Instructor: Matt Fredrikson TA: Ryan Wagner

Induction and Semantics in Dafny

Encoding the syntax of Imp

Recall the abstract syntax of Imp:

$$\begin{array}{rcl} a \in \textbf{AExp} & ::= & n \in \mathbb{Z} \mid x \in \mathsf{Var} \mid a_1 + a_2 \\ b \in \textbf{BExp} & ::= & \mathsf{true} \mid \mathsf{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\ c \in \textbf{Com} & ::= & \textbf{skip} \mid x := a \mid c_1 ; c_2 \\ & & \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \\ & & \mid \textbf{while } b \textbf{ do } c \end{array}$$

We can encode this in Dafny using *inductive datatypes*. Intuitively, values of inductive datatypes can be seen as finite trees, where nodes correspond to *constructors* with zero or more arguments. Constructors can take parameters, and when the type of a parameter is the inductive datatype itself, the corresponding node is internal.

We begin by declaring types for variables and values, which we will represent using strings and integers, respectively. The remaining definition defines inductive datatypes for AExp, BExp, and Com.

```
type Var = string
type Val = int

datatype AExp = N(Val)
              | V(Var)
              | Plus(AExp, AExp)
datatype BExp = B(bool)
              | Less(AExp, AExp)
              | Eq(AExp, AExp)
              | Not(BExp)
              | And(BExp)
datatype Com  = Skip
              | Assign(Var, AExp)
              | Seq(Com, Com)
              | If(BExp, Com, Com)
              | While(BExp, Com)
```

To see how we might use these types, consider the expression $(5 + x) < y$:

```
var b := Less(Plus(N(5), V("x")), V("y"));
```

To encode a simple program:

$$\textbf{while } (5 + x) < y \textbf{ do } x := x + 1$$

We would construct the type:

```
var b := Less(Plus(N(5), V("x")), V("y"));
var c := While(b, Assign("x", Plus(V("x"), N(1))));
```

Inference Rules

Now let's implement the big-step semantics. Recall that our semantics defined an environment as a mapping from variables to their values, and a configuration as a pair consisting of a command and an environment. Additionally, we discussed derivations, which we can model as a finite sequence of configurations.

```
type Env = map<Var, Val>
type Config = (Com, Env)
type Derivation = seq<Config>
```

Let's start implementing the inference rules themselves by covering the rules for the expressions.

$$\textsc{Const}\ \frac{}{\langle n, \sigma\rangle \Downarrow_a n} \qquad \textsc{Var}\ \frac{}{\langle x, \sigma\rangle \Downarrow_a \sigma(x)} \qquad \textsc{Add}\ \frac{\langle a_1, \sigma\rangle \Downarrow_a n_1 \qquad \langle a_2, \sigma\rangle \Downarrow_a n_2}{\langle a_1 + a_2, \sigma\rangle \Downarrow_a n_1 +_{\mathbb{Z}} n_2}$$

We know that evaluating expressions always terminates, so we can model these rules with a single function. We'll use a `match` expression to split into rules based on the abstract syntax of the expression passed in.

```
function bigstep_aexp(a: AExp, s: Env): Val
{
  match(a) {
    case N(n) => n
    case V(x) => if x in s then s[x] else 0
    case Plus(a0, a1) => bigstep_aexp(a0, s) + bigstep_aexp(a1, s)
  }
}
```

The only aspect of this function that may be unexpected is the rule for variable evaluation. Previously, we've assumed that the environment will have a mapping for every possible variable, but we made no assumptions about what the mapping could be. Here we first check to see that the referenced variable is in the environment, and if it isn't, the function returns zero. This introduces an additional assumption that we haven't made before, which is that all variables start off initialized to zero. Alternatively, we could have written the function with a precondition that requires any environment passed in to contain all mappings for all variables mentioned in the expression:

```
function bigstep_aexp(a: AExp, s: Env): Val
  requires forall v: Var :: v in free_vars(a) ==> v in s
{
  match(a) {
    case N(n) => n
    case V(x) => if x in s then s[x] else 0
    case Plus(a0, a1) => bigstep_aexp(a0, s) + bigstep_aexp(a1, s)
  }
}
```

We would then need to define a function `free_vars`, and whenever we called `bigstep_aexp`, we'd need to make sure that this precondition were met. To simplify things today, we'll use the former approach, and assume everything starts out initialized to zero.

