Require Import ZArith.
Require Import String.
Open Scope string_scope.
Open Scope Z_scope.

Require Import L05_syntax.

(* ** expr : big step *)

(* Heaps :
To evaluate an expression containing variables, we need some representation of memory to get the value of variables from.

We need to model memory as some mapping from variables to ints. Functions can do just that! *)

Definition heap : Type := string → Z.

(* The empty memory just maps everything to zero. *)

Definition empty : heap := fun v => 0.

(* We will also need to evaluate our operators over ints. Since there’s a bunch, we’ll define a helper function for this. *)

Definition exec_op (op: binop) (i1 i2: Z) : Z :=
match op with
| Add => i1 + i2  | Sub => i1 − i2  | Mul => i1 * i2  | Div => i1 / i2  | Mod => i1 mod i2  | Lt  => if Z_lt_dec i1 i2 then 0 else 1  | Disj => if Z_eq_dec i1 0 then              if Z_eq_dec i2 0 then 0 else 1            else 1  end.

(* SearchAbout Z. *)

(* Now we can define a relation to capture the semantics of expressions *)

Inductive eval : heap → expr → Z → Prop :=
| eval_int:    forall h i, eval h (Int i) i| eval_var:    forall h v, eval h (Var v) (h v)  | eval_binop:    forall h op e1 e2 l1 l2 i1 i2, eval h (BinOp op e1 e2) i3.

Lemma eval_ex1:
 eval empty ("x" [+ ] 1) 1.
Proof.
 apply eval_binop with (i1 := 0)
 | (i2 := 1).
 | apply eval_var.
 | apply eval_int.
 | simpl. reflexivity.
Qed.

Lemma eval_ex2:
 ~ eval empty ("x" [+ ] 1) 2.
Proof.
 unfold not.
 intros.
 inversion H.
 subst.
 inversion H4.
 unfold empty in H1.
 subst.
 inversion H6.
 subst.
 simpl in H7.
 discriminate.
Qed.

(* We can also define an interpreter for expressions. *)

Fixpoint interp_expr (h: heap) (e: expr) : Z :=
match e with
| Int i => i  | Var v => h v  | BinOp op e1 e2 =>
 | exec_op op (interp_expr h e1) (interp_expr h e2)  end.

Eval cbv in (interp_expr empty ("x" [+ ] 1)).

(* ... and prove relational and functional versions agree *)

Lemma interp_expr_ok:
forall h e i, interp_expr h e = i −> eval h e i.
Proof.
 induction e; simpl goal and context in all subgoals;
 (* simpl in *. )
 | subst. (* replace z with i everywhere *)
 | constructor. (* 'apply eval_int' would also work here *)
 | subst. (* 'apply eval_int' would also work here *)
 | constructor. (* 'apply eval_binop' won't work, even though it seems like it should unify. Coq complains: << Error: Unable to find an instance for the variables l1, l2. >> because it needs to know those to apply the constructor. We can use a variant of apply to tell Coq exactly what l1 and l2 should be. *)
 | apply eval_binop with (i1 := interp_expr h e1)
Lemma interp_expr_ok:  
forall h e i,  
interp_expr h e = i ->  
eval h e i.  
Proof.  
intros h e.  
induction e; simpl in *; intros.  
- subst; constructor.  
- subst; constructor.  
- (** OK, now IHe1 and IHe2 look stronger *)  
  apply eval_binop with (i1 := interp_expr h e1)  
  (i2 := interp_expr h e2).  
  + apply IHe1. auto.  
  + apply IHe2. auto.  
  + assumption.  
Qed.

(** 'interp_expr_ok' only shows that if the interpreter produces 'i' as the result of evaluating expr 'e' in heap 'h', then eval relates 'h', 'e', and 'i' as well. We can prove the other direction: if the eval relates 'h', 'e', and 'i', then the interpreter will produce 'i' as the result of evaluation expr 'e' in heap 'h'. *)

Lemma eval_interp:  
forall h e i,  
eval h e i ->  
interp_expr h e = i.  
Proof.  
intros h e.  
induction e; simpl in *; intros.  
- subst; constructor.  
- subst; constructor.  
- (** 'constructor.' will not work here because the goal does not unify with the eval_binop case. 'econstructor' is a more flexible version of constructor that introduces existentials that will allow things to unify behind the scenes. Check it out! *) 
econstructor.  
+ (** 'assumption.' will not work here because our goal has an existential in it. However 'eassumption' knows how to handle it! *) 
eassumption.  
+ eassumption.  
+ reflexivity.  
Qed.

