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.

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 % i2 | Lt => if Zlt_dec i1 i2 then 0 else 1 | Disj => if Zeq_dec i1 0 then
  if Zeq_dec i2 0 then 0 else 1
  else 1
end.

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.
  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.
  subst. (* replace z with i everywhere *)
  constructor. (* 'apply eval_int' would also work here *)
  subst. (* 'apply interp_expr' would also work here *)
  subst. (* 'apply interp_expr' won’t work, even though it seems like it should unify. Coq complains: *)
  apply eval_binop with (i1 := interp_expr h e1) (i2 := interp_expr h e2).

(* 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 i1 i2 i3,
  eval h (BinOp op e1 e2) i3.

Lemma eval_ex1:
eval empty (Int 1) i3.
Proof.
  apply eval_binop with (i1 := 0) (i2 := 1).
    - apply eval_var.
    - apply eval_int.
    - simpl. reflexivity.
Abort.

**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. (** careful not to intro too much *)

- induction e; simpl in *; intros.
  - (** inversion tells coq to let us do case analysis on all the ways H could have been produced *)
    inversion H.
  - (** we get a bunch of equalities in our context, subst will clean them up *)
    subst. reflexivity.
  - inversion H1; subst; reflexivity.
  - inversion H2; subst; reflexivity.
  - rewrite (IHe11 H4). (** we can "fill in" an equality to rewrite with *)
    rewrite (IHe22 H6). reflexivity.

Qed.

(** we actually could have proved the above lemma in an even cooler way: by doing induction on the derivation of eval! *)

**Lemma eval_interp':**

forall h e i, eval h e i -> interp_expr h e = i.

**Proof.**

- intros. induction H; simpl.
  - reflexivity.
  - subst. reflexivity.

Qed.

(** notice how much cleaner that was! *)
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.
  (** replace lets us rewrite a subterm *)
  replace (i1 + 0) with i1 by omega.
  assumption.
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 <-> i = 0.
Proof. split; intros.
  inversion H; subst.
  inversion H6; subst.
  simpl. omega.
  subst.
  pose (interp_expr h e).
  eapply eval_binop with (il := 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. intros; simpl. omega. Qed.
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 stuck (s: stmt) :=
forall h h' s',
~ step h s h' s'.

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

(** In general, we say that any stmt that
cannot step is "stuck" *)

Definition stuck' (s : stmt) :=
exists h,forall h' s',
~ step h s h' s'.

Lemma step_exists_heap :
forall h1 h1' h2 s s1',
step h1 s h1' s1' ->
exists h2' s2',
step h2 s h2' s2'.
Proof.
Admitted.

Theorem stuck'_stuck :
forall s, stuck' s -> stuck s.
Proof.
Admitted.

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

(** So instead, we’ll define a helper to simplify the match. *)

Definition isNop (s: stmt) : bool :=
mismatch 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.

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.
end.

(** and we can prove that our step interpreter
agrees with our relational semantics *)

Definition 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 s) eqn:?.
    (** use the weird 'eqn:? after a destruct
to remember what you destructed! *)
    + rewrite isNop_ok in Heqb. subst.
    inversion H. subst. constructor.
    + destruct (interp_step h s1) as [[foo bar]] eqn:?:.
      (** and you can control the names of parts of
      constructors using "destruct ... as ..." *)
      * inversion H. subst.
      apply lh1 in H0. subst.
      constructor. assumption.
    * discriminate.
    - destruct (Z.eq_dec (interp_expr h e) 0) eqn:?:.
      + inversion H. subst.
      eapply step_cond_true; eauto.      apply interp_eval; auto.
    - (*
      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.

(** 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_n h1 s1 n h2 s2 −>  step_n h2 s2 h3 s3 −>  step_n h1 s1 (S n) h3 s3.

Qed.

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

Fixpoint run (fuel: nat) (h: heap) (s: stmt) : option (heap * stmt) :=
  match fuel with
  | O => None  | S n =>      match interp_step h s
  with
    | Some (h', s') => run n h' s'
    | None => Some (h, s) (** why not None? *)
  end.

Lemma run_ok:
  forall fuel h s h' s',
  run fuel h s = Some (h', s') −>
  exists n, step_n h s n h' s'.
Proof.
  induction fuel; simpl; intros.
  - discriminate.
  - destruct (interp_step h s) as [[foo bar]] eqn:?:.
    eapply IHfuel in H.
    apply interp_step_ok in H0.
    destruct H. exists (S x).
    eapply step_n_left; eauto.
  + inversion H; subst.
  exists O. constructor; auto.
Qed.