SOFTWARE FOUNDATIONS

VOLUME 2: PROGRAMMING LANGUAGE FOUNDATIONS

PE

PARTIAL EVALUATION

```
(* Chapter written and maintained by Chung-chieh Shan *)
```

The Equiv chapter introduced constant folding as an example of a program transformation and proved that it preserves the meaning of programs. Constant folding operates on manifest constants such as `ANum` expressions. For example, it simplifies the command `Y ::= 3 + 1` to the command `Y ::= 4`. However, it does not propagate known constants along data flow. For example, it does not simplify the sequence

```
X ::= 3;; Y ::= X + 1
```

to

```
X ::= 3;; Y ::= 4
```

because it forgets that `X` is `3` by the time it gets to `Y`.

We might naturally want to enhance constant folding so that it propagates known constants and uses them to simplify programs. Doing so constitutes a rudimentary form of *partial evaluation*. As we will see, partial evaluation is so called because it is like running a program, except only part of the program can be evaluated because only part of the input to the program is known. For example, we can only simplify the program

```
X ::= 3;; Y ::= (X + 1) - Y
```

to

```
X ::= 3;; Y ::= 4 - Y
```

without knowing the initial value of `Y`.

```
Require Import Coq.Bool.Bool.
Require Import Coq.Arith.Arith.
Require Import Coq.Arith.EqNat.
Require Import Coq.omega.Omega.
Require Import Coq.Logic.FunctionalExtensionality.
```

```
Require Import Coq.Lists.List.
Import ListNotations.

Require Import Maps.
Require Import Imp.
Require Import Smallstep.
```

Generalizing Constant Folding

The starting point of partial evaluation is to represent our partial knowledge about the state. For example, between the two assignments above, the partial evaluator may know only that `X` is `3` and nothing about any other variable.

Partial States

Conceptually speaking, we can think of such partial states as the type `string → option nat` (as opposed to the type `string → nat` of concrete, full states). However, in addition to looking up and updating the values of individual variables in a partial state, we may also want to compare two partial states to see if and where they differ, to handle conditional control flow. It is not possible to compare two arbitrary functions in this way, so we represent partial states in a more concrete format: as a list of `string * nat` pairs.

```
Definition pe_state := list (string * nat).
```

The idea is that a variable (of type `string`) appears in the list if and only if we know its current `nat` value. The `pe_lookup` function thus interprets this concrete representation. (If the same variable appears multiple times in the list, the first occurrence wins, but we will define our partial evaluator to never construct such a `pe_state`.)

```
Fixpoint pe_lookup (pe_st : pe_state) (V:string) : option nat :=
  match pe_st with
  | [] ⇒ None
  | (V',n')::pe_st ⇒ if beq_string V V' then Some n'
                      else pe_lookup pe_st V
  end.
```

For example, `empty_pe_state` represents complete ignorance about every variable — the function that maps every identifier to `None`.

```
Definition empty_pe_state : pe_state := [].
```

More generally, if the `list` representing a `pe_state` does not contain some identifier, then that `pe_state` must map that identifier to `None`. Before we prove this fact, we first define a useful tactic for reasoning with `string` equality. The tactic

```
compare V V'
```

means to reason by cases over `beq_string V V'`. In the case where `V = V'`, the tactic substitutes `V` for `V'` throughout.

```
Tactic Notation "compare" ident(i) ident(j) :=
  let H := fresh "Heq" i j in
  destruct (beq_stringP i j);
  [ subst j | ].

Theorem pe_domain: ∀ pe_st V n,
  pe_lookup pe_st V = Some n →
  In V (map (@fst _ _) pe_st).
Proof. intros pe_st V n H. induction pe_st as [| [V' n'] pe_st].
  - (*  *) inversion H.
  - (* :: *) simpl in H. simpl. compare V V'; auto. Qed.
```

In what follows, we will make heavy use of the `In` property from the standard library, also defined in `Logic.v`:

```
Print In.
(* ===> Fixpoint In {A:Type} (a: A) (l:list A) : Prop :=
           match l with
           |   => False
           | b :: m => b = a \/ In a m
            end
        : forall A : Type, A -> list A -> Prop *)
```

Besides the various lemmas about `In` that we've already come across, the following one (taken from the standard library) will also be useful:

```
Check filter_In.
(* ===> filter_In : forall (A : Type) (f : A -
> bool) (x : A) (l : list A),
            In x (filter f l) <-> In x l /\ f x = true  *)
```

If a type `A` has an operator `beq` for testing equality of its elements, we can compute a boolean `inb beq a l` for testing whether `In a l` holds or not.

```
Fixpoint inb {A : Type} (beq : A → A → bool) (a : A) (l : list
A) :=
  match l with
  | [] ⇒ false
  | a'::l' ⇒ beq a a' || inb beq a l'
  end.
```

It is easy to relate `inb` to `In` with the `reflect` property:

```
Lemma inbP : ∀ A : Type, ∀ beq : A→A→bool,
  (∀ a1 a2, reflect (a1 = a2) (beq a1 a2)) →
  ∀ a l, reflect (In a l) (inb beq a l).
Proof.
  intros A beq beqP a l.
  induction l as [|a' l' IH].
  - constructor. intros [].
  - simpl. destruct (beqP a a').
    + subst. constructor. left. reflexivity.
    + simpl. destruct IH; constructor.
```

```
      * right. trivial.
      * intros [H₁ | H₂]; congruence.
Qed.
```

Arithmetic Expressions

Partial evaluation of `aexp` is straightforward — it is basically the same as constant folding, `fold_constants_aexp`, except that sometimes the partial state tells us the current value of a variable and we can replace it by a constant expression.

```
Fixpoint pe_aexp (pe_st : pe_state) (a : aexp) : aexp :=
  match a with
  | ANum n ⇒ ANum n
  | AId i ⇒ match pe_lookup pe_st i with (* <----- NEW *)
            | Some n ⇒ ANum n
            | None ⇒ AId i
            end
  | APlus a1 a2 ⇒
      match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
      | (ANum n1, ANum n2) ⇒ ANum (n1 + n2)
      | (a1', a2') ⇒ APlus a1' a2'
      end
  | AMinus a1 a2 ⇒
      match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
      | (ANum n1, ANum n2) ⇒ ANum (n1 - n2)
      | (a1', a2') ⇒ AMinus a1' a2'
      end
  | AMult a1 a2 ⇒
      match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
      | (ANum n1, ANum n2) ⇒ ANum (n1 * n2)
      | (a1', a2') ⇒ AMult a1' a2'
      end
  end.
```

This partial evaluator folds constants but does not apply the associativity of addition.

```
Open Scope aexp_scope.
Open Scope bexp_scope.

Example test_pe_aexp1:
  pe_aexp [(X,3)] (X + 1 + Y)
  = (4 + Y).
+

Example text_pe_aexp2:
  pe_aexp [(Y,3)] (X + 1 + Y)
  = (X + 1 + 3).
+
```

Now, in what sense is `pe_aexp` correct? It is reasonable to define the correctness of `pe_aexp` as follows: whenever a full state `st:state` is *consistent* with a partial state `pe_st:pe_state` (in other words, every variable to which `pe_st` assigns a value is assigned the same value by st), evaluating a and evaluating pe_aexp pe_st a in st yields the same result. This statement is indeed true.

```
Definition pe_consistent (st:state) (pe_st:pe_state) :=
  ∀ V n, Some n = pe_lookup pe_st V → st V = n.

Theorem pe_aexp_correct_weak: ∀ st pe_st, pe_consistent st pe_st
→
  ∀ a, aeval st a = aeval st (pe_aexp pe_st a).
Proof. unfold pe_consistent. intros st pe_st H a.
  induction a; simpl;
    try reflexivity;
    try (destruct (pe_aexp pe_st a1);
         destruct (pe_aexp pe_st a2);
         rewrite IHa1; rewrite IHa2; reflexivity).
  (* Compared to fold_constants_aexp_sound,
     the only interesting case is AId *)
  - (* AId *)
    remember (pe_lookup pe_st s) as l. destruct l.
    + (* Some *) rewrite H with (n:=n) by apply Heql.
reflexivity.
    + (* None *) reflexivity.
Qed.
```

However, we will soon want our partial evaluator to remove assignments. For example, it will simplify

```
X ::= 3;; Y ::= X - Y;; X ::= 4
```

to just

```
Y ::= 3 - Y;; X ::= 4
```

by delaying the assignment to X until the end. To accomplish this simplification, we need the result of partial evaluating

```
pe_aexp [(X,3)] (X - Y)
```

to be equal to 3 - Y and *not* the original expression X - Y. After all, it would be incorrect, not just inefficient, to transform

```
X ::= 3;; Y ::= X - Y;; X ::= 4
```

to

```
Y ::= X - Y;; X ::= 4
```

even though the output expressions 3 - Y and X - Y both satisfy the correctness criterion that we just proved. Indeed, if we were to just define pe_aexp pe_st a = a then the theorem pe_aexp_correct' would already trivially hold.

