```
(** * Lecture 04 *)

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 :=
| Int : Z -> expr
| Var : string -> expr
| BinOp : binop -> expr -> expr -> expr.

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 Mul X Y) (at level 50, left associativity).
Notation "X [/] Y" := (BinOp Div X Y) (at level 50, left associativity).
(** NOTE: get me to tell story of Div/Mod bug at end! *)
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 Lte 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 :=
| Nop : stmt
| Assign : string -> expr -> stmt
| Seq : stmt -> stmt -> stmt
| Cond : expr -> stmt -> stmt
| While : expr -> stmt -> stmt.

Notation "'nop'" := (Nop) (at level 60).
Notation "X <- Y" := (Assign X Y) (at level 60).
Notation "X ;; Y" := (Seq X Y) (at level 61).
Notation "'if' X {{ Y }}" := (Cond X Y) (at level 60).
Notation "'while' X {{ Y }}" := (While X Y) (at level 60).

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 [<] "x" [%] "i" [||] 0 [<] "y" [%] "i") {{
    "i" <- "i" [-] 1
```

```
  }}.

(** SEMANTICS *)

(** 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 1 else 0
  | Lte => if Z_le_dec i1 i2 then 1 else 0
  | Conj => if Z_eq_dec i1 0 then 0 else
              if Z_eq_dec i2 0 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.

(** 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 e1 i1 ->
    eval h e2 i2 ->
    exec_op op i1 i2 = i3 ->
    eval h (BinOp op e1 e2) i3.

(** 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.
  induction e;
    (** simpl goal and context in all subgoals *)
    simpl in *.
  - (** NOTE: coercions make it look like types are bogus! *)
    subst. (** replace z with i everywhere *)
    constructor. (** 'apply eval_int' would also work here *)
  - subst. (** replace i with (h s) everywhere *)
    constructor. (** 'apply eval_var' would also work here *)
  - (** '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 i1, i2.
>>
        because it needs to know those to apply
        the constructor.  We can use a variant
        of apply to tell Coq exacly what i1 and i2
        should be. *)
    apply eval_binop with (i1 := interp_expr h e1)
                          (i2 := interp_expr h e2).
    (** now we have extra subgoals for the
        hypotheses of the eval_binop constructor *)
    + (** UGH.  IHe1 is too weak, for a specific i :( *)
      (** back out and try again *)
      (** REMEMBER: don't intro too many things too soon!!! *)
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 H; subst; reflexivity.
  - inversion H; subst.
    rewrite (IHe1 i1 H4). (** we can "fill in" an equality to rewrite with *)
    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.
  - 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,
  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 ->
  eval h e i2 ->
  i1 = i2.
Proof.
  (** set up a strong induction hyp *)
  intros h e i1 i2 H. revert i2.
  induction H; intros.
  - inversion H. subst. reflexivity.
  - inversion H. subst. reflexivity.
  - inversion H2. subst.
    apply IHeval1 in H7.
    apply IHeval2 in H9.
    subst. reflexivity.
Qed.

Lemma eval_swap_add:
  forall h e1 e2 i,
  eval h (BinOp Add e1 e2) i <-> eval h (BinOp Add e2 e1) i.
Proof.
  (** TODO complete this proof *)
Admitted.

Lemma interp_expr_swap_add:
  forall h e1 e2,
  interp_expr h (BinOp Add e1 e2) = interp_expr h (BinOp Add e2 e1).
Proof.
  (** TODO complete this proof *)
Admitted.

Lemma eval_add_zero:
  forall h e i,
  eval h (BinOp Add e (Int 0)) i <-> eval h e i.
Proof.
  (** TODO complete this proof *)
Admitted.

Lemma interp_expr_add_zero:
  forall h e,
  interp_expr h (BinOp Add e (Int 0)) = interp_expr h e.
Proof.
  (** TODO complete this proof *)
Admitted.

Lemma eval_mul_zero:
  forall h e i,
  eval h (BinOp Mul e (Int 0)) i <-> i = 0.
Proof.
  (** TODO complete this proof *)
Admitted.

Lemma interp_expr_mul_zero:
  forall h e,
  interp_expr h (BinOp Mul e (Int 0)) = 0.
Proof.
  (** TODO complete this proof *)
Admitted.

(** Huh, so why ever have relational semantics? *)
```