Moving on to the rules for BExp, we have the following rules.

$$\textsc{True}\ \frac{}{\langle \mathsf{true}, \sigma\rangle \Downarrow_b \mathit{true}} \qquad \textsc{False}\ \frac{}{\langle \mathsf{false}, \sigma\rangle \Downarrow_b \mathit{false}} \qquad \textsc{Eq}\ \frac{\langle a_1, \sigma\rangle \Downarrow_a n_1 \qquad \langle a_2, \sigma\rangle \Downarrow_a n_2}{\langle a_1 = a_2, \sigma\rangle \Downarrow_a n_1 =_{\mathbb{Z}} n_2}$$

$$\textsc{Less}\ \frac{\langle a_1, \sigma\rangle \Downarrow_a n_1 \qquad \langle a_2, \sigma\rangle \Downarrow_a n_2}{\langle a_1 < a_2, \sigma\rangle \Downarrow_a n_1 <_{\mathbb{Z}} n_2} \qquad \textsc{NotTrue}\ \frac{\langle b, \sigma\rangle \Downarrow_b \mathit{true}}{\langle \neg b, \sigma\rangle \Downarrow_b \mathit{false}} \qquad \textsc{NotFalse}\ \frac{\langle b, \sigma\rangle \Downarrow_b \mathit{false}}{\langle \neg b, \sigma\rangle \Downarrow_b \mathit{true}}$$

$$\textsc{And}\ \frac{\langle b_1, \sigma\rangle \Downarrow_b v_1 \qquad \langle b_2, \sigma\rangle \Downarrow_b v_2}{\langle b_1 \wedge b_2, \sigma\rangle \Downarrow_b v_1 \wedge v_2}$$

Using the same reasoning we did for the arithmetic expressions, we arrive at the following function. There are no surprises here, we use the same approach as before.

```
function bigstep_bexp(b: BExp, s: Env): bool
{
  match(b) {
    case B(v) => v
    case Less(a0, a1) => bigstep_aexp(a0, s) < bigstep_aexp(a1, s)
    case Eq(a0, a1) => bigstep_aexp(a0, s) == bigstep_aexp(a1, s)
    case Not(op) => !bigstep_bexp(op, s)
    case And(b0, b1) => bigstep_bexp(b0, s) && bigstep_bexp(b1, s)
  }
}
```

Recall the rules for big-step evaluation of commands:

$$\text{ASGN}\ \frac{\langle a, \sigma\rangle \Downarrow_a n}{\langle x := a, \sigma\rangle \Downarrow \sigma[x \mapsto n]} \qquad \text{SKIP}\ \frac{}{\langle \textbf{skip}, \sigma\rangle \Downarrow \sigma} \qquad \text{SEQ}\ \frac{\langle c_1, \sigma_1\rangle \Downarrow \sigma_1' \qquad \langle c_2, \sigma_1'\rangle \Downarrow \sigma_2}{\langle c_1; c_2, \sigma_1\rangle \Downarrow \sigma_2}$$

$$\text{IFTRUE}\ \frac{\langle b, \sigma\rangle \Downarrow_b \textit{true} \qquad \langle c_1, \sigma\rangle \Downarrow \sigma_2}{\langle \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2, \sigma\rangle \Downarrow \sigma_2} \qquad \text{IFFALSE}\ \frac{\langle b, \sigma\rangle \Downarrow_b \textit{false} \qquad \langle c_2, \sigma\rangle \Downarrow \sigma_2}{\langle \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2, \sigma\rangle \Downarrow \sigma_2}$$

$$\text{WHILEFALSE}\ \frac{\langle b, \sigma\rangle \Downarrow_b \textit{false}}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma\rangle \Downarrow \sigma}$$

$$\text{WHILETRUE}\ \frac{\langle b, \sigma\rangle \Downarrow_b \textit{true} \qquad \langle c, \sigma\rangle \Downarrow \sigma' \qquad \langle \textbf{while}\ b\ \textbf{do}\ c, \sigma'\rangle \Downarrow \sigma''}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma\rangle \Downarrow \sigma''}$$

If we were to implement these in the same way that we did for expressions, we would run into trouble with termination. Unlike expressions, commands need not terminate, so the function that we wrote would not be defined on certain inputs. Instead, we'll define the semantics for commands using a predicate, which you'll recall is just a Boolean-valued function. The predicate will take as arguments a command `c`, environments `s` and `s'`, and a natural number `k`. We want the predicate to return *true* whenever evaluating `c` starting in `s` reduces to `s'` with a derivation containing at most `k` steps.

```
predicate bigstep(c: Com, s: Env, s': Env, k: nat)
  decreases c,k
{
  match(c) {
    case Skip =>
      k == 1 && s == s'
    case Assign(v, a) =>
      k == 1 && s[v := bigstep_aexp(a, s)] == s'
    case Seq(c1, c2) =>
      exists k': nat, s'': Env ::
        0 < k' < k &&
        bigstep(c1, s, s'', k') &&
        bigstep(c2, s'', s', k-k')
    case If(b, c1, c2) =>
      if(bigstep_bexp(b, s)) then
        bigstep(c1, s, s', k)
      else
        bigstep(c2, s, s', k)
    case While(b, c') =>
      if(bigstep_bexp(b, s)) then
        exists s'': Env, k': nat :: 0 < k' < k &&
          bigstep(c', s, s'', k') &&
          bigstep(c, s'', s', k-k')
      else
        k == 1 && s == s'
  }
}
```

Unlike our functions expression semantics, this function does not have a case for each rule. Instead, it has a case for each syntactic form of Com, and composes recursive calls in a way that faithfully accounts for each rule. Notice that our termination metric refers to two variables, `c` and `k`. This is necessary because some cases only reduce the size of the expression (e.g., `if`), and some only reduce the size of the derivation (e.g., `while`) when they recursively call the semantics.