(** Notice how much cleaner that was! *)

(** we can also write the one of the earlier lemmas in a slightly cleaner way *)

Lemma interp_eval:  
forall h e,  
eval h e (interp_expr h e).  
Proof.  
intros.  
induction e; simpl.  
- constructor.  
- constructor.  
- (** 'constructor.' will not work here because the goal does not unify with the eval_binop case. 'econstructor' is a more flexible version of constructor that introduces existentials that will allow things to unify behind the scenes. Check it out! *) 
econstructor.  
+ (** 'assumption.' will not work here because our goal has an existential in it. However 'eassumption' knows how to handle it! *) 
eassumption.  
+ eassumption.  
+ reflexivity.  
Qed.

(** OK, so we've shown that our relational semantics for expr agrees with our functional interpreter. One nice consequence of this is that we can easily show that our eval relation is deterministic. *)

Lemma eval_det:  
forall h e i1 i2,  
eval h e i1 ->  
eval h e i2 ->  
i1 = i2.  
Proof.  
intros.  
apply eval_interp in H.  
apply eval_interp in H0.  
subst. reflexivity.  
Qed.

(** it's a bit more work without interp, but not too bad *)

Lemma eval_det':  
forall h e i1 i2,  
eval h e i1 i2 ->  
subst. reflexivity.  
Qed.

(** set up a strong induction hyp *)

Lemma eval_swap_add:  
forall h e1 e2 i,  
eval h (BinOp Add e1 e2) i <=> eval h (BinOp Add e2 e1) i.  
Proof.  
split; intros.
Lemma interp_expr_swap_add:
  forall h e1 e2, interp_expr h (BinOp Add e1 e2) = interp_expr h (BinOp Add e2 e1).
Proof.
  intros; simpl; omega.
Qed.

Lemma eval_add_zero:
  forall h e i, eval h (BinOp Add e (Int 0)) i <= eval h e i.
Proof.
  split; intros. 
  inversion H; subst. 
  inversion H6; subst. 
  simpl. omega. 
  (** cut *) 
  cut (i1 + 0 = i1). 
  + intro. rewrite H0. 
    assumption. 
  + omega. 
  (** replace lets us rewrite a subterm *) 
  replace (i1 + 0) with i1 by omega. 
  assumption. 
  - econstructor; eauto. 
  + econstructor; eauto. 
  + simpl. omega. 
Qed.

Lemma interp_expr_add_zero:
  forall h e, interp_expr h (BinOp Add e (Int 0)) = interp_expr h e.
Proof.
  intros; simpl. omega.
Qed.

Lemma eval_mul_zero:
  forall h e i, eval h (BinOp Mul e (Int 0)) i <= 0. 
Proof.
  split; intros. 
  inversion H; subst. 
  inversion H6; subst. 
  simpl. omega. 
  sub. 
  pose (interp_expr h e). 
  eapply eval_binop with (11 := z). 
  + eapply interp_expr_ok. auto. 
  + econstructor; eauto. 
  + simpl. omega. 
Qed.

Lemma interp_expr_mul_zero:
  forall h e, interp_expr h (BinOp Mul e (Int 0)) = 0. 
Proof.

(* stmt : small step *) 
To define the semantics for statements, 
we'll need to be able to update the heap. 

Definition update (h: heap) (v: string) (i: Z) : heap := 
  if string_dec v' v then 
    i 
  else 
    h v'.

Inductive step : heap -> stmt -> heap -> stmt -> Prop := 
  step_assign: 
    forall h v e i, eval h e i -> step h (Assign v e) (update h v i) Nop 
  step_seq_nop: 
    forall h s, step h (Seq Nop s) h s 
  step_while_false: 
    forall h e s i, eval h e i -> i <> 0 -> step h (While e s) h s 
  step_cond_true: 
    forall h e s i, eval h e i -> i = 0 -> step h (While e s) h Nop 
  step_cond_false: 
    forall h e s i, eval h e i -> i = 0 -> step h (While e s) h s 
  step_while: 
    forall h e s, step h (While e s) h (Seq s (While e s)) (*)
  (** note that there are several other ways 
  we could have done semantics for while *)
  (** We can also define an interpreter to run 
  a single step of a stmt, but we’ll have 
  to learn some new types to write it down. 
  Note that, unlike eval, the step relation is partial: not every heap and stmt is related to another heap and stmt! *)
Lemma step_partial: 
  exists h, exists s, 
  step
forall h' s', ~ step h s h' s'.
Proof.
exists empty.
exists Nop.
intros. unfold not. intros.
inversion H. (** impossible! *)
Qed.

(** In general, we say that any stmt that
cannot step is "stuck" *)
Definition ellis_stuck (h: heap) (s: stmt) : Prop :=
forall h' s',
~ step h s h' s'.
Definition stuck (s: stmt) : Prop :=
forall h h' s',
~ step h s h' s'.

Lemma nop_stuck:
stuck Nop.
Proof.
unfold stuck, not, intros.
inversion H.
Qed.