Instead, we want to prove that the pe_aexp is correct in a stronger sense: evaluating the expression produced by partial evaluation (aeval st (pe_aexp pe_st a)) must not depend on those parts of the full state st that are already specified in the partial state pe_st. To be more precise, let us define a function pe_override, which updates `st` with the contents of `pe_st`. In other words, `pe_override` carries out the assignments listed in `pe_st` on top of `st`.

```
Fixpoint pe_update (st:state) (pe_st:pe_state) : state :=
  match pe_st with
  | [] ⇒ st
  | (V,n)::pe_st ⇒ t_update (pe_update st pe_st) V n
  end.

Example test_pe_update:
  pe_update { Y --> 1 } [(X,3);(Z,2)]
  = { Y --> 1 ; Z --> 2 ; X --> 3 }.
```

+

Although `pe_update` operates on a concrete `list` representing a `pe_state`, its behavior is defined entirely by the `pe_lookup` interpretation of the `pe_state`.

```
Theorem pe_update_correct: ∀ st pe_st V0,
  pe_update st pe_st V0 =
  match pe_lookup pe_st V0 with
  | Some n ⇒ n
  | None ⇒ st V0
  end.
Proof. intros. induction pe_st as [| [V n] pe_st]. reflexivity.
  simpl in *. unfold t_update.
  compare V0 V; auto. rewrite <- beq_string_refl; auto. rewrite
false_beq_string; auto. Qed.
```

We can relate `pe_consistent` to `pe_update` in two ways. First, overriding a state with a partial state always gives a state that is consistent with the partial state. Second, if a state is already consistent with a partial state, then overriding the state with the partial state gives the same state.

```
Theorem pe_update_consistent: ∀ st pe_st,
  pe_consistent (pe_update st pe_st) pe_st.
Proof. intros st pe_st V n H. rewrite pe_update_correct.
  destruct (pe_lookup pe_st V); inversion H. reflexivity. Qed.

Theorem pe_consistent_update: ∀ st pe_st,
  pe_consistent st pe_st → ∀ V, st V = pe_update st pe_st V.
Proof. intros st pe_st H V. rewrite pe_update_correct.
  remember (pe_lookup pe_st V) as l. destruct l; auto. Qed.
```

Now we can state and prove that `pe_aexp` is correct in the stronger sense that will help us define the rest of the partial evaluator.

Intuitively, running a program using partial evaluation is a two-stage process. In the first, *static* stage, we partially evaluate the given program with respect to some partial state to get a *residual* program. In the second, *dynamic* stage, we evaluate the residual program with respect to the rest of the state. This dynamic state provides values for those variables that are unknown in the static (partial) state. Thus, the residual program should be equivalent to *prepending* the assignments listed in the partial state to the original program.

```
Theorem pe_aexp_correct: ∀ (pe_st:pe_state) (a:aexp) (st:state),
  aeval (pe_update st pe_st) a = aeval st (pe_aexp pe_st a).
Proof.
  intros pe_st a st.
  induction a; simpl;
    try reflexivity;
    try (destruct (pe_aexp pe_st a1);
         destruct (pe_aexp pe_st a2);
         rewrite IHa1; rewrite IHa2; reflexivity).
  (* Compared to fold_constants_aexp_sound, the only
     interesting case is AId. *)
  rewrite pe_update_correct. destruct (pe_lookup pe_st s);
reflexivity.
Qed.
```

Boolean Expressions

The partial evaluation of boolean expressions is similar. In fact, it is entirely analogous to the constant folding of boolean expressions, because our language has no boolean variables.

```
Fixpoint pe_bexp (pe_st : pe_state) (b : bexp) : bexp :=
  match b with
  | BTrue ⇒ BTrue
  | BFalse ⇒ BFalse
  | BEq a1 a2 ⇒
      match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
      | (ANum n1, ANum n2) ⇒ if beq_nat n1 n2 then BTrue else
BFalse
      | (a1', a2') ⇒ BEq a1' a2'
      end
  | BLe a1 a2 ⇒
      match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
      | (ANum n1, ANum n2) ⇒ if leb n1 n2 then BTrue else BFalse
      | (a1', a2') ⇒ BLe a1' a2'
      end
  | BNot b1 ⇒
      match (pe_bexp pe_st b1) with
      | BTrue ⇒ BFalse
      | BFalse ⇒ BTrue
      | b1' ⇒ BNot b1'
      end
  | BAnd b1 b2 ⇒
      match (pe_bexp pe_st b1, pe_bexp pe_st b2) with
      | (BTrue, BTrue) ⇒ BTrue
      | (BTrue, BFalse) ⇒ BFalse
      | (BFalse, BTrue) ⇒ BFalse
      | (BFalse, BFalse) ⇒ BFalse
      | (b1', b2') ⇒ BAnd b1' b2'
```

```
      end
    end.

  Example test_pe_bexp1:
    pe_bexp [(X,3)] (!(X ≤ 3))
    = false.
  +

  Example test_pe_bexp2: ∀ b:bexp,
    b = !(X ≤ (X + 1)) →
    pe_bexp [] b = b.
  Proof. intros b H. rewrite → H. reflexivity. Qed.
```

The correctness of pe_bexp is analogous to the correctness of pe_aexp above.

```
Theorem pe_bexp_correct: ∀ (pe_st:pe_state) (b:bexp) (st:state),
  beval (pe_update st pe_st) b = beval st (pe_bexp pe_st b).
Proof.
  intros pe_st b st.
  induction b; simpl;
    try reflexivity;
    try (remember (pe_aexp pe_st a) as a';
         remember (pe_aexp pe_st a0) as a0';
         assert (Ha: aeval (pe_update st pe_st) a = aeval st
a');
         assert (Ha0: aeval (pe_update st pe_st) a0 = aeval st
a0');
           try (subst; apply pe_aexp_correct);
         destruct a'; destruct a0'; rewrite Ha; rewrite Ha0;
         simpl; try destruct (beq_nat n n0);
         try destruct (leb n n0); reflexivity);
    try (destruct (pe_bexp pe_st b); rewrite IHb; reflexivity);
    try (destruct (pe_bexp pe_st b1);
         destruct (pe_bexp pe_st b2);
         rewrite IHb1; rewrite IHb2; reflexivity).
Qed.
```

Partial Evaluation of Commands, Without Loops

What about the partial evaluation of commands? The analogy between partial evaluation and full evaluation continues: Just as full evaluation of a command turns an initial state into a final state, partial evaluation of a command turns an initial partial state into a final partial state. The difference is that, because the state is partial, some parts of the command may not be executable at the static stage. Therefore, just as pe_aexp returns a residual aexp and pe_bexp returns a residual bexp above, partially evaluating a command yields a residual command.

Another way in which our partial evaluator is similar to a full evaluator is that it does not terminate on all commands. It is not hard to build a partial evaluator that terminates on all commands; what is hard is building a partial evaluator that terminates on all commands yet automatically performs desired optimizations such as unrolling loops. Often a partial evaluator can be coaxed into terminating more often and performing more optimizations by writing the source program differently so that the separation between static and dynamic information becomes more apparent. Such coaxing is the art of *binding-time improvement*. The binding time of a variable tells when its value is known — either "static", or "dynamic."

Anyway, for now we will just live with the fact that our partial evaluator is not a total function from the source command and the initial partial state to the residual command and the final partial state. To model this non-termination, just as with the full evaluation of commands, we use an inductively defined relation. We write

```
c1 / st \\ c1' / st'
```

to mean that partially evaluating the source command `c1` in the initial partial state `st` yields the residual command `c1'` and the final partial state `st'`. For example, we want something like

```
(X ::= 3 ;; Y ::= Z * (X + X)
/ [] \\ (Y ::= Z * 6) / [(X,3)]
```

to hold. The assignment to `X` appears in the final partial state, not the residual command.

Assignment

Let's start by considering how to partially evaluate an assignment. The two assignments in the source program above needs to be treated differently. The first assignment `X ::= 3`, is *static*: its right-hand-side is a constant (more generally, simplifies to a constant), so we should update our partial state at `X` to `3` and produce no residual code. (Actually, we produce a residual `SKIP`.) The second assignment `Y ::= Z * (X + X)` is *dynamic*: its right-hand-side does not simplify to a constant, so we should leave it in the residual code and remove `Y`, if present, from our partial state. To implement these two cases, we define the functions `pe_add` and `pe_remove`. Like `pe_update` above, these functions operate on a concrete `list` representing a `pe_state`, but the theorems `pe_add_correct` and `pe_remove_correct` specify their behavior by the `pe_lookup` interpretation of the `pe_state`.

```
Fixpoint pe_remove (pe_st:pe_state) (V:string) : pe_state :=
  match pe_st with
  | [] ⇒ []
  | (V',n')::pe_st ⇒ if beq_string V V' then pe_remove pe_st V
                       else (V',n') :: pe_remove pe_st V
  end.

Theorem pe_remove_correct: ∀ pe_st V V0,
  pe_lookup (pe_remove pe_st V) V0
  = if beq_string V V0 then None else pe_lookup pe_st V0.
Proof. intros pe_st V V0. induction pe_st as [| [V' n'] pe_st].
```

```
    - (*   *) destruct (beq_string V V0); reflexivity.
    - (* :: *) simpl. compare V V'.
      + (* equal *) rewrite IHpe_st.
        destruct (beq_stringP V V0). reflexivity.
        rewrite false_beq_string; auto.
      + (* not equal *) simpl. compare V0 V'.
        * (* equal *) rewrite false_beq_string; auto.
        * (* not equal *) rewrite IHpe_st. reflexivity.
Qed.

Definition pe_add (pe_st:pe_state) (V:string) (n:nat) : pe_state
:=
  (V,n) :: pe_remove pe_st V.

Theorem pe_add_correct: ∀ pe_st V n V0,
  pe_lookup (pe_add pe_st V n) V0
  = if beq_string V V0 then Some n else pe_lookup pe_st V0.
Proof. intros pe_st V n V0. unfold pe_add. simpl.
  compare V V0.
  - (* equal *) rewrite <- beq_string_refl; auto.
  - (* not equal *) rewrite pe_remove_correct.
    repeat rewrite false_beq_string; auto.
Qed.
```

