Chapter 1

Library SemanticEquivelence

```
Require Import Coq.Init.Peano.
Require Import Coq.Init.Nat.
Require Import Coq.Arith.EqNat.
Require Import Coq.Arith.PeanoNat.
Include Coq.Init.Nat.
Check _/_.
Require Import List.
Import ListNotations.
Check [1].
Definition stack := list nat.
Inductive bit : Set :=
  | I : bit
  | O : bit.
Definition bitstr := list bit.
Inductive expr (ops : Set) (lits : Set) : Set :=
  | Lit : lits → expr ops lits
  | Bin : ops → expr ops lits → expr ops lits → expr ops lits.
Arguments Lit {ops} {lits}.
Arguments Bin {ops} {lits}.
Check Lit 1.
Inductive semantics (L : Set) (D : Set) : Type :=
  sem_f : (L → D) → semantics L D.
Arguments sem_f {L} {D}.
Inductive semantic_evaluation {L D : Set} (sem : L → D) | (e : L) (d : D) : Prop
  := sem_eval : sem e = d → semantic_evaluation e d.
Definition sem {L D} (s : semantics L D) : L → D :=
```

```
  let 'sem_f sem_fun := s in sem_fun.

Notation "[[ x ]][ s ]" := (sem s x).

Fixpoint expr_semantics'
  {ops lits : Set} {D : Set} (sem_lit : lits → D)
  (sem_op : ops → D → D → D) (e : expr ops lits) : D :=
  let expr_sem := expr_semantics' sem_lit sem_op in
    match e with
      | Lit x ⇒ sem_lit x
      | Bin op e1 e2 ⇒ sem_op op (expr_sem e1) (expr_sem e2)
      end.

Definition expr_semantics
  {ops lits D : Set} (sem_lit : lits → D) (sem_op : ops → D → D → D) : semantics (expr
ops lits) D :=
  sem_f (expr_semantics' sem_lit sem_op).

Inductive expr_semantics_definition (lits ops D : Set) : Set :=
  | expr_sem_def_parts : ∀ (sem_lit : lits → D) (sem_op : ops → D → D → D),
expr_semantics_definition lits ops D.

Inductive bin_ops : Set := bin_add.

Definition expr_bits := expr bin_ops bitstr.

Fixpoint expr_bits_sem_lits (bs : bitstr) : nat :=
  match bs with
    | I :: bs' ⇒ 1 + (2 × expr_bits_sem_lits bs')
    | O :: bs' ⇒ 2 × expr_bits_sem_lits bs'
    | [] ⇒ 0
    end.

Definition expr_bits_sem_op (o : bin_ops) : nat → nat → nat :=
  match o with
    | bin_add ⇒ fun a b ⇒ a + b
    end.

Definition bin_expr_sems := expr_semantics expr_bits_sem_lits expr_bits_sem_op.

Check bin_expr_sems : semantics expr_bits nat.

Definition eval_expr := expr_semantics' expr_bits_sem_lits expr_bits_sem_op.

Compute (sem bin_expr_sems (Lit [I])).
Compute (eval_expr (Lit [I])).

Fixpoint bits_to_nat (b : bitstr) : nat :=
  match b with
    | I :: b' ⇒ S (2 × bits_to_nat b')
    | O :: b' ⇒ (2 × bits_to_nat b')
```

```
    | [] ⇒ 0
    end.

Definition nat_to_bit (n : nat) : bit :=
  match n with
    | 0 ⇒ O
    | 1 ⇒ I
    | _ ⇒ O
  end.

Fixpoint bit_inc (bs : bitstr) : bitstr :=
  match bs with
    | [] ⇒ [I]
    | I :: bs' ⇒ O :: bit_inc bs'
    | O :: bs' ⇒ I :: bs'
    end.

Fixpoint nat_to_bits (n : nat) : bitstr :=
  match n with
    | 0 ⇒ [O]
    | S n' ⇒ bit_inc (nat_to_bits n')
    end.

Compute bits_to_nat (I :: I :: O :: []).

Compute nat_to_bits 4.

Lemma bits_to_nat_function : ∀ (b1 b2 : bitstr), b1 = b2 → bits_to_nat b1 = bits_to_nat
b2.
Proof.
  intros b1 b2.
  destruct b1.
  intros. rewrite H. reflexivity.
  destruct b2.
  intros. rewrite H. reflexivity.
  intros. rewrite H. reflexivity.
Qed.

Fixpoint bit_add (b1 b2 : bitstr) : bitstr :=
  match b1,b2 with
    | I :: b1', b :: b2' ⇒ bit_inc (b :: bit_add b1' b2')
    | O :: b1', b :: b2' ⇒ b :: bit_add b1' b2'
    | [],_ ⇒ b2
    | _,[] ⇒ b1
    end.

Notation "x +.+ y" := (bit_add x y) (at level 40, left associativity).

Compute bit_add (nat_to_bits 100) (nat_to_bits 23).
```