(** Since the step relation is partial, but all
functions have to be total, we will use the
'option' type to represent the results of
step interpreter. *)

Print option.

(** We could define our interpreter this way,
but we end up with a case explosion in
the Seq nop / non-nop cases... *)

(** Fixpoint interp_step (h: heap) (s: stmt) : option (heap * stmt) :=
match s with
| Nop => None
| Assign v e =>
  Some (update h v (interp_expr h e), Nop)
| Seq s1 s2 =>
  if isNop s1 then
    Some (h, s2)
  else
    match interp_step h s1
    with
    | Some (h', s1') => Some (h', Seq s1' s2)
    | None => None
    end
| Cond e s =>
  if Z_eq_dec (interp_expr h e) 0 then
    Some (h, Nop)
  else
    Some (h, s)
| While e s =>
  if Z_eq_dec (interp_expr h e) 0 then
    Some (h, Nop)
  else
    Some (h, Seq s (While e s))
end.

(** and we can prove that our step interpreter
agrees with our relational semantics *)
Lemma interp_step_ok:
forall h s h' s',
interp_step h s = Some (h', s') ->
step h s h' s'.
Proof.
intros h s. revert h.
induction s; simpl; intros.
- discriminate.
- inversion h. subst.
  constructor. apply interp_eval.
  destruct (isNop s1) eqn:?.
  (** use the weird 'eqn:? after a destruct
to remember what you destructed! *)
  rewrite isNop_ok in Heqb. subst.
  inversion H. subst.
  constructor. assumption.
  discriminate.
  destruct (2_eq_dec (interp_expr h e) 0) eqn:?.
  + rewrite 2_eq_dec_ok in H. subst.
  + inversion H. subst.
  (** Once again 'constructor' and even 'apply step_cond_false'
  will not work because the conclusion of the step_cond_false
  constructor needs to know what 'i' should be.

(** So instead, we’ll define a helper to simplify the match. *)
Check (forall s, s = Nop).

Definition isNop (s: stmt) : bool :=
match s with
| Nop => true
| _ => false
end.

Lemma isNop_ok:
forall s,
isNop s = true <-> s = Nop.
Proof.
(** a lot of times we don’t really need intros *)
destruct s; simpl; split; intros;
auto; discriminate.
Qed.
We could explicitly use the 'apply step_cond_false with (i := ...)' flavor of apply to specify 'i', but using 'econstructor' is more convenient and flexible. *)

(* now that we have these existentials in our context we have to use the 'e' versions of all our regular tactics. *)
eapply interp_eval. (** existential resolved! *)

assumption.
+ inversion H; subst.
eapply step_cond_true; eauto.

- (** while is pretty similar to cond *)
destruct (Z.eq_dec (interp_expr h e) 0) eqn:?.
  + inversion H; subst.
eapply step_while_false; eauto.
  eapply interp_eval; auto.
  + inversion H; subst.
eapply step_while_true; eauto.
eapply interp_eval; auto.

Qed.

Lemma step_interp_step:
  forall h s h' s',
  step h s h' s' ->
  interp_step h s = Some (h', s').
Proof.
  intros. induction H; simpl; auto.
  destruct (isNop s1) eqn:?.
  + apply isNop_ok in Heqb. subst.
    inversion H. (* nop can' step *)
  + rewrite IHstep. auto.
  eapply apply interp_eval in H. subst.
  destruct (Z.eq_dec (interp_expr h e) 0); auto.
  exfalso. unfold n in H. apply H0. assumption.
  eapply apply interp_eval in H. subst.
  destruct (Z.eq_dec (interp_expr h e) 0); auto.
  unfold not in n. apply n in H0.
  exfalso. assumption.
  eapply apply interp_eval in H. subst.
  destruct (Z.eq_dec (interp_expr h e) 0); auto.
  unfold not in n. eapply n in H0.
  eapply apply interp_eval in H. subst.
  destruct (Z.eq_dec (interp_expr h e) 0); auto.
  omega.

Qed.

(** So far, 'step' only does one "step" of an execution of a stmt. We can build the transitive closure of this relation though to reason about with more than one step. *)

Inductive step_n : heap -> stmt -> nat -> heap -> stmt -> Prop :=
  sn_refl:
  forall h s,
  step n h s 0 h s
  | sn_step:
    forall h1 s1 n h2 s2 h3 s3,
    step h1 s1 n h2 s2 h3 s3 ->
    step n h2 s2 n h3 s3
    | step_n h1 s1 (S n) h3 s3.

(** Defining an interpreter for more than one step is trickier! Since a stmt may not terminate, we can’t just naively write a recursive function to run a...**)
(**
Try using this definition and doing the above proofs.
*)

(** TAH DAH! We have a verified interpreter! *)