We will use the two theorems below to show that our partial evaluator correctly deals with dynamic assignments and static assignments, respectively.

```
Theorem pe_update_update_remove: ∀ st pe_st V n,
  t_update (pe_update st pe_st) V n =
  pe_update (t_update st V n) (pe_remove pe_st V).
Proof. intros st pe_st V n. apply functional_extensionality.
  intros V0. unfold t_update. rewrite !pe_update_correct.
  rewrite pe_remove_correct. destruct (beq_string V V0);
reflexivity.
  Qed.

Theorem pe_update_update_add: ∀ st pe_st V n,
  t_update (pe_update st pe_st) V n =
  pe_update st (pe_add pe_st V n).
Proof. intros st pe_st V n. apply functional_extensionality.
intros V0.
  unfold t_update. rewrite !pe_update_correct. rewrite
pe_add_correct.
  destruct (beq_string V V0); reflexivity. Qed.
```

Conditional

Trickier than assignments to partially evaluate is the conditional, `IFB` b_1 `THEN` c_1 `ELSE` c_2 `FI`. If b_1 simplifies to `BTrue` or `BFalse` then it's easy: we know which branch will be taken, so just take that branch. If b_1 does not simplify to a constant, then we need to take both branches, and the final partial state may differ between the two branches!

The following program illustrates the difficulty:

```
X ::= 3;;
IFB Y ≤ 4 THEN
    Y ::= 4;;
    IFB X ≤ Y THEN Y ::= 999 ELSE SKIP FI
ELSE SKIP FI
```

Suppose the initial partial state is empty. We don't know statically how `Y` compares to `4`, so we must partially evaluate both branches of the (outer) conditional. On the `THEN` branch, we know that `Y` is set to `4` and can even use that knowledge to simplify the code somewhat. On the `ELSE` branch, we still don't know the exact value of `Y` at the end. What should the final partial state and residual program be?

One way to handle such a dynamic conditional is to take the intersection of the final partial states of the two branches. In this example, we take the intersection of `(Y,4),(X,3)` and `(X,3)`, so the overall final partial state is `(X,3)`. To compensate for forgetting that `Y` is `4`, we need to add an assignment `Y ::= 4` to the end of the `THEN` branch. So, the residual program will be something like

```
SKIP;;
IFB Y ≤ 4 THEN
    SKIP;;
    SKIP;;
    Y ::= 4
ELSE SKIP FI
```

Programming this case in Coq calls for several auxiliary functions: we need to compute the intersection of two `pe_states` and turn their difference into sequences of assignments.

First, we show how to compute whether two `pe_states` to disagree at a given variable. In the theorem `pe_disagree_domain`, we prove that two `pe_states` can only disagree at variables that appear in at least one of them.

```
Definition pe_disagree_at (pe_st1 pe_st2 : pe_state) (V:string) :
bool :=
  match pe_lookup pe_st1 V, pe_lookup pe_st2 V with
  | Some x, Some y ⇒ negb (beq_nat x y)
  | None, None ⇒ false
  | _, _ ⇒ true
  end.

Theorem pe_disagree_domain: ∀ (pe_st1 pe_st2 : pe_state)
(V:string),
  true = pe_disagree_at pe_st1 pe_st2 V →
  In V (map (@fst _ _) pe_st1 ++ map (@fst _ _) pe_st2).
Proof. unfold pe_disagree_at. intros pe_st1 pe_st2 V H.
  apply in_app_iff.
  remember (pe_lookup pe_st1 V) as lookup1.
  destruct lookup1 as [n1|]. left. apply pe_domain with n1. auto.
  remember (pe_lookup pe_st2 V) as lookup2.
```

```
    destruct lookup2 as [n2|]. right. apply pe_domain with n2.
  auto.
    inversion H. Qed.
```

We define the `pe_compare` function to list the variables where two given `pe_states` disagree. This list is exact, according to the theorem `pe_compare_correct`: a variable appears on the list if and only if the two given `pe_states` disagree at that variable. Furthermore, we use the `pe_unique` function to eliminate duplicates from the list.

```
  Fixpoint pe_unique (l : list string) : list string :=
    match l with
    | [] ⇒ []
    | x::l ⇒
        x :: filter (fun y ⇒ if beq_string x y then false else
  true) (pe_unique l)
    end.

  Theorem pe_unique_correct: ∀ l x,
    In x l ↔ In x (pe_unique l).
  Proof. intros l x. induction l as [| h t]. reflexivity.
    simpl in *. split.
    - (* -> *)
      intros. inversion H; clear H.
        left. assumption.
        destruct (beq_stringP h x).
           left. assumption.
           right. apply filter_In. split.
             apply IHt. assumption.
             rewrite false_beq_string; auto.
    - (* <- *)
      intros. inversion H; clear H.
         left. assumption.
         apply filter_In in H0. inversion H0. right. apply IHt.
  assumption.
  Qed.

  Definition pe_compare (pe_st1 pe_st2 : pe_state) : list string :=
    pe_unique (filter (pe_disagree_at pe_st1 pe_st2)
      (map (@fst _ _) pe_st1 ++ map (@fst _ _) pe_st2)).

  Theorem pe_compare_correct: ∀ pe_st1 pe_st2 V,
    pe_lookup pe_st1 V = pe_lookup pe_st2 V ↔
    ¬ In V (pe_compare pe_st1 pe_st2).
  Proof. intros pe_st1 pe_st2 V.
    unfold pe_compare. rewrite <- pe_unique_correct. rewrite
  filter_In.
    split; intros Heq.
    - (* -> *)
      intro. destruct H. unfold pe_disagree_at in H0. rewrite Heq
  in H0.
      destruct (pe_lookup pe_st2 V).
      rewrite <- beq_nat_refl in H0. inversion H0.
```

```
      inversion H0.
    - (* <- *)
      assert (Hagree: pe_disagree_at pe_st1 pe_st2 V = false).
      { (* Proof of assertion *)
        remember (pe_disagree_at pe_st1 pe_st2 V) as disagree.
        destruct disagree; [| reflexivity].
        apply pe_disagree_domain in Heqdisagree.
        exfalso. apply Heq. split. assumption. reflexivity. }
      unfold pe_disagree_at in Hagree.
      destruct (pe_lookup pe_st1 V) as [n1|];
      destruct (pe_lookup pe_st2 V) as [n2|];
        try reflexivity; try solve_by_invert.
      rewrite negb_false_iff in Hagree.
      apply beq_nat_true in Hagree. subst. reflexivity. Qed.
```

The intersection of two partial states is the result of removing from one of them all the variables where the two disagree. We define the function `pe_removes`, in terms of `pe_remove` above, to perform such a removal of a whole list of variables at once.

The theorem `pe_compare_removes` testifies that the `pe_lookup` interpretation of the result of this intersection operation is the same no matter which of the two partial states we remove the variables from. Because `pe_update` only depends on the `pe_lookup` interpretation of partial states, `pe_update` also does not care which of the two partial states we remove the variables from; that theorem `pe_compare_update` is used in the correctness proof shortly.

```
Fixpoint pe_removes (pe_st:pe_state) (ids : list string) :
pe_state :=
  match ids with
  | [] ⇒ pe_st
  | V::ids ⇒ pe_remove (pe_removes pe_st ids) V
  end.

Theorem pe_removes_correct: ∀ pe_st ids V,
  pe_lookup (pe_removes pe_st ids) V =
  if inb beq_string V ids then None else pe_lookup pe_st V.
Proof. intros pe_st ids V. induction ids as [| V' ids].
reflexivity.
  simpl. rewrite pe_remove_correct. rewrite IHids.
  compare V' V.
  - rewrite <- beq_string_refl. reflexivity.
  - rewrite false_beq_string; try congruence. reflexivity.
Qed.

Theorem pe_compare_removes: ∀ pe_st1 pe_st2 V,
  pe_lookup (pe_removes pe_st1 (pe_compare pe_st1 pe_st2)) V =
  pe_lookup (pe_removes pe_st2 (pe_compare pe_st1 pe_st2)) V.
Proof.
  intros pe_st1 pe_st2 V. rewrite !pe_removes_correct.
  destruct (inbP _ _ beq_stringP V (pe_compare pe_st1 pe_st2)).
  - reflexivity.
  - apply pe_compare_correct. auto. Qed.
```

```
Theorem pe_compare_update: ∀ pe_st₁ pe_st₂ st,
  pe_update st (pe_removes pe_st₁ (pe_compare pe_st₁ pe_st₂)) =
  pe_update st (pe_removes pe_st₂ (pe_compare pe_st₁ pe_st₂)).
Proof. intros. apply functional_extensionality. intros V.
  rewrite !pe_update_correct. rewrite pe_compare_removes.
reflexivity.
Qed.
```