Using the semantics

Proving facts about specific programs

Now that we have an implementation of the semantics, we can get started applying them to reason about Imp programs. First we'll take a look at proving things about specific programs given as `Com` types.

Suppose that we're given the program:

$$\textbf{while } 0 < x \textbf{ do } x := x - 1$$

and we'd like to show that staring out in an environment $\sigma = [x \mapsto 2]$ leads to a final environment $\sigma' = [x \mapsto 0]$. Although this might be obvious to us by simple inspection of the program, to prove that this is the case, we'll need to make use of the semantics. Formally, we'd like to show that,

$$\langle \textbf{while } 0 < x \textbf{ do } x := x - 1, [x \mapsto 2]\rangle \Downarrow [x \mapsto 0]$$

Before seeing how to do this using Dafny, let's review how to do it "on paper" first. Recall that we can prove specific derivations by finding a proof tree that applies the relevant inference rules for our judgement. In this case, we know that the last rule applied in such a tree will be WhileTrue, because the command is a while loop and in the environment $\sigma = [x \mapsto 2]$, $\langle 0 < x, \sigma\rangle \Downarrow_b \mathit{true}$. To the root of the proof tree will look as follows:

$$\textsc{WhileTrue}\ \frac{T_1 \qquad T_2 \qquad T_3}{\langle \textbf{while } 0 < x \textbf{ do } x := x - 1, [x \mapsto 2]\rangle \Downarrow [x \mapsto 0]}$$

By inspecting the rule WhileTrue, we know that T_1 will need to end with the conclusion $\langle 0 < x, [x \mapsto 2]\rangle \Downarrow_b \mathit{true}$, T_2 will need to end with the conclusion $\langle x := x - 1, [x \mapsto 2]\rangle \Downarrow \sigma'$, and T_3 will need to end with $\langle \textbf{while } 0 < x \textbf{ do } x := x - 1, \sigma'\rangle \Downarrow [x \mapsto 0]$. To finish the proof, we need to construct T_1, T_2, T_3.

We now need to derive the tree T_1, which is an inequality comparison. We see that:

$$\textsc{Less}\ \frac{\textsc{Const}\ \dfrac{}{\langle 0, [x \mapsto 2]\rangle \Downarrow_a 0} \qquad \textsc{Var}\ \dfrac{}{\langle x, [x \mapsto 2]\rangle \Downarrow_a 2}}{\langle 0 < x, [x \mapsto 2]\rangle \Downarrow_b \mathit{true}}$$

Moving on to T_2, which is an assignment, we need to apply Asgn. By inspecting the command and inital environment, we know that σ', the final environment, must be equal to $[x \mapsto 1]$. We justify this as follows:

$$\textsc{Asgn}\ \frac{\textsc{Plus}\ \dfrac{\textsc{Var}\ \dfrac{}{\langle x, [x \mapsto 2]\rangle \Downarrow_a 2} \qquad \textsc{Const}\ \dfrac{}{\langle -1, [x \mapsto 2]\rangle \Downarrow_a -1}}{\langle x - 1, [x \mapsto 2]\rangle \Downarrow_a 1}}{\langle x := x - 1, [x \mapsto 2]\rangle \Downarrow [x \mapsto 1]}$$

Moving on to T_3, we now know that $\sigma' = [x \mapsto 1]$. Again, we're looking at a loop, and we know that $\langle 0 < x, [x \mapsto 1]\rangle \Downarrow_b \textit{true}$, so T_3 will be rooted by WhileTrue.

$$\textsc{WhileTrue}\ \frac{T_4 \qquad T_5 \qquad T_6}{\langle \textbf{while } 0 < x \textbf{ do } x := x - 1, [x \mapsto 1]\rangle \Downarrow [x \mapsto 0]}$$

We're going to re-use much of our work from T_1, T_2, and T_3, keeping in mind that the initial environment has changed to $[x \mapsto 1]$. Starting with T_4:

$$\textsc{Less}\ \frac{\textsc{Const}\ \dfrac{}{\langle 0, [x \mapsto 1]\rangle \Downarrow_a 0} \qquad \textsc{Var}\ \dfrac{}{\langle x, [x \mapsto 1]\rangle \Downarrow_a 1}}{\langle 0 < x, [x \mapsto 1]\rangle \Downarrow_b \textit{true}}$$

Continuing with T_5, we find that the next environment is $[x \mapsto 0]$:

$$\textsc{Asgn}\ \frac{\textsc{Plus}\ \dfrac{\textsc{Var}\ \dfrac{}{\langle x, [x \mapsto 1]\rangle \Downarrow_a 1} \qquad \textsc{Const}\ \dfrac{}{\langle -1, [x \mapsto 1]\rangle \Downarrow_a -1}}{\langle x - 1, [x \mapsto 1]\rangle \Downarrow_a 0}}{\langle x := x - 1, [x \mapsto 1]\rangle \Downarrow [x \mapsto 0]}$$

