Require Import Bool.
Require Import ZArith.
Require Import IMPSyntax.
Require Import IMPSemantics.

(* ** Pseudo Denotational Semantics *)

(* Here we’re going to explore what’s called "denoting" *)
(* When we take a program and denote it, we simply give the meaning of the program *)
(* in terms of something else *)

(* Here, we’ll use the existing meaning of Coq to denote our programs *)

(* If we have a binary operation, the meaning of that binary operation is a *)
(* coq function of type Z → Z → Z *)
Definition denote_binop (op: binop) : Z → Z → Z := exec_op op.

(* When we denote an expression, the meaning of it depends on the current heap *)
(* Thus, the coq type of a denoted expression is heap → Z *)
Fixpoint denote_expr (e: expr) : heap → Z :=
match e with
| Int i => fun _ h => i
| Var v => fun h => h v
| BinOp op e1 e2 => let f := denote_binop op in
  let x := denote_expr e1 in
  let y := denote_expr e2 in
  fun h => f (x h) (y h)
end.

(* Let’s play with denoting a few toy examples *)
Eval cbv in (denote_expr ("x" [+] "y")) empty.
Eval cbv in (denote_expr ("x" [+] 1)) empty.
Eval cbv in (denote_expr ("x" [+] "y")) empty.

(* Note that for expressions, essentially the only difference between denoting and interpreting is *)
(* the stage at which the heap matters. When interpreting a program, we start with a *)
(* heap and expression, and crawl over the expression tree with both. *)
(* When denoting an expression, we crawl over the entire expression without the heap, giving meaning to the expression for all heaps *)
(* Only afterwards do we derive meaning by providing a particular heap *)

(* Here we can prove that we denoted expressions correctly. *)
(* We want the meaning to match up in all cases *)
Lemma denote_expr_interp_expr: forall e h i, (interp_expr e h) i = (interp_expr e h) i.
Proof.
  intros. rewrite interp_expr_interp_expr.
  split.
  - apply interp_expr_eval.
  - apply eval_interp_expr.
Qed.

(* Helpful little function *)
(* Stands for "option bind" *)
(* For more info as to why it’s named that, see documentation about monads *)
Definition obind {A B: Type} (oa: option A) (f: A → option B) : option B :=
match oa with
| None => None
| Some a => f a
end.

(* Here we give meaning to statements *)
(* Note that this has type nat → heap → option heap *)
(* instead of simply heap → heap *)
(* the nat will encode the amount of fuel we give to the program *)
(* as the program could diverge *)
(* the option encodes the fact that evaluation could fail *)
(* though the only way to fail is running out of fuel *)
(* careful to detect timeout (running out of fuel)! *)
Fixpoint denote_stmt (s: stmt) : nat → heap → option heap :=
match s with
| Nop => fun _ h => Some h
| Assign v e => let de := denote_expr e in
  fun _ h => Some (update h v (de h))
| Seq e1 e2 => let d1 := denote_stmt e1 in
  let d2 := denote_stmt e2 in
  fun n h => obind (d1 n h) (d2 n h)
| Cond e s1 s2 => let de := denote_expr e in
  let ds := denote_stmt s1 in
  let ds := denote_stmt s2 in
  fun n h =>
    if 2 eq_dec 0 (de h) then
      Some h
    else
ds n h
| While e s => let de := denote_expr e in
  let ds := denote_stmt s in
  fix loop n h :=
  match n with
  | 0 => None
  | S m =>
    if 2 eq_dec 0 (de h) then
      Some h
    else
      obind (ds n h) (loop m)
  end.
end.

Theorem nat_strong_ind’ : forall P : nat → Prop, P 0nat →
(forall n, (forall m, (m <= n)%nat → P m) → P (S n)) →
forall n, (forall m, (m <= n)%nat → P m) → P n.
Proof.
  intros. unfold n. intros.
  - assert (m = 0%nat) by omega. subst.
- assert \((m \leq n) \rightarrow m = S n\) by omega.
  intuition. subst. auto.
Qed.

Lemma nat_strong_ind :
  forall (P : nat -> Prop),
  P 0%nat ->
  (forall n, (forall m, (m \leq n) -> P m) -> P (S n)) ->
  forall n, P n.
Proof.
  intros.
  eapply nat_strong_ind'; eauto.
Qed.

(** Here’s what we might use for a different kind of induction on nats *)
(** If we wanted to do different induction like we talked about in class *)
(** This is what we might do *)
Lemma nat_parity_ind :
 forall (P : nat -> Prop),
  P 0%nat ->
  P 1%nat ->
  (forall n, P n -> P (S (S n))) ->
  forall n, P n.
Proof.
  induction n using nat_strong_ind; intros.
eauto.
destruct n. eauto.
eapply H1. eapply H2.
  omega.
Qed.