Finally, we define an `assign` function to turn the difference between two partial states into a sequence of assignment commands. More precisely, `assign pe_st ids` generates an assignment command for each variable listed in `ids`.

```
Fixpoint assign (pe_st : pe_state) (ids : list string) : com :=
  match ids with
  | [] ⇒ SKIP
  | V::ids ⇒ match pe_lookup pe_st V with
             | Some n ⇒ (assign pe_st ids;; V ::= ANum n)
             | None ⇒ assign pe_st ids
             end
  end.
```

The command generated by `assign` always terminates, because it is just a sequence of assignments. The (total) function `assigned` below computes the effect of the command on the (dynamic state). The theorem `assign_removes` then confirms that the generated assignments perfectly compensate for removing the variables from the partial state.

```
Definition assigned (pe_st:pe_state) (ids : list string)
(st:state) : state :=
  fun V ⇒ if inb beq_string V ids then
              match pe_lookup pe_st V with
              | Some n ⇒ n
              | None ⇒ st V
              end
          else st V.

Theorem assign_removes: ∀ pe_st ids st,
  pe_update st pe_st =
  pe_update (assigned pe_st ids st) (pe_removes pe_st ids).
Proof. intros pe_st ids st. apply functional_extensionality.
intros V.
  rewrite !pe_update_correct. rewrite pe_removes_correct. unfold
assigned.
  destruct (inbP _ _ beq_stringP V ids); destruct (pe_lookup
pe_st V); reflexivity.
Qed.

Lemma ceval_extensionality: ∀ c st st₁ st₂,
  c / st \\ st₁ → (∀ V, st₁ V = st₂ V) → c / st \\ st₂.
Proof. intros c st st₁ st₂ H Heq.
  apply functional_extensionality in Heq. rewrite <- Heq. apply
H. Qed.
```

```
Theorem eval_assign: ∀ pe_st ids st,
  assign pe_st ids / st \\ assigned pe_st ids st.
Proof. intros pe_st ids st. induction ids as [| V ids]; simpl.
  - (*  *) eapply ceval_extensionality. apply E_Skip.
reflexivity.
  - (* V::ids *)
    remember (pe_lookup pe_st V) as lookup. destruct lookup.
    + (* Some *) eapply E_Seq. apply IHids. unfold assigned.
simpl.
      eapply ceval_extensionality. apply E_Ass. simpl.
reflexivity.
      intros V0. unfold t_update. compare V V0.
      * (* equal *) rewrite <- Heqlookup. rewrite <-
beq_string_refl. reflexivity.
      * (* not equal *) rewrite false_beq_string; simpl;
congruence.
    + (* None *) eapply ceval_extensionality. apply IHids.
      unfold assigned. intros V0. simpl. compare V V0.
      * (* equal *) rewrite <- Heqlookup.
        rewrite <- beq_string_refl.
        destruct (inbP _ _ beq_stringP V ids); reflexivity.
      * (* not equal *) rewrite false_beq_string; simpl;
congruence.
Qed.
```

The Partial Evaluation Relation

At long last, we can define a partial evaluator for commands without loops, as an inductive relation! The inequality conditions in `PE_AssDynamic` and `PE_If` are just to keep the partial evaluator deterministic; they are not required for correctness.

```
Reserved Notation "c1 '/' st '\\' c1' '/' st'"
  (at level 40, st at level 39, c1' at level 39).

Inductive pe_com : com → pe_state → com → pe_state → Prop :=
  | PE_Skip : ∀ pe_st,
      SKIP / pe_st \\ SKIP / pe_st
  | PE_AssStatic : ∀ pe_st a1 n1 l,
      pe_aexp pe_st a1 = ANum n1 →
      (l ::= a1) / pe_st \\ SKIP / pe_add pe_st l n1
  | PE_AssDynamic : ∀ pe_st a1 a1' l,
      pe_aexp pe_st a1 = a1' →
      (∀ n, a1' ≠ ANum n) →
      (l ::= a1) / pe_st \\ (l ::= a1') / pe_remove pe_st l
  | PE_Seq : ∀ pe_st pe_st' pe_st'' c1 c2 c1' c2',
      c1 / pe_st \\ c1' / pe_st' →
      c2 / pe_st' \\ c2' / pe_st'' →
      (c1 ;; c2) / pe_st \\ (c1' ;; c2') / pe_st''
  | PE_IfTrue : ∀ pe_st pe_st' b1 c1 c2 c1',
      pe_bexp pe_st b1 = BTrue →
      c1 / pe_st \\ c1' / pe_st' →
      (IFB b1 THEN c1 ELSE c2 FI) / pe_st \\ c1' / pe_st'
```

```
  | PE_IfFalse : ∀ pe_st pe_st' b1 c1 c2 c2',
      pe_bexp pe_st b1 = BFalse →
      c2 / pe_st \\ c2' / pe_st' →
      (IFB b1 THEN c1 ELSE c2 FI) / pe_st \\ c2' / pe_st'
  | PE_If : ∀ pe_st pe_st1 pe_st2 b1 c1 c2 c1' c2',
      pe_bexp pe_st b1 ≠ BTrue →
      pe_bexp pe_st b1 ≠ BFalse →
      c1 / pe_st \\ c1' / pe_st1 →
      c2 / pe_st \\ c2' / pe_st2 →
      (IFB b1 THEN c1 ELSE c2 FI) / pe_st
        \\ (IFB pe_bexp pe_st b1
             THEN c1' ;; assign pe_st1 (pe_compare pe_st1 pe_st2)
             ELSE c2' ;; assign pe_st2 (pe_compare pe_st1 pe_st2)
FI)
            / pe_removes pe_st1 (pe_compare pe_st1 pe_st2)

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

Hint Constructors pe_com.
Hint Constructors ceval.
```

Examples

Below are some examples of using the partial evaluator. To make the `pe_com` relation actually usable for automatic partial evaluation, we would need to define more automation tactics in Coq. That is not hard to do, but it is not needed here.

```
Example pe_example1:
  (X ::= 3 ;; Y ::= Z * (X + X))
  / [] \\ (SKIP;; Y ::= Z * 6) / [(X,3)].
Proof. eapply PE_Seq. eapply PE_AssStatic. reflexivity.
  eapply PE_AssDynamic. reflexivity. intros n H. inversion H.
Qed.

Example pe_example2:
  (X ::= 3 ;; IFB X ≤ 4 THEN X ::= 4 ELSE SKIP FI)
  / [] \\ (SKIP;; SKIP) / [(X,4)].
Proof. eapply PE_Seq. eapply PE_AssStatic. reflexivity.
  eapply PE_IfTrue. reflexivity.
  eapply PE_AssStatic. reflexivity. Qed.

Example pe_example3:
  (X ::= 3;;
   IFB Y ≤ 4 THEN
     Y ::= 4;;
     IFB X = Y THEN Y ::= 999 ELSE SKIP FI
   ELSE SKIP FI) / []
  \\ (SKIP;;
       IFB Y ≤ 4 THEN
         (SKIP;; SKIP);; (SKIP;; Y ::= 4)
       ELSE SKIP;; SKIP FI)
      / [(X,3)].
Proof. erewrite f_equal2 with (f := fun c st ⇒ _ / _ \\ c / st).
```

```
  eapply PE_Seq. eapply PE_AssStatic. reflexivity.
  eapply PE_If; intuition eauto; try solve_by_invert.
  econstructor. eapply PE_AssStatic. reflexivity.
  eapply PE_IfFalse. reflexivity. econstructor.
  reflexivity. reflexivity. Qed.
```