Now that we get to T_6, we have to re-consider the **while** loop. We won't apply WhileTrue this time, because the initial environment maps x to 0. Now we can apply WhileFalse:

$$\textsc{Asgn}\ \frac{\textsc{Less}\ \dfrac{\textsc{Const}\ \dfrac{}{\langle 0, [x \mapsto 0]\rangle \Downarrow_a 0} \qquad \textsc{Var}\ \dfrac{}{\langle 0, [x \mapsto 0]\rangle \Downarrow_a 0}}{\langle 0 < x, [x \mapsto 0]\rangle \Downarrow_b \textit{false}}}{\langle \textbf{while } 0 < x \textbf{ do } x := x - 1, [x \mapsto 0]\rangle \Downarrow [x \mapsto 0]}$$

This completes the proof, because there are no further assumptions that need to be discharged.

Now we'll see how to do this in Dafny. We prove things about functions and predicates in Dafny using lemmas. You can think of lemmas as methods that are not meant to be compiled, so their utility draws from the specifications that we give them. Dafny lemmas can be later invoked to prove to the verifier that a particular property holds for some objects. For the present example, recall that we want to show that,

$$\langle \textbf{while } 0 < x \textbf{ do } x := x - 1, [x \mapsto 2]\rangle \Downarrow [x \mapsto 0]$$

In our implementation, `bigstep` represents the $\Downarrow$ relation, and we represent environments as maps, so $[x \mapsto 2]$ is represented by `map["x" := 2]` and $[x \mapsto 0]$ by `map["x" := 0]`. The program that we're reasoning about is a value of type `Com`, which we can write as:

```
While(Less(N(0), V("x")), Assign("x", Plus(V("x"), N(-1))))
```

Additionally, from our earlier proof, we know that this program evaluates on its initial environment in three steps (including the "exit" from the loop). Putting this together, our lemma will start out with:

```
lemma loop_to_zero()
  ensures bigstep(While(Less(N(0), V("x")), Assign("x", Plus(V("x"), N(-1)))),
                  map["x" := 2],
                  map["x" := 0],
                  3)
```

Notice that we use the postcondition of the lemma, specified using the `ensures` syntax, to state the property that we're trying to prove. If we had any assumptions or premises to work with, i.e., if we were proving a statement of the form $F \to G$, then we could have used a `requires` annotation to specify this in our lemma.

We'll now write the proof of this property in the body of the lemma, much as we'd write a method. But we'll start out by defining some variables as shorthand for the main components of our property:

```
  var c := While(Less(N(0), V("x")), Assign("x", Plus(V("x"), N(-1))));
  var b := Less(N(0), V("x"));
  var body := Assign("x", Plus(V("x"), N(-1)));
  var a :=  Plus(V("x"), N(-1));
  var s := map["x" := 2];
  var s' := map["x" := 0];
```

We're now ready to begin proving the lemma. To prove this property, we need to show that the rules, which we encoded in the predicate `bigstep`, give us a way to reach our desired conclusion. Our lemma has no `requires` annotations, so we don't have any premises to start out with other than *true*. This means we we need to show that:

$$\mathit{true} \to (\langle \textbf{while } 0 < x \textbf{ do } x := x - 1, [x \mapsto 2]\rangle \Downarrow [x \mapsto 0])$$

To do this, we'll use a *calculational proof*. You've seen these before, but probably not named as such: starting from an initial premise (in our case, *true*), we'll provide a chain of implications that leads to the desired conclusion. You've probably written proofs of the form:

$$\begin{aligned} P &\to F_1 \\ &\vdots \\ &\to Q \end{aligned}$$

Dafny has a special syntax for calculational proofs, which looks like the following:

```
  calc ==> {
    P;
      { justification }
    F1;
      { justification }
    ...
      { justification }
    Q;
  }
```

The symbol `==>` tells Dafny that we're going to show a sequence of right-implications. The justifications are optional when Dafny is able to prove that each step follows from its predecessor, but if this is not the case, then you may need to fill these in with appropriate `assert` statements or call existing lemmas to help Dafny prove your claim.

Calculational proofs in Dafny are somewhat generic, so we could have also used the symbol `<==`, which would reverse the chain using left-implications to work from the conclusion back to the premise:

```
calc <== {
  Q
    { justification }
  Fn
    { justification }
  ...
    { justification }
  P
}
```

When we worked out the proof for the current example by hand, we actually proceeded backwards from the conclusion to applications of axioms. We'll follow that example when we work this in Dafny, making use of `calc <==` to do so.

