Require Import ZArith.
Require Import String.

Open Scope string_scope.
Open Scope Z_scope.

(* SYNTAX *)

Inductive binop : Set :=
| Add | Sub | Mul | Div | Mod | Lt | Lte | Conj | Disj.

Inductive expr : Set :=

Coercion Int : Z >-> expr.
Coercion Var : string >-> expr.

Notation "X [+] Y" := (BinOp Add X Y) (at level 51, left associativity).
Notation "X [-] Y" := (BinOp Sub X Y) (at level 51, left associativity).
Notation "X [*] Y" := (BinOp Div X Y) (at level 50, left associativity).
Notation "X [/] Y" := (BinOp Div X Y) (at level 50, left associativity).
Notation "X [%] Y" := (BinOp Mod X Y) (at level 50, left associativity).
Notation "X [<] Y" := (BinOp Lt X Y) (at level 52).
Notation "X <<= Y" := (BinOp Left X Y) (at level 52).
Notation "X && Y" := (BinOp Conj X Y) (at level 53, left associativity).
Notation "X || Y" := (BinOp Disj X Y) (at level 54, left associativity).

Inductive stmt : Set :=

Definition fib_x_y : stmt :=
"y" <- 0;;
"y0" <- 1;;
"y1" <- 0;;
*i* <- 0;;
while ("i*" [=] "*x") {|
  "y*" <- "y0*" [+] "y1*";;
  "y0*" <- "y1*";;
  "y1*" <- "y*";;
  "i*" <- "i*" [+] 1 |
}.

Definition gcd_xy_i : stmt :=
"i*" <- "*x";;
while (0 [=] "a" [+] ["i*" [++] 0 [++] "y" [++] "*i*" [++] 1]) {
Lemma eval_ex1 :  
  eval empty ("x" [+] 1) 1.  
Proof.  
  apply eval_binop with (i1 := 0) (i2 := 1).  
  apply eval_var.  
  subst; apply eval_int.  
  reflexivity.  
Qed.  
(** As a sanity check, let's make sure eval *doesn't* hold where it shouldn't *)

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

(* ... 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.  
  intros h e.  
  induction e; simpl in *; intros.  
  subst; constructor.  
  subst; constructor.  
  subst.  
  subst. simpl in *.  
  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.  
  inversion H.  
  subst; reflexivity.  
  inversion H; subst; reflexivity.  
  inversion H; subst; rewrite (IHe1 i1 H4).  
  rewrite (IHe2 i2 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.  
  apply reflexivity.  
  subst. 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,  
  interp_expr h e.  
Proof.
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. 
  − econstructor; eauto. 
  + econstructor; eauto. 
  + simpl. omega. 
Qed.

Lemma interp_expr_add_zero: 
  forall h e, 
  interp_expr h (BinOp Add e (Int 0)) i <-> interp_expr h e. 
Proof. 
  intros; 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.

(* Huh, so why ever have relational semantics? *)
(* 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 :=
  fun v' =>
    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 s Nop) h Nop
  | step典型: forall h s1 s2 s' h',
    step h (Seq s1 s2) h s1 h' s' ->
    step h (Seq s1 s2) h' (Seq s1' s2)
  | step_cond_true: forall h e s i,
    eval h e i ->
    i <> 0 ->
    step h (Cond e s) h s
  | step_cond_false: forall h e s i,
    eval h e i ->
    i = 0 ->
    step h (Cond e s) h Nop
  | step_while_true: forall h e s i,
    eval h e i ->
    i <> 0 ->
    step h (While e s) h (Seq s (While e s))
  | step_while_false: forall h e s i,
    eval h e i ->
    i = 0 ->
    step h (While e s) h Nop.

(** 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, 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) : 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 =>
    if Z_eq_dec (interp_expr h e) 0 then
      Some (h, Nop)
    else
      Some (h, Seq s (While e s))
  end.