Correctness of Partial Evaluation

Finally let's prove that this partial evaluator is correct!

```
Reserved Notation "c' '/' pe_st' '/' st '\\' st''"
  (at level 40, pe_st' at level 39, st at level 39).

Inductive pe_ceval
  (c':com) (pe_st':pe_state) (st:state) (st'':state) : Prop :=
  | pe_ceval_intro : ∀ st',
    c' / st \\ st' →
    pe_update st' pe_st' = st'' →
    c' / pe_st' / st \\ st''
  where "c' '/' pe_st' '/' st '\\' st''" := (pe_ceval c' pe_st'
st st'').

Hint Constructors pe_ceval.

Theorem pe_com_complete:
  ∀ c pe_st pe_st' c', c / pe_st \\ c' / pe_st' →
  ∀ st st'',
  (c / pe_update st pe_st \\ st'') →
  (c' / pe_st' / st \\ st'').
Proof. intros c pe_st pe_st' c' Hpe.
  induction Hpe; intros st st'' Heval;
  try (inversion Heval; subst;
       try (rewrite → pe_bexp_correct, → H in *;
solve_by_invert);
       []);
  eauto.
  - (* PE_AssStatic *) econstructor. econstructor.
    rewrite → pe_aexp_correct. rewrite <- pe_update_update_add.
    rewrite → H. reflexivity.
  - (* PE_AssDynamic *) econstructor. econstructor. reflexivity.
    rewrite → pe_aexp_correct. rewrite <-
pe_update_update_remove.
    reflexivity.
  - (* PE_Seq *)
    edestruct IHHpe1. eassumption. subst.
    edestruct IHHpe2. eassumption.
    eauto.
  - (* PE_If *) inversion Heval; subst.
    + (* E'IfTrue *) edestruct IHHpe1. eassumption.
      econstructor. apply E_IfTrue. rewrite <- pe_bexp_correct.
assumption.
      eapply E_Seq. eassumption. apply eval_assign.
      rewrite <- assign_removes. eassumption.
    + (* E_IfFalse *) edestruct IHHpe2. eassumption.
      econstructor. apply E_IfFalse. rewrite <- pe_bexp_correct.
assumption.
      eapply E_Seq. eassumption. apply eval_assign.
      rewrite → pe_compare_update.
```

```
      rewrite <- assign_removes. eassumption.
Qed.

Theorem pe_com_sound:
  ∀ c pe_st pe_st' c', c / pe_st \\ c' / pe_st' →
  ∀ st st'',
  (c' / pe_st' / st \\ st'') →
  (c / pe_update st pe_st \\ st'').
Proof. intros c pe_st pe_st' c' Hpe.
  induction Hpe;
    intros st st'' [st' Heval Heq];
    try (inversion Heval; []; subst); auto.
  - (* PE_AssStatic *) rewrite <- pe_update_update_add. apply
E_Ass.
    rewrite → pe_aexp_correct. rewrite → H. reflexivity.
  - (* PE_AssDynamic *) rewrite <- pe_update_update_remove.
apply E_Ass.
    rewrite <- pe_aexp_correct. reflexivity.
  - (* PE_Seq *) eapply E_Seq; eauto.
  - (* PE_IfTrue *) apply E_IfTrue.
    rewrite → pe_bexp_correct. rewrite → H. reflexivity. eauto.
  - (* PE_IfFalse *) apply E_IfFalse.
    rewrite → pe_bexp_correct. rewrite → H. reflexivity. eauto.
  - (* PE_If *)
    inversion Heval; subst; inversion H7;
      (eapply ceval_deterministic in H8; [| apply eval_assign]);
subst.
    + (* E_IfTrue *)
      apply E_IfTrue. rewrite → pe_bexp_correct. assumption.
      rewrite <- assign_removes. eauto.
    + (* E_IfFalse *)
      rewrite → pe_compare_update.
      apply E_IfFalse. rewrite → pe_bexp_correct. assumption.
      rewrite <- assign_removes. eauto.
Qed.
```

The main theorem. Thanks to David Menendez for this formulation!

```
Corollary pe_com_correct:
  ∀ c pe_st pe_st' c', c / pe_st \\ c' / pe_st' →
  ∀ st st'',
  (c / pe_update st pe_st \\ st'') ↔
  (c' / pe_st' / st \\ st'').
Proof. intros c pe_st pe_st' c' H st st''. split.
  - (* -> *) apply pe_com_complete. apply H.
  - (* <- *) apply pe_com_sound. apply H.
Qed.
```

Partial Evaluation of Loops

It may seem straightforward at first glance to extend the partial evaluation relation `pe_com` above to loops. Indeed, many loops are easy to deal with. Considered this repeated-squaring loop, for example:

```
WHILE 1 ≤ X DO
    Y ::= Y * Y;;
    X ::= X - 1
END
```

If we know neither `X` nor `Y` statically, then the entire loop is dynamic and the residual command should be the same. If we know `X` but not `Y`, then the loop can be unrolled all the way and the residual command should be, for example,

```
Y ::= Y * Y;;
Y ::= Y * Y;;
Y ::= Y * Y
```

if `X` is initially `3` (and finally `0`). In general, a loop is easy to partially evaluate if the final partial state of the loop body is equal to the initial state, or if its guard condition is static.

But there are other loops for which it is hard to express the residual program we want in Imp. For example, take this program for checking whether `Y` is even or odd:

```
X ::= 0;;
WHILE 1 ≤ Y DO
    Y ::= Y - 1 ;;
    X ::= 1 - X
END
```

The value of `X` alternates between `0` and `1` during the loop. Ideally, we would like to unroll this loop, not all the way but *two-fold*, into something like

```
WHILE 1 ≤ Y DO
    Y ::= Y - 1;;
    IF 1 ≤ Y THEN
        Y ::= Y - 1
    ELSE
        X ::= 1;; EXIT
    FI
END;;
X ::= 0
```

Unfortunately, there is no `EXIT` command in Imp. Without extending the range of control structures available in our language, the best we can do is to repeat loop-guard tests or add flag variables. Neither option is terribly attractive.

Still, as a digression, below is an attempt at performing partial evaluation on Imp commands. We add one more command argument `c''` to the `pe_com` relation, which keeps track of a loop to roll up.

```
Module Loop.

Reserved Notation "c1 '/' st '\\' c1' '/' st' '/' c''"
  (at level 40, st at level 39, c1' at level 39, st' at level
```

```
39).

Inductive pe_com : com → pe_state → com → pe_state → com → Prop
:=
  | PE_Skip : ∀ pe_st,
      SKIP / pe_st \\ SKIP / pe_st / SKIP
  | PE_AssStatic : ∀ pe_st a1 n1 l,
      pe_aexp pe_st a1 = ANum n1 →
      (l ::= a1) / pe_st \\ SKIP / pe_add pe_st l n1 / SKIP
  | PE_AssDynamic : ∀ pe_st a1 a1' l,
      pe_aexp pe_st a1 = a1' →
      (∀ n, a1' ≠ ANum n) →
      (l ::= a1) / pe_st \\ (l ::= a1') / pe_remove pe_st l / SKIP
  | PE_Seq : ∀ pe_st pe_st' pe_st'' c1 c2 c1' c2' c'',
      c1 / pe_st \\ c1' / pe_st' / SKIP →
      c2 / pe_st' \\ c2' / pe_st'' / c'' →
      (c1 ;; c2) / pe_st \\ (c1' ;; c2') / pe_st'' / c''
  | PE_IfTrue : ∀ pe_st pe_st' b1 c1 c2 c1' c'',
      pe_bexp pe_st b1 = BTrue →
      c1 / pe_st \\ c1' / pe_st' / c'' →
      (IFB b1 THEN c1 ELSE c2 FI) / pe_st \\ c1' / pe_st' / c''
  | PE_IfFalse : ∀ pe_st pe_st' b1 c1 c2 c2' c'',
      pe_bexp pe_st b1 = BFalse →
      c2 / pe_st \\ c2' / pe_st' / c'' →
      (IFB b1 THEN c1 ELSE c2 FI) / pe_st \\ c2' / pe_st' / c''
  | PE_If : ∀ pe_st pe_st1 pe_st2 b1 c1 c2 c1' c2' c'',
      pe_bexp pe_st b1 ≠ BTrue →
      pe_bexp pe_st b1 ≠ BFalse →
      c1 / pe_st \\ c1' / pe_st1 / c'' →
      c2 / pe_st \\ c2' / pe_st2 / c'' →
      (IFB b1 THEN c1 ELSE c2 FI) / pe_st
        \\ (IFB pe_bexp pe_st b1
               THEN c1' ;; assign pe_st1 (pe_compare pe_st1 pe_st2)
               ELSE c2' ;; assign pe_st2 (pe_compare pe_st1 pe_st2)
FI)
            / pe_removes pe_st1 (pe_compare pe_st1 pe_st2)
            / c''
  | PE_WhileFalse : ∀ pe_st b1 c1,
      pe_bexp pe_st b1 = BFalse →
      (WHILE b1 DO c1 END) / pe_st \\ SKIP / pe_st / SKIP
  | PE_WhileTrue : ∀ pe_st pe_st' pe_st'' b1 c1 c1' c2' c2'',
      pe_bexp pe_st b1 = BTrue →
      c1 / pe_st \\ c1' / pe_st' / SKIP →
      (WHILE b1 DO c1 END) / pe_st' \\ c2' / pe_st'' / c2'' →
      pe_compare pe_st pe_st'' ≠ [] →
      (WHILE b1 DO c1 END) / pe_st \\ (c1';;c2') / pe_st'' / c2''
  | PE_While : ∀ pe_st pe_st' pe_st'' b1 c1 c1' c2' c2'',
      pe_bexp pe_st b1 ≠ BFalse →
```

```
      pe_bexp pe_st b₁ ≠ BTrue →
      c₁ / pe_st \\ c₁' / pe_st' / SKIP →
      (WHILE b₁ DO c₁ END) / pe_st' \\ c₂' / pe_st'' / c₂'' →
      pe_compare pe_st pe_st'' ≠ [] →
      (c₂'' = SKIP ∨ c₂'' = WHILE b₁ DO c₁ END) →
      (WHILE b₁ DO c₁ END) / pe_st
        \\ (IFB pe_bexp pe_st b₁
             THEN c₁';; c₂';; assign pe_st'' (pe_compare pe_st
pe_st'')
             ELSE assign pe_st (pe_compare pe_st pe_st'') FI)
            / pe_removes pe_st (pe_compare pe_st pe_st'')
            / c₂''
  | PE_WhileFixedEnd : ∀ pe_st b₁ c₁,
      pe_bexp pe_st b₁ ≠ BFalse →
      (WHILE b₁ DO c₁ END) / pe_st \\ SKIP / pe_st / (WHILE b₁ DO
c₁ END)
  | PE_WhileFixedLoop : ∀ pe_st pe_st' pe_st'' b₁ c₁ c₁' c₂',
      pe_bexp pe_st b₁ = BTrue →
      c₁ / pe_st \\ c₁' / pe_st' / SKIP →
      (WHILE b₁ DO c₁ END) / pe_st'
        \\ c₂' / pe_st'' / (WHILE b₁ DO c₁ END) →
      pe_compare pe_st pe_st'' = [] →
      (WHILE b₁ DO c₁ END) / pe_st
        \\ (WHILE BTrue DO SKIP END) / pe_st / SKIP
      (* Because we have an infinite loop, we should actually
         start to throw away the rest of the program:
         (WHILE b₁ DO c₁ END) / pe_st
         \\ SKIP / pe_st / (WHILE BTrue DO SKIP END) *)
  | PE_WhileFixed : ∀ pe_st pe_st' pe_st'' b₁ c₁ c₁' c₂',
      pe_bexp pe_st b₁ ≠ BFalse →
      pe_bexp pe_st b₁ ≠ BTrue →
      c₁ / pe_st \\ c₁' / pe_st' / SKIP →
      (WHILE b₁ DO c₁ END) / pe_st'
        \\ c₂' / pe_st'' / (WHILE b₁ DO c₁ END) →
      pe_compare pe_st pe_st'' = [] →
      (WHILE b₁ DO c₁ END) / pe_st
        \\ (WHILE pe_bexp pe_st b₁ DO c₁';; c₂' END) / pe_st /
SKIP

  where "c₁ '/' st '\\' c₁' '/' st' '/' c''" := (pe_com c₁ st c₁'
st' c'').

Hint Constructors pe_com.
```