Starting with the conclusion `bigstep(c, s, s', 3)`, which corresponds to the root of our proof tree from before, we know that $\langle x, [x \mapsto 2]\rangle \Downarrow 2$, so the head of the loop (`0 < x`) evaluates to *true*. We also know that to reach this conclusion, we'll need to look at the part of the semantics that deals with `while` loops:

```
  case While(b, c') =>
    if(bigstep_bexp(b, s)) then
      exists s'': Env, k': nat :: 0 < k' < k &&
        bigstep(c', s, s'', k') &&
        bigstep(c, s'', s', k-k')
    else
      k == 1 && s == s'
```

In particular, the branch of the `if` expression corresponding to the positive evaluation of the guard `b`. This says that to show big-step evaluation from `s` to `s'` in 3 steps, we need to show the existence of an intermediate state `s''` and natural number `k'` that takes the body from `s` to `s''` in `k'` steps, and then takes the rest of the loop from `s''` to `s'` in `k-k'` steps. Because the body subtracts 1 from `x`, and `x` takes the value 2 in `s`, we know that the intermediate state must be `s["x" := 1]`. Furthermore, the rule for assignment consumes 1 step, so we know that `k'` must be 1. This lets us write:

```
calc <== {
  bigstep(c, s, s', 3);
    // def. of while
  bigstep_bexp(b, s) &&
    bigstep(body, s, s["x" := 1], 1) &&
    bigstep(c, s["x" := 1], s', 2);
}
```

Although Dafny doesn't complain about any of the steps in our calculation, we're not done yet, as Dafny still doesn't agree that the lemma's postcondition is satisfied. This means that we need to justify the last line of our calculation. In particular, we need to show that evaluating the loop body starting in `s` leads to a new state where `x` takes the value 1, and why it's safe to assume that executing `c` in that new state leads to `s'` in 2 steps. We get the former from the semantics of the plus operator (which, recall, is implemented in `bigstep_aexp`), and the latter we'll need to justify in subsequent steps:

```
calc <== {
  bigstep(c, s, s', 3);
    // def. of while
  bigstep_bexp(b, s) &&
    bigstep(body, s, s["x" := 1], 1) &&
    bigstep(c, s["x" := 1], s', 2);
  // def. of plus, def. of while
  bigstep_aexp(a, s) == 1 && bigstep(c, s["x" := 1], s', 2);
}
```

Again, we need to prove the last line, as Dafny cannot do it on its own. We'll begin with the second conjunct, as it is the more difficult of the two. Using reasoning similar to what we did for the second line, we can write:

```
calc <== {
  bigstep(c, s, s', 3);
    // def. of while
  bigstep_bexp(b, s) &&
    bigstep(body, s, s["x" := 1], 1) &&
    bigstep(c, s["x" := 1], s', 2);
    // def. of plus, def. of while
  bigstep_aexp(a, s) == 1 && bigstep(c, s["x" := 1], s', 2);
    // def. of while
  bigstep_bexp(b, s["x" := 1]) &&
    bigstep(body, s["x" := 1], s', 1) &&
    bigstep(c, s', s', 1);
}
```

Again, we used the *true* case of the definition of semantics for **while** to derive this step. We know that executing the body $x := x - 1$ in the environment $[x \mapsto 1]$ leads to $[x \mapsto 0]$, which is the environment `s'` mentioned in our goal. Notice that Dafny accepts this line in the calculational proof without asking for justification of the conjunct `bigstep_aexp(a, s) == 1`. This means that we don't need to prove it—Dafny can do so on its own, in contrast to our by-hand proof, where completeness required a worked-out justification.

Continuing just as before, Dafny isn't able to prove that evaluating the body results in the post-environment `s'`. Previously, we dealt with this by telling Dafny that `bigstep_aexp(a, s) == 1`. In this case, we point out that `bigstep_aexp(a, s["x" := 1]) == 0`:

```
calc <== {
  bigstep(c, s, s', 3);
    // def. of while
  bigstep_bexp(b, s) &&
    bigstep(body, s, s["x" := 1], 1) &&
    bigstep(c, s["x" := 1], s', 2);
    { assert bigstep_aexp(a, s) == 1; }
    // def. of plus, def. of while
  bigstep_aexp(a, s) == 1 && bigstep(c, s["x" := 1], s', 2);
    // def. of while
  bigstep_bexp(b, s["x" := 1]) &&
    bigstep(body, s["x" := 1], s', 1) &&
    bigstep(c, s', s', 1);
    // def. of assign, seq
  bigstep_aexp(a, s["x" := 1]) == 0;
}
```

At this point, it seems like we've given Dafny everything it needs to know, but it still thinks we're not done, as it tells us that the lemma postcondition might not hold. Let $c = \textbf{while } 0 < x \textbf{ do } x := x - 1$. Looking at the body of the lemma, we've just shown that:

$$\begin{aligned}
&\langle x - 1, [x \mapsto 1]\rangle \Downarrow_a 0 \\
&\rightarrow \langle 0 < x, [x \mapsto 1]\rangle \Downarrow_b \mathit{true} \wedge \langle x := x - 1, [x \mapsto 1]\rangle \Downarrow [x \mapsto 0] \wedge \langle c, [x \mapsto 0]\rangle \Downarrow_a [x \mapsto 0] \\
&\rightarrow \langle c, [x \mapsto 1]\rangle \Downarrow [x \mapsto 0] \\
&\rightarrow \langle 0 < x, [x \mapsto 2]\rangle \Downarrow_b \mathit{true} \wedge \langle x := x - 1, [x \mapsto 2]\rangle \Downarrow [x \mapsto 1] \wedge \langle c, [x \mapsto 1]\rangle \Downarrow_a [x \mapsto 0] \\
&\rightarrow \langle c, [x \mapsto 2]\rangle \Downarrow [x \mapsto 0]
\end{aligned}$$

