```
(** * 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.

(** Don't worry about these! *)
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.
*)

(** The "logical" operators (Conj, Disj) are a little tricky--remember that "0"
is false! *)
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.

(** In class at this point we talked about "SearchAbout". For example: *)
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,
    (** remember that the "memory" h is just a function *)
    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.
(** As Zach stressed in class, "eval" defines a MEANING for expressions!!!! *)
```

```
Lemma eval_ex1 :
  eval empty ("x" [+] 1) 1.
Proof.
  apply eval_binop with (i1 := 0) (i2 := 1).
  - apply eval_var.
  - 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.
  inversion H4.
  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.
  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.




(**

<<


                                    ~-.
          ''';              ~-.~-.~-
         (.../             ~-.~-.~-.~-.~-.
         } o~`,           ~-.~-.~-.~-.~-.~-.
         (/    \        ~-.~-.~-.~-.~-.~-.~-.
          ;    \      ~-.~-.~-.~-.~-.~-.~-.
         ;     {_.~-.~-.~-.~-.~-.~-.~
```



```
        ;:  .-~'     ~-.~-.~-.~-.~-.
       ;.: :'     ._    ~-.~-.~-.~-.~-
        ;::`-.     '-._  ~-.~-.~-.~-
         ;::.  `-.     '-,~-.~-.~-.
          ';::::.`''-.-'
           ';::;;:,:'
              '||"
              / |
            ~' ~"'


>>

*)




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.
  - inversion H; subst.
    econstructor; eauto.
    simpl. omega.
  - inversion H; subst.
    econstructor; eauto.
    simpl. omega.
Qed.

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.
    (** 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 <-> i = 0.
Proof.
  split; intros.
  - inversion H; subst.
    inversion H6; subst.
```

```
    simpl. omega.
  - subst.
    pose (interp_expr h e).
    eapply eval_binop with (i1 := z).
    + apply 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.

(** 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 Nop s) h s
| step_seq:
  forall h s1 s2 s1' h',
  step h s1 h' s1' ->
  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. (** umpossible! *)
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 =>
      Some (update h v (interp_expr h e), Nop)
  | Seq Nop s =>
      Some (h, s)
  | Seq s1 s2 =>
      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.

*)

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

```
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.

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.
    + 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 IHs1 in Heqo.
        constructor. assumption.
      * discriminate.
  - destruct (Z.eq_dec (interp_expr h e) 0) eqn:?.
    + 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.
```



```
          We could explicity use the
            'apply step_cond_false with (i := ...)'
          flavor of apply to specify 'i', but using
          'econstructor' is more convenient and flexible. *)
      econstructor.
      (** 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.
      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.
      apply interp_eval; auto.
    + inversion H; subst.
      eapply step_while_true; eauto.
      apply 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 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.

(** Notice how we "add a step" to the end for step_n.
    We can also "add a step" on the beginning. *)
Lemma step_n_left:
  forall h1 s1 h2 s2 n h3 s3,
  step h1 s1 h2 s2 ->
  step_n h2 s2 n h3 s3 ->
  step_n h1 s1 (S n) h3 s3.
Proof.
  intros. induction H0.
  - econstructor.
    + econstructor.
    + assumption.
  - econstructor.
    + eapply IHstep_n; eauto.
    + assumption.
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) :=
  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
```

```
  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 IHfuel in H.
      apply interp_step_ok in Heqo.
      destruct H. exists (S x).
      eapply step_n_left; eauto.
    + inversion H; subst.
      exists O. constructor; auto.
Qed.

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

Extraction interp_expr.
Extraction interp_step.
Extraction run.

Extraction fib_x_y.
Extraction gcd_xy_i.

(** 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_xy_i.

(** 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.
```