Examples

```
Ltac step i :=
  (eapply i; intuition eauto; try solve_by_invert);
  repeat (try eapply PE_Seq;
          try (eapply PE_AssStatic; simpl; reflexivity);
```

```
          try (eapply PE_AssDynamic;
               [ simpl; reflexivity
               | intuition eauto; solve_by_invert])).

Definition square_loop: com :=
  WHILE 1 ≤ X DO
    Y ::= Y * Y;;
    X ::= X - 1
  END.

Example pe_loop_example1:
  square_loop / []
  \\ (WHILE 1 ≤ X DO
         (Y ::= Y * Y;;
          X ::= X - 1);; SKIP
       END) / [] / SKIP.
Proof. erewrite f_equal2 with (f := fun c st ⇒ _ / _ \\ c / st /
SKIP).
  step PE_WhileFixed. step PE_WhileFixedEnd. reflexivity.
  reflexivity. reflexivity. Qed.

Example pe_loop_example2:
  (X ::= 3;; square_loop) / []
  \\ (SKIP;;
       (Y ::= Y * Y;; SKIP);;
       (Y ::= Y * Y;; SKIP);;
       (Y ::= Y * Y;; SKIP);;
       SKIP) / [(X,0)] / SKIP.
Proof. erewrite f_equal2 with (f := fun c st ⇒ _ / _ \\ c / st /
SKIP).
  eapply PE_Seq. eapply PE_AssStatic. reflexivity.
  step PE_WhileTrue.
  step PE_WhileTrue.
  step PE_WhileTrue.
  step PE_WhileFalse.
  inversion H. inversion H. inversion H.
  reflexivity. reflexivity. Qed.

Example pe_loop_example3:
  (Z ::= 3;; subtract_slowly) / []
  \\ (SKIP;;
       IFB !(X = 0) THEN
         (SKIP;; X ::= X - 1);;
         IFB !(X = 0) THEN
           (SKIP;; X ::= X - 1);;
           IFB !(X = 0) THEN
             (SKIP;; X ::= X - 1);;
             WHILE !(X = 0) DO
               (SKIP;; X ::= X - 1);; SKIP
             END;;
             SKIP;; Z ::= 0
           ELSE SKIP;; Z ::= 1 FI;; SKIP
         ELSE SKIP;; Z ::= 2 FI;; SKIP
       ELSE SKIP;; Z ::= 3 FI) / [] / SKIP.
Proof. erewrite f_equal2 with (f := fun c st ⇒ _ / _ \\ c / st /
SKIP).
  eapply PE_Seq. eapply PE_AssStatic. reflexivity.
```

```
    step PE_While.
    step PE_While.
    step PE_While.
    step PE_WhileFixed.
    step PE_WhileFixedEnd.
    reflexivity. inversion H. inversion H. inversion H.
    reflexivity. reflexivity. Qed.

Example pe_loop_example4:
  (X ::= 0;;
   WHILE X ≤ 2 DO
     X ::= 1 - X
   END) / [] \\ (SKIP;; WHILE true DO SKIP END) / [(X,0)] /
SKIP.
Proof. erewrite f_equal2 with (f := fun c st ⇒ _ / _ \\ c / st /
SKIP).
  eapply PE_Seq. eapply PE_AssStatic. reflexivity.
  step PE_WhileFixedLoop.
  step PE_WhileTrue.
  step PE_WhileFixedEnd.
  inversion H. reflexivity. reflexivity. reflexivity. Qed.
```