The conclusion of this calculation corresponds to the postcondition of our lemma. However, Dafny can only use it to conclude that the postcondition holds if it believes that the premise is true. We can point this out by adding a corresponding assertion after the calculation:

```
calc <== {
  bigstep(c, s, s', 3);
    // def. of while
  bigstep_bexp(b, s) &&
    bigstep(body, s, s["x" := 1], 1) &&
    bigstep(c, s["x" := 1], s', 2);
    { assert bigstep_aexp(a, s) == 1; }
    // def. of plus, def. of while
  bigstep_aexp(a, s) == 1 && bigstep(c, s["x" := 1], s', 2);
    // def. of while
  bigstep_bexp(b, s["x" := 1]) &&
    bigstep(body, s["x" := 1], s', 1) &&
    bigstep(c, s', s', 1);
    // def. of assign, seq
  bigstep_aexp(a, s["x" := 1]) == 0;
}
assert bigstep_aexp(a, s["x" := 1]) == 0;
```

Alternatively, we could add another line to the calculation that Dafny will certainly believe to be true, i.e., `true`. This should force it to attempt a proof that the first "real" line of the calculation is valid. Putting all of this together, the full lemma becomes:

```
lemma loop_to_zero()
  ensures bigstep(While(Less(N(0), V("x")), Assign("x", Plus(V("x"), N(-1)))),
                  map["x" := 2],
                  map["x" := 0],
                  3)
{
  var c := While(Less(N(0), V("x")), Assign("x", Plus(V("x"), N(-1))));
  var b := Less(N(0), V("x"));
  var body := Assign("x", Plus(V("x"), N(-1)));
  var a :=  Plus(V("x"), N(-1));
  var s := map["x" := 2];
  var s' := map["x" := 0];

  calc <== {
    bigstep(c, s, s', 3);
      // def. of while
    bigstep_bexp(b, s) &&
      bigstep(body, s, s["x" := 1], 1) &&
      bigstep(c, s["x" := 1], s', 2);
      { assert bigstep_aexp(a, s) == 1; }
      // def. of plus, def. of while
    bigstep_aexp(a, s) == 1 && bigstep(c, s["x" := 1], s', 2);
      // def. of while
    bigstep_bexp(b, s["x" := 1]) &&
      bigstep(body, s["x" := 1], s', 1) &&
      bigstep(c, s', s', 1);
      // def. of assign, seq
    bigstep_aexp(a, s["x" := 1]) == 0;
    true;
  }
}
```

Proving semantic properties

Now let's return to the fact that the big-step semantics for Imp are deterministic, which we discussed in lecture. Recall that we proved the property:

$$\forall \sigma, \sigma_1, \sigma_2, c.(\langle c, \sigma\rangle \Downarrow \sigma_1 \wedge \langle c, \sigma\rangle \Downarrow \sigma_2) \rightarrow \sigma_1 = \sigma_2$$

In other words, for a given command, if we begin evaluating in any environment, then the environment we end with (if any) is unique. We proved this by induction on derivations, because induction on syntax is not well-founded. To prove this using our Dafny semantics, we'll begin by writing the lemma:

```
lemma bigstep_determ(c: Com, s0: Env, s1: Env, s1': Env, k1: nat, k2: nat)
  requires bigstep(c, s0, s1, k1)
  requires bigstep(c, s0, s1', k2)
  ensures s1 == s1'
  ensures k1 == k2
```

There are several differences between this lemma from the previous one. First of all, `bigstep_determ` takes arguments. In particular, a subset of its arguments correspond to the universally-quantified variables $\sigma, \sigma_1, \sigma_2$, and c from the statement we're trying to prove. When lemmas take arguments that are mentioned in their postconditions, they have to prove that the postcondition holds for whatever values are passed in as long as they satisfy the lemma's preconditions, which is like making a universal statement. Second, `bigstep_determ` has two `requires` annotations, which together correspond to the antecedent of our proof goal.

The last two arguments to `bigstep_determ` correspond to the length of the derivations that we assume to exist. As we will see, including these as arguments gives us a well-founded induction later on. Alternatively, we could have defined a predicate:

```
predicate bigstep_deriv(c: Com, d: Derivation)
```

We could define `bigstep_deriv` to return *true* whenever `d` is a valid derivation of `c` under the big-step semantics, i.e.,

```
d[0].0 == c &&
  forall i :: 0 <= i < |d|-1 ==> bigstep(d[i].0, d[i].1, d[i+1].1, |d|-i)
```

We might then have written the lemma:

```
lemma bigstep_determ(c: Com, d: Derivation, d': Derivation)
  requires 0 < |d| && 0 < |d'|
  requires c == d[0].0 && c == d'[0].0
  requires bigstep_deriv(c, d)
  requires bigstep_deriv(c, d')
  ensures d[|d|-1].1 == d'[|d'|-1].1
  ensures |d| == |d'|
```

However, to avoid introducing new predicates and functions, we'll continue with the proof using our existing definitions.

