards, and if we can prove that the previous implications are true, then we can prove the final goal. So in the case of 6,we need to show that 4 is even to prove that 6 is even, according to the even_2n rule, repeat for 4 and prove that 2 is even, then repeat for 2 and show that 0 is even, and use even_0 to prove that 0 is even. Then using that, we can prove 2 is even, and 4 is even, and then finally, 6 is even. So essentially, inference rules are ways of using inductive reasoning to prove a final goal. As for using these hypotheses in Coq, these inference rules function as normal lemmas or theorems, meaning you can call these rules at any time to prove a goal using **apply** or **rewrite**.

Purpose of Equivalence and Reasoning on languages (Imp and Equiv):

So, you may be wondering, what is the purpose of all this? The main purpose of this is for program verification. An interesting paper denotes that proving program equivalence can verify programs that have gone through or will go through a transformation in some way, shape or form. The example discussed in this paper calls for using dynamic single alignment. It is an interesting read regarding the future uses and functionality behind program verification and equivalence. The methods discussed in the paper discuss equivalence and verification in regards to state transformation, and even a bit of memory checking. [1]

4 continued on next page...

References

[1] K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens, "Verification of source code transformations by program equivalence checking," 2005.