Correctness

Because this partial evaluator can unroll a loop n-fold where n is a (finite) integer greater than one, in order to show it correct we need to perform induction not structurally on dynamic evaluation but on the number of times dynamic evaluation enters a loop body.

```
Reserved Notation "c1 '/' st '\\' st' '#' n"
  (at level 40, st at level 39, st' at level 39).

Inductive ceval_count : com → state → state → nat → Prop :=
  | E'Skip : ∀ st,
      SKIP / st \\ st # 0
  | E'Ass : ∀ st a1 n l,
      aeval st a1 = n →
      (l ::= a1) / st \\ (t_update st l n) # 0
  | E'Seq : ∀ c1 c2 st st' st'' n1 n2,
      c1 / st \\ st' # n1 →
      c2 / st' \\ st'' # n2 →
      (c1 ;; c2) / st \\ st'' # (n1 + n2)
  | E'IfTrue : ∀ st st' b1 c1 c2 n,
      beval st b1 = true →
      c1 / st \\ st' # n →
      (IFB b1 THEN c1 ELSE c2 FI) / st \\ st' # n
  | E'IfFalse : ∀ st st' b1 c1 c2 n,
      beval st b1 = false →
      c2 / st \\ st' # n →
      (IFB b1 THEN c1 ELSE c2 FI) / st \\ st' # n
  | E'WhileFalse : ∀ b1 st c1,
      beval st b1 = false →
```

```
      (WHILE b1 DO c1 END) / st \\ st # 0
  | E'WhileTrue : ∀ st st' st'' b1 c1 n1 n2,
      beval st b1 = true →
      c1 / st \\ st' # n1 →
      (WHILE b1 DO c1 END) / st' \\ st'' # n2 →
      (WHILE b1 DO c1 END) / st \\ st'' # S (n1 + n2)

  where "c1 '/' st '\\' st' # n" := (ceval_count c1 st st' n).

Hint Constructors ceval_count.

Theorem ceval_count_complete: ∀ c st st',
  c / st \\ st' → ∃ n, c / st \\ st' # n.
Proof. intros c st st' Heval.
  induction Heval;
    try inversion IHHeval1;
    try inversion IHHeval2;
    try inversion IHHeval;
    eauto. Qed.

Theorem ceval_count_sound: ∀ c st st' n,
  c / st \\ st' # n → c / st \\ st'.
Proof. intros c st st' n Heval. induction Heval; eauto. Qed.

Theorem pe_compare_nil_lookup: ∀ pe_st1 pe_st2,
  pe_compare pe_st1 pe_st2 = [] →
  ∀ V, pe_lookup pe_st1 V = pe_lookup pe_st2 V.
Proof. intros pe_st1 pe_st2 H V.
  apply (pe_compare_correct pe_st1 pe_st2 V).
  rewrite H. intro. inversion H0. Qed.

Theorem pe_compare_nil_update: ∀ pe_st1 pe_st2,
  pe_compare pe_st1 pe_st2 = [] →
  ∀ st, pe_update st pe_st1 = pe_update st pe_st2.
Proof. intros pe_st1 pe_st2 H st.
  apply functional_extensionality. intros V.
  rewrite !pe_update_correct.
  apply pe_compare_nil_lookup with (V:=V) in H.
  rewrite H. reflexivity. Qed.

Reserved Notation "c' '/' pe_st' '/' c'' '/' st '\\' st'' '#' n"
  (at level 40, pe_st' at level 39, c'' at level 39,
   st at level 39, st'' at level 39).

Close Scope bexp_scope.

Inductive pe_ceval_count (c':com) (pe_st':pe_state) (c'':com)
                         (st:state) (st'':state) (n:nat) : Prop
:=
  | pe_ceval_count_intro : ∀ st' n',
    c' / st \\ st' →
    c'' / pe_update st' pe_st' \\ st'' # n' →
    n' ≤ n →
    c' / pe_st' / c'' / st \\ st'' # n
```

```
  where "c' '/' pe_st' '/' c'' '/' st '\\' st'' '#' n" :=
        (pe_ceval_count c' pe_st' c'' st st'' n).

Hint Constructors pe_ceval_count.

Lemma pe_ceval_count_le: ∀ c' pe_st' c'' st st'' n n',
  n' ≤ n →
  c' / pe_st' / c'' / st \\ st'' # n' →
  c' / pe_st' / c'' / st \\ st'' # n.
Proof. intros c' pe_st' c'' st st'' n n' Hle H. inversion H.
  econstructor; try eassumption. omega. Qed.

Theorem pe_com_complete:
  ∀ c pe_st pe_st' c' c'', c / pe_st \\ c' / pe_st' / c'' →
  ∀ st st'' n,
  (c / pe_update st pe_st \\ st'' # n) →
  (c' / pe_st' / c'' / st \\ st'' # n).
Proof. intros c pe_st pe_st' c' c'' Hpe.
  induction Hpe; intros st st'' n Heval;
  try (inversion Heval; subst;
       try (rewrite → pe_bexp_correct, → H in *;
solve_by_invert);
       []);
  eauto.
  - (* PE_AssStatic *) econstructor. econstructor.
    rewrite → pe_aexp_correct. rewrite <- pe_update_update_add.
    rewrite → H. apply E'Skip. auto.
  - (* PE_AssDynamic *) econstructor. econstructor. reflexivity.
    rewrite → pe_aexp_correct. rewrite <-
pe_update_update_remove.
    apply E'Skip. auto.
  - (* PE_Seq *)
    edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
    inversion Hskip. subst.
    edestruct IHHpe2. eassumption.
    econstructor; eauto. omega.
  - (* PE_If *) inversion Heval; subst.
    + (* E'IfTrue *) edestruct IHHpe1. eassumption.
      econstructor. apply E_IfTrue. rewrite <- pe_bexp_correct.
assumption.
      eapply E_Seq. eassumption. apply eval_assign.
      rewrite <- assign_removes. eassumption. eassumption.
    + (* E_IfFalse *) edestruct IHHpe2. eassumption.
      econstructor. apply E_IfFalse. rewrite <- pe_bexp_correct.
assumption.
      eapply E_Seq. eassumption. apply eval_assign.
      rewrite → pe_compare_update.
      rewrite <- assign_removes. eassumption. eassumption.
  - (* PE_WhileTrue *)
    edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
    inversion Hskip. subst.
    edestruct IHHpe2. eassumption.
    econstructor; eauto. omega.
  - (* PE_While *) inversion Heval; subst.
    + (* E_WhileFalse *) econstructor. apply E_IfFalse.
      rewrite <- pe_bexp_correct. assumption.
      apply eval_assign.
```

```
      rewrite <- assign_removes. inversion H2; subst; auto.
      auto.
    + (* E_WhileTrue *)
      edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
      inversion Hskip. subst.
      edestruct IHHpe2. eassumption.
      econstructor. apply E_IfTrue.
      rewrite <- pe_bexp_correct. assumption.
      repeat eapply E_Seq; eauto. apply eval_assign.
      rewrite → pe_compare_update, <- assign_removes. eassumption.
      omega.
  - (* PE_WhileFixedLoop *) exfalso.
    generalize dependent (S (n1 + n2)). intros n.
    clear - H H0 IHHpe1 IHHpe2. generalize dependent st.
    induction n using lt_wf_ind; intros st Heval. inversion Heval; subst.
    + (* E'WhileFalse *) rewrite pe_bexp_correct, H in H7. inversion H7.
    + (* E'WhileTrue *)
      edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
      inversion Hskip. subst.
      edestruct IHHpe2. eassumption.
      rewrite <- (pe_compare_nil_update _ _ H0) in H7.
      apply H1 in H7; [| omega]. inversion H7.
  - (* PE_WhileFixed *) generalize dependent st.
    induction n using lt_wf_ind; intros st Heval. inversion Heval; subst.
    + (* E'WhileFalse *) rewrite pe_bexp_correct in H8. eauto.
    + (* E'WhileTrue *) rewrite pe_bexp_correct in H5.
      edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
      inversion Hskip. subst.
      edestruct IHHpe2. eassumption.
      rewrite <- (pe_compare_nil_update _ _ H1) in H8.
      apply H2 in H8; [| omega]. inversion H8.
      econstructor; [ eapply E_WhileTrue; eauto | eassumption | omega].
Qed.

Theorem pe_com_sound:
  ∀ c pe_st pe_st' c' c'', c / pe_st \\ c' / pe_st' / c'' →
  ∀ st st'' n,
  (c' / pe_st' / c'' / st \\ st'' # n) →
  (c / pe_update st pe_st \\ st'').
Proof. intros c pe_st pe_st' c' c'' Hpe.
  induction Hpe;
    intros st st'' n [st' n' Heval Heval' Hle];
    try (inversion Heval; []; subst);
    try (inversion Heval'; []; subst); eauto.
  - (* PE_AssStatic *) rewrite <- pe_update_update_add. apply E_Ass.
    rewrite → pe_aexp_correct. rewrite → H. reflexivity.
  - (* PE_AssDynamic *) rewrite <- pe_update_update_remove. apply E_Ass.
    rewrite <- pe_aexp_correct. reflexivity.
```

```
    - (* PE_Seq *) eapply E_Seq; eauto.
    - (* PE_IfTrue *) apply E_IfTrue.
      rewrite → pe_bexp_correct. rewrite → H. reflexivity.
      eapply IHHpe. eauto.
    - (* PE_IfFalse *) apply E_IfFalse.
      rewrite → pe_bexp_correct. rewrite → H. reflexivity.
      eapply IHHpe. eauto.
    - (* PE_If *) inversion Heval; subst; inversion H7; subst;
clear H7.
      + (* E_IfTrue *)
        eapply ceval_deterministic in H8; [| apply eval_assign].
subst.
        rewrite <- assign_removes in Heval'.
        apply E_IfTrue. rewrite → pe_bexp_correct. assumption.
        eapply IHHpe1. eauto.
      + (* E_IfFalse *)
        eapply ceval_deterministic in H8; [| apply eval_assign].
subst.
        rewrite → pe_compare_update in Heval'.
        rewrite <- assign_removes in Heval'.
        apply E_IfFalse. rewrite → pe_bexp_correct. assumption.
        eapply IHHpe2. eauto.
    - (* PE_WhileFalse *) apply E_WhileFalse.
      rewrite → pe_bexp_correct. rewrite → H. reflexivity.
    - (* PE_WhileTrue *) eapply E_WhileTrue.
      rewrite → pe_bexp_correct. rewrite → H. reflexivity.
      eapply IHHpe1. eauto. eapply IHHpe2. eauto.
    - (* PE_While *) inversion Heval; subst.
      + (* E_IfTrue *)
        inversion H9. subst. clear H9.
        inversion H10. subst. clear H10.
        eapply ceval_deterministic in H11; [| apply eval_assign].
subst.
        rewrite → pe_compare_update in Heval'.
        rewrite <- assign_removes in Heval'.
        eapply E_WhileTrue. rewrite → pe_bexp_correct. assumption.
        eapply IHHpe1. eauto.
        eapply IHHpe2. eauto.
      + (* E_IfFalse *) apply ceval_count_sound in Heval'.
        eapply ceval_deterministic in H9; [| apply eval_assign].
subst.
        rewrite <- assign_removes in Heval'.
        inversion H2; subst.
        * (* c2'' = SKIP *) inversion Heval'. subst. apply
E_WhileFalse.
          rewrite → pe_bexp_correct. assumption.
        * (* c2'' = WHILE b1 DO c1 END *) assumption.
    - (* PE_WhileFixedEnd *) eapply ceval_count_sound. apply
Heval'.
    - (* PE_WhileFixedLoop *)
      apply loop_never_stops in Heval. inversion Heval.
    - (* PE_WhileFixed *)
      clear - H1 IHHpe1 IHHpe2 Heval.
      remember (WHILE pe_bexp pe_st b1 DO c1';; c2' END) as c'.
```

```
      induction Heval;
        inversion Heqc'; subst; clear Heqc'.
      + (* E_WhileFalse *) apply E_WhileFalse.
        rewrite pe_bexp_correct. assumption.
      + (* E_WhileTrue *)
        assert (IHHeval2' := IHHeval2 (refl_equal _)).
        apply ceval_count_complete in IHHeval2'. inversion
IHHeval2'.
        clear IHHeval1 IHHeval2 IHHeval2'.
        inversion Heval1. subst.
        eapply E_WhileTrue. rewrite pe_bexp_correct. assumption.
eauto.
        eapply IHHpe2. econstructor. eassumption.
        rewrite <- (pe_compare_nil_update _ _ H1). eassumption.
apply le_n.
Qed.

Corollary pe_com_correct:
  ∀ c pe_st pe_st' c', c / pe_st \\ c' / pe_st' / SKIP →
  ∀ st st'',
  (c / pe_update st pe_st \\ st'') ↔
  (∃ st', c' / st \\ st' ∧ pe_update st' pe_st' = st'').
Proof. intros c pe_st pe_st' c' H st st''. split.
  - (* -> *) intros Heval.
    apply ceval_count_complete in Heval. inversion Heval as [n
Heval'].
    apply pe_com_complete with (st:=st) (st'':=st'') (n:=n) in
H.
    inversion H as [? ? ? Hskip ?]. inversion Hskip. subst.
eauto.
    assumption.
  - (* <- *) intros [st' [Heval Heq]]. subst st''.
    eapply pe_com_sound in H. apply H.
    econstructor. apply Heval. apply E'Skip. apply le_n.
Qed.

End Loop.
```

Partial Evaluation of Flowchart Programs

Instead of partially evaluating `WHILE` loops directly, the standard approach to partially evaluating imperative programs is to convert them into *flowcharts*. In other words, it turns out that adding labels and jumps to our language makes it much easier to partially evaluate. The result of partially evaluating a flowchart is a residual flowchart. If we are lucky, the jumps in the residual flowchart can be converted back to `WHILE` loops, but that is not possible in general; we do not pursue it here.