We'll proceed by induction, using a match on `c` to guide our coverage of the possible derivations. Starting with the base case corresponding to **skip**:

```
  match(c) {
    case Skip =>
      assert s0 == s1 == s1';
```

Moving to assignment, we add the case:

```
    case Assign(v, a) =>
      assert s0[v := bigstep_aexp(a, s0)] == s1 == s1';
```

Notice that Dafny accepts this assertion. In lecture, we needed to prove that evaluation of expressions is deterministic. Dafny can deduce this automatically, which you can check by writing the lemma that verifies without assistance:

```
lemma aexp_determ(a: AExp, s: Env, n1: Val, n2: Val)
  requires bigstep_aexp(a, s) == n1
  requires bigstep_aexp(a, s) == n2
  ensures n1 == n2
{}
```

The case for `Seq` is a bit more involved. Because we can assume that $\langle c_1; c_2, \sigma_0\rangle \Downarrow \sigma_1$, there must be a derivation that looks like:

$$\textsc{Seq}\ \frac{\dfrac{T_1}{\langle c_1, \sigma_0\rangle \Downarrow \sigma_0'} \qquad \dfrac{T_2}{\langle c_2, \sigma_0'\rangle \Downarrow \sigma_1}}{\langle c_1; c_2, \sigma_0\rangle \Downarrow \sigma_1}$$

Similarly for the second evaluation:

$$\textsc{Seq}\ \frac{\dfrac{T_3}{\langle c_1, \sigma_0\rangle \Downarrow \sigma_0''} \qquad \dfrac{T_4}{\langle c_2, \sigma_0''\rangle \Downarrow \sigma_1'}}{\langle c_1; c_2, \sigma_0\rangle \Downarrow \sigma_1'}$$

We can apply the inductive hypothesis on T_1, T_2, T_3, and T_4 to conclude that $\sigma_0' = \sigma_0''$ and thus $\sigma_1 = \sigma_1'$. To write this in Dafny, we say the following:

```
    case Seq(c1, c2) =>
      var s0', k1' :| 0 < k1' < k1 &&
        bigstep(c1, s0, s0', k1') &&
        bigstep(c2, s0', s1, k1-k1');
      var s0'', k2' :| 0 < k2' < k2 &&
        bigstep(c1, s0, s0'', k2') &&
        bigstep(c2, s0'', s1', k2-k2');
      bigstep_determ(c1, s0, s0', s0'', k1', k2');
      assert s0' == s0'';
      bigstep_determ(c2, s0', s1, s1', k1-k1', k2-k2');
      assert s1 == s1';
```

We use some new syntax here, namely the assign-such-that statement:

```
var x :| p(x)
```

If `p` is a predicate, then you should think of this statement as returning an arbitrary value for `x`, such that it satisfies `p(x)`. We use this to reason about the intermediate environments `s0'` (corresponding to σ_0') and `s0''` (corresponding to σ_0''), as well as the step indices `k1'` and `k2'` needed by `bigstep`. In order to use the assign-such-that statement, Dafny must be able to determine that at least one value satisfying the right hand side exists. In this case, we know that such values exist because we know that `bigstep(c,s0,s1,k1)` and `bigstep(c,s0,s1',k2)` are true, and the semantic definition for `Seq` is:

```
    case Seq(c1, c2) =>
      exists k': nat, s'': Env ::
        0 < k' < k && bigstep(c1, s, s'', k') && bigstep(c2, s'', s', k-k')
```

The next part of the proof for `Seq` also uses a new concept. Namely, it calls the lemma recursively:

```
    bigstep_determ(c1, s0, s0', s0'', k1', k2');
```

Important: This is how we apply the inductive hypothesis in Dafny. Notice that we're applying this to smaller derivations than the ones our current preconditions give us: we know that `0 < k1' < k1` and `0 < k2' < k2`. By making this call on smaller arguments, the postcondition given by the lemma is now available to Dafny as an assertion would be. The following line that asserts `s0' == s0''` points this out for the reader's benefit, but is not strictly necessary; however, it is considered good form in this course to make your proofs explicit and easy to follow. We close out the case for `Seq` by making another inductive call on the second evaluation.