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

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

**Lemma** isNop_ok:

```coq
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) :=

```coq
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, Seq s1 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, Seq s (While e s))
end.
```

**Lemma** interp_step_ok:

```coq
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. eapply 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.
  eapply interp_eval; eauto.
- eapply step_cond_true; eauto.
- eapply step_while_true; eauto.
- eapply interp_eval; eauto.
Qed.
```

**Inductive** step_n : heap -> stmt -> nat -> heap -> stmt -> Prop :=

```coq
| sn_refl: forall h s, step_n h s O h s
| sn_step: forall h1 s1 n h2 s2 h3 s3, step_n h1 s1 n h2 s2 -> step h2 s2 h3 s3
-> step_n h1 s1 (S n) h3 s3.
```

**Lemma** step_n_left:

```coq
forall h1 s1 h2 s2 h3 s3, step_n h1 s1 n h2 s2 -> step h2 s2 h3 s3
-> step_n h1 s1 (S n) h3 s3.
Proof.
intros. induction H0.
- econstructor.
- eapply IHstep_n; eauto.
Qed.
```

**Fixpoint** run (fuel : nat) (h : heap) (s : stmt) : option (heap * stmt) :=

```coq
match fuel with
| O => None
| S n =>
  match interp_step h s
  | Some (h', s') => run n h' s'
  | None => Some (h, s)
  end.
```

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.

---

**Inductive** step_n : heap -> stmt -> nat -> heap -> stmt -> Prop :=

```coq
| sn_refl: forall h s, step_n h s O h s
| sn_step: forall h1 s1 n h2 s2 h3 s3, step_n h1 s1 n h2 s2
-> step h2 s2 h3 s3
-> step_n h1 s1 (S n) h3 s3.
```

**Lemma** step_n_left:

```coq
forall h1 s1 h2 s2 h3 s3, step_n h1 s1 n h2 s2
-> step h2 s2 h3 s3
-> step_n h1 s1 (S n) h3 s3.
Proof.
intros. induction H0.
- econstructor.
- eapply IHstep_n; eauto.
Qed.
```

**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 stmt. Instead, we'll use a notion of "fuel" to guarantee that our function always terminates.

**Fixpoint** run (fuel : nat) (h : heap) (s : stmt) : option (heap * stmt) :=

```coq
match fuel with
| O => None
| S n =>
  match interp_step h s
  | Some (h', s') => run n h' s'
  | None => Some (h, s)
  end.
```

---

**Definition**

```coq
match s with
| Nop => true
| _ => false
end.
```

**Lemma** isNop_ok:

```coq
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) :=

```coq
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, Seq s1 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, Seq s (While e s))
end.
```

**Lemma** interp_step_ok:

```coq
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. eapply 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.
  eapply interp_eval; eauto.
- eapply step_cond_true; eauto.
- eapply step_while_true; eauto.
- eapply interp_eval; eauto.
Qed.
```

**Inductive** step_n : heap -> stmt -> nat -> heap -> stmt -> Prop :=

```coq
| sn_refl: forall h s, step_n h s O h s
| sn_step: forall h1 s1 n h2 s2 h3 s3, step_n h1 s1 n h2 s2
-> step h2 s2 h3 s3
-> step_n h1 s1 (S n) h3 s3.
```

**Lemma** step_n_left:

```coq
forall h1 s1 h2 s2 h3 s3, step_n h1 s1 n h2 s2
-> step h2 s2 h3 s3
-> step_n h1 s1 (S n) h3 s3.
Proof.
intros. induction H0.
- econstructor.
- eapply IHstep_n; eauto.
Qed.
```

**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 stmt. Instead, we'll use a notion of "fuel" to guarantee that our function always terminates.

**Fixpoint** run (fuel : nat) (h : heap) (s : stmt) : option (heap * stmt) :=

```coq
match fuel with
| O => None
| S n =>
  match interp_step h s
  | Some (h', s') => run n h' s'
  | None => Some (h, s)
  end.
```
end.

(** and we can verify our interpreter too *)

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:?:.
    + apply Hfuel in H.
      apply interp_step_ok in Hqo.
      destruct H. exists (S x).
      eapply step_n_left; eauto.
    + inversion H; subst.
      exists 0. constructor; auto.
Qed.

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

Extraction interp_expr.
Extraction interp_step.
Extraction run.

Extraction fib_x_y.
Extraction gcd_x_y.

(** we can also make versions of our programs
    that set up an useful initial heaps *)

Definition fib_prog (x: Z) : stmt :=
  "x" <- x;
  fib_x_y.

Definition gcd_prog (x y: Z) : stmt :=
  "x" <- x;
  "y" <- y;
  gcd_x_y.

(** use analogous OCaml types *)
Require Import ExtrOcamlBasic.
Require Import ExtrOcamlNatInt.
Require Import ExtrOcamlZInt.
Require Import ExtrOcamlString.

(** we can even put these in a file and run them! *)
Cd "~/505-au15/www/L04/".
Extraction "Imp.ml" run empty fib_prog gcd_prog.