Basic blocks

A flowchart is made of *basic blocks*, which we represent with the inductive type `block`. A basic block is a sequence of assignments (the constructor `Assign`), concluding with a conditional jump (the constructor `If`) or an unconditional jump (the constructor `Goto`). The destinations of the jumps are specified by *labels*, which can be of any type. Therefore, we parameterize the `block` type by the type of labels.

```
Inductive block (Label:Type) : Type :=
  | Goto : Label → block Label
  | If : bexp → Label → Label → block Label
  | Assign : string → aexp → block Label → block Label.

Arguments Goto {Label} _.
Arguments If {Label} _ _ _.
Arguments Assign {Label} _ _ _.
```

We use the "even or odd" program, expressed above in Imp, as our running example. Converting this program into a flowchart turns out to require 4 labels, so we define the following type.

```
Inductive parity_label : Type :=
  | entry : parity_label
  | loop : parity_label
  | body : parity_label
  | done : parity_label.
```

The following `block` is the basic block found at the `body` label of the example program.

```
Definition parity_body : block parity_label :=
  Assign Y (Y - 1)
   (Assign X (1 - X)
     (Goto loop)).
```

To evaluate a basic block, given an initial state, is to compute the final state and the label to jump to next. Because basic blocks do not *contain* loops or other control structures, evaluation of basic blocks is a total function — we don't need to worry about non-termination.

```
Fixpoint keval {L:Type} (st:state) (k : block L) : state * L :=
  match k with
  | Goto l ⇒ (st, l)
  | If b l₁ l₂ ⇒ (st, if beval st b then l₁ else l₂)
  | Assign i a k ⇒ keval (t_update st i (aeval st a)) k
  end.

Example keval_example:
  keval { --> 0 } parity_body
  = ({ Y --> 0 ; X --> 1 }, loop).
```

+

Flowchart programs

A flowchart program is simply a lookup function that maps labels to basic blocks. Actually, some labels are *halting states* and do not map to any basic block. So, more precisely, a flowchart `program` whose labels are of type `L` is a function from `L` to `option (block L)`.

```
Definition program (L:Type) : Type := L → option (block L).

Definition parity : program parity_label := fun l ⇒
  match l with
  | entry ⇒ Some (Assign X 0 (Goto loop))
  | loop ⇒ Some (If (1 ≤ Y) body done)
  | body ⇒ Some parity_body
  | done ⇒ None (* halt *)
  end.
```

Unlike a basic block, a program may not terminate, so we model the evaluation of programs by an inductive relation `peval` rather than a recursive function.

```
Inductive peval {L:Type} (p : program L)
  : state → L → state → L → Prop :=
  | E_None: ∀ st l,
    p l = None →
    peval p st l st l
  | E_Some: ∀ st l k st' l' st'' l'',
    p l = Some k →
    keval st k = (st', l') →
    peval p st' l' st'' l'' →
    peval p st l st'' l''.

Example parity_eval: peval parity { --> 0 } entry { --> 0 } done.
Proof. erewrite f_equal with (f := fun st ⇒ peval _ _ _ st _).
  eapply E_Some. reflexivity. reflexivity.
  eapply E_Some. reflexivity. reflexivity.
  apply E_None. reflexivity.
  apply functional_extensionality. intros i. rewrite
t_update_same; auto.
Qed.
```

Partial Evaluation of Basic Blocks and Flowchart Programs

Partial evaluation changes the label type in a systematic way: if the label type used to be `L`, it becomes `pe_state * L`. So the same label in the original program may be unfolded, or blown up, into multiple labels by being paired with different partial states. For example, the label `loop` in the `parity` program will become two labels: `([(X,0)], loop)` and `([(X,1)], loop)`. This change of label type is reflected in the types of `pe_block` and `pe_program` defined presently.

```
Fixpoint pe_block {L:Type} (pe_st:pe_state) (k : block L)
  : block (pe_state * L) :=
  match k with
  | Goto l ⇒ Goto (pe_st, l)
  | If b l1 l2 ⇒
    match pe_bexp pe_st b with
    | BTrue ⇒ Goto (pe_st, l1)
    | BFalse ⇒ Goto (pe_st, l2)
    | b' ⇒ If b' (pe_st, l1) (pe_st, l2)
    end
  | Assign i a k ⇒
    match pe_aexp pe_st a with
    | ANum n ⇒ pe_block (pe_add pe_st i n) k
```

```
    | a' ⇒ Assign i a' (pe_block (pe_remove pe_st i) k)
    end
  end.

Example pe_block_example:
  pe_block [(X,0)] parity_body
  = Assign Y (Y - 1) (Goto ([(X,1)], loop)).
 +

Theorem pe_block_correct: ∀ (L:Type) st pe_st k st' pe_st'
(l':L),
  keval st (pe_block pe_st k) = (st', (pe_st', l')) →
  keval (pe_update st pe_st) k = (pe_update st' pe_st', l').
Proof. intros. generalize dependent pe_st. generalize dependent
st.
  induction k as [l | b l1 l2 | i a k];
    intros st pe_st H.
  - (* Goto *) inversion H; reflexivity.
  - (* If *)
    replace (keval st (pe_block pe_st (If b l1 l2)))
       with (keval st (If (pe_bexp pe_st b) (pe_st, l1) (pe_st,
l2)))
       in H by (simpl; destruct (pe_bexp pe_st b); reflexivity).
    simpl in *. rewrite pe_bexp_correct.
    destruct (beval st (pe_bexp pe_st b)); inversion H;
reflexivity.
  - (* Assign *)
    simpl in *. rewrite pe_aexp_correct.
    destruct (pe_aexp pe_st a); simpl;
      try solve [rewrite pe_update_update_add; apply IHk; apply
H];
      solve [rewrite pe_update_update_remove; apply IHk; apply
H].
Qed.

Definition pe_program {L:Type} (p : program L)
  : program (pe_state * L) :=
  fun pe_l ⇒ match pe_l with | (pe_st, l) ⇒
                option_map (pe_block pe_st) (p l)
              end.

Inductive pe_peval {L:Type} (p : program L)
  (st:state) (pe_st:pe_state) (l:L) (st'o:state) (l':L) : Prop
:=
  | pe_peval_intro : ∀ st' pe_st',
    peval (pe_program p) st (pe_st, l) st' (pe_st', l') →
    pe_update st' pe_st' = st'o →
    pe_peval p st pe_st l st'o l'.

Theorem pe_program_correct:
  ∀ (L:Type) (p : program L) st pe_st l st'o l',
  peval p (pe_update st pe_st) l st'o l' ↔
  pe_peval p st pe_st l st'o l'.
Proof. intros.
  split.
  - (* -> *) intros Heval.
    remember (pe_update st pe_st) as sto.
```

```
    generalize dependent pe_st. generalize dependent st.
    induction Heval as
      [ sto l Hlookup | sto l k st'o l' st''o l'' Hlookup Hkeval
Heval ];
      intros st pe_st Heqsto; subst sto.
    + (* E_None *) eapply pe_peval_intro. apply E_None.
      simpl. rewrite Hlookup. reflexivity. reflexivity.
    + (* E_Some *)
      remember (keval st (pe_block pe_st k)) as x.
      destruct x as [st' [pe_st' l'_]].
      symmetry in Heqx. erewrite pe_block_correct in Hkeval by
apply Heqx.
      inversion Hkeval. subst st'o l'_. clear Hkeval.
      edestruct IHHeval. reflexivity. subst st''o. clear
IHHeval.
      eapply pe_peval_intro; [| reflexivity]. eapply E_Some;
eauto.
      simpl. rewrite Hlookup. reflexivity.
  - (* <- *) intros [st' pe_st' Heval Heqst'o].
    remember (pe_st, l) as pe_st_l.
    remember (pe_st', l') as pe_st'_l'.
    generalize dependent pe_st. generalize dependent l.
    induction Heval as
      [ st [pe_st_ l_] Hlookup
      | st [pe_st_ l_] pe_k st' [pe_st'_ l'_] st'' [pe_st'' l'']
        Hlookup Hkeval Heval ];
      intros l pe_st Heqpe_st_l;
      inversion Heqpe_st_l; inversion Heqpe_st'_l'; repeat
subst.
    + (* E_None *) apply E_None. simpl in Hlookup.
      destruct (p l'); [ solve [ inversion Hlookup ] |
reflexivity ].
    + (* E_Some *)
      simpl in Hlookup. remember (p l) as k.
      destruct k as [k|]; inversion Hlookup; subst.
      eapply E_Some; eauto. apply pe_block_correct. apply
Hkeval.
Qed.
```