The next case covers **if** commands. Like the previous case, it makes an inductive call for each branch of the statment:

```
    case If(b, c1, c2) =>
      if(bigstep_bexp(b, s0)) {
        bigstep_determ(c1, s0, s1, s1', k1, k2);
        assert s1 == s1';
      } else {
        bigstep_determ(c2, s0, s1, s1', k1, k2);
        assert s1 == s1';
      }
```

Finally, we come to **while**. As we've discussed in lecture, semantically a **while** loop is like an **if** command composed with a **seq**. In lecture we said that in this case we could reason as follows. If the guard evaluates to *true*, then we can assume derivations:

$$\textsc{WhileTrue}\ \frac{\dfrac{T_1}{\langle b, \sigma_0\rangle \Downarrow_b \mathit{true}} \quad \dfrac{T_2}{\langle c, \sigma_0\rangle \Downarrow \sigma_0'} \quad \dfrac{T_3}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma_0'\rangle \Downarrow \sigma_1}}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma_0\rangle \Downarrow \sigma_1}$$

$$\textsc{WhileTrue}\ \frac{\dfrac{T_4}{\langle b, \sigma_0\rangle \Downarrow_b \mathit{true}} \quad \dfrac{T_5}{\langle c, \sigma_0\rangle \Downarrow \sigma_0''} \quad \dfrac{T_6}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma_0''\rangle \Downarrow \sigma_1'}}{\langle \textbf{while}\ b\ \textbf{do}\ c, \sigma_0\rangle \Downarrow \sigma_2}$$

Applying the inductive hypothesis twice leads us to conclude that $\sigma_0' = \sigma_0''$ and $\sigma_1 = \sigma_1'$. As this is very similar to the case for `Seq`, our proof in Dafny follows form:

```
    case While(b, c') =>
      if(bigstep_bexp(b, s0)) {
        var s0', k1' :| 0 < k1' < k1 &&
          bigstep(c', s0, s0', k1') &&
          bigstep(c, s0', s1, k1-k1');
        var s0'', k2' :| 0 < k2' < k2 &&
          bigstep(c', s0, s0'', k2') &&
          bigstep(c, s0'', s1', k2-k2');
        bigstep_determ(c', s0, s0', s0'', k1', k2');
        assert s0' == s0'';
        bigstep_determ(c, s0', s1, s1', k1-k1', k2-k2');
        assert s1 == s1';
```

We use assign-such-that, with a such-that condition coming directly from the big-step definition of **while**, to obtain intermediate states `s0'`, `s0''` and step indices `k1'`, `k2'`. We then apply the inductive hypothesis twice by making two recursive calls to the lemma, and conclude the WhileTrue case. WhileFalse is straigtforward, as it has the same semantics as **skip**.

```
      } else {
        assert s0 == s1 == s1';
      }
```

Although we've completed all of the relevant cases, Dafny does not accept our proof. The problem is well-foundedness, which requires providing a termination metric that the verifier will understand.

For this proof, we need a compound termination metric, as one case fails to decrease the step indices `k1`, `k2` (namely, the case for `If`), and another fails to decrease the command (i.e., `While`). We combine all three for our termination metric, on which Dafny will impose a lexicographic ordering. Putting all this together, we have the following lemma with proof:

```
lemma bigstep_determ(c: Com, s0: Env, s1: Env, s1': Env, k1: nat, k2: nat)
  decreases c, k1, k2
  requires bigstep(c, s0, s1, k1)
  requires bigstep(c, s0, s1', k2)
  ensures s1 == s1'
  ensures k1 == k2
{
  match(c) {
    case Skip =>
      assert s0 == s1 == s1';
    case Assign(v, a) =>
      assert s0[v := bigstep_aexp(a, s0)] == s1 == s1';
    case Seq(c1, c2) =>
      var s0', k1' :| 0 < k1' < k1 &&
        bigstep(c1, s0, s0', k1') &&
        bigstep(c2, s0', s1, k1-k1');
      var s0'', k2' :| 0 < k2' < k2 &&
        bigstep(c1, s0, s0'', k2') &&
        bigstep(c2, s0'', s1', k2-k2');
      bigstep_determ(c1, s0, s0', s0'', k1', k2');
      assert s0' == s0'';
      bigstep_determ(c2, s0', s1, s1', k1-k1', k2-k2');
      assert s1 == s1';
    case If(b, c1, c2) =>
      if(bigstep_bexp(b, s0)) {
        bigstep_determ(c1, s0, s1, s1', k1, k2);
        assert s1 == s1';
      } else {
        bigstep_determ(c2, s0, s1, s1', k1, k2);
        assert s1 == s1';
      }
    case While(b, c') =>
      if(bigstep_bexp(b, s0)) {
        var s0', k1' :| 0 < k1' < k1 &&
          bigstep(c', s0, s0', k1') &&
          bigstep(c, s0', s1, k1-k1');
        var s0'', k2' :| 0 < k2' < k2 &&
          bigstep(c', s0, s0'', k2') &&
          bigstep(c, s0'', s1', k2-k2');
        bigstep_determ(c', s0, s0', s0'', k1', k2');
        assert s0' == s0'';
        bigstep_determ(c, s0', s1, s1', k1-k1', k2-k2');
        assert s1 == s1';
      } else {
        assert s0 == s1 == s1';
      }
  }
}
```