Compute $bits\_to\_nat$ (bit_add (nat_to_bits 100) (nat_to_bits 23)).

Definition $binary := bitstr$.

```
Lemma binary_add_unit_left : ∀ (b : binary), b +.+ [] = b.
Proof.
  intros b.
  induction b.
  - simpl. reflexivity.
  - simpl. destruct a. all: reflexivity.
Qed.
```

```
Lemma binary_add_unit_right : ∀ (b : binary), [] +.+ b = b.
Proof.
  intros b.
  induction b.
  - simpl. reflexivity.
  - simpl. destruct a. all: reflexivity.
Qed.
```

```
Lemma binary_add_id_left : ∃ (e : binary), ∀ (b : binary), e +.+ b = b.
Proof.
  ∃ [].
  intros.
  simpl.
  reflexivity.
Qed.
```

```
Lemma binary_add_id_right : ∃ (e : binary), ∀ (b : binary), b +.+ e = b.
Proof.
  ∃ [].
  intros.
  induction b.
  - simpl. reflexivity.
  - simpl. destruct a. all: reflexivity.
Qed.
```

```
Lemma add_id : ∀ (n : nat), n + 0 = n.
Proof.
  intros n.
  apply Nat.add_comm.
Qed.
```

```
Lemma add_succ_assoc : ∀ (n m : nat), S (n + m) = (S n) + m.
Proof.
  auto.
Qed.
```

```
Lemma bits_nat_succ_eq : ∀ (n : nat), bit_inc (nat_to_bits n) = nat_to_bits (S n).
Proof.
  induction n.
  simpl. reflexivity.
  simpl. reflexivity.
Qed.

Lemma nat_bits_succ_eq : ∀ (b : bitstr), S (bits_to_nat b) = bits_to_nat (bit_inc b).
Proof.
  intros b.
  induction b.
   - simpl. reflexivity.
   - pose proof (bits_nat_succ_eq (bits_to_nat b)) as H.
   destruct a.
   all: simpl.
   all: rewrite (add_id (bits_to_nat b)).
   all: try reflexivity.
   rewrite (add_id (bits_to_nat (bit_inc b))).
   rewrite ← IHb.
   pose proof (add_succ_assoc (bits_to_nat b) (S (bits_to_nat b))) as H2.
   pose proof (add_succ_assoc (S (bits_to_nat b)) (bits_to_nat b)) as H3.
   rewrite ← H2.
   pose proof (Nat.add_comm (bits_to_nat b) (S (bits_to_nat b))) as H4.
   rewrite → H4.
   pose proof (add_succ_assoc (bits_to_nat b) (bits_to_nat b)) as H5.
   rewrite ← H5.
   reflexivity.
Qed.

Notation "[[ x ]]" := (bits_to_nat x).

Lemma binary_add_linearity_1 :
     (∀ (b : bitstr), bits_to_nat ([O] +.+ b) = bits_to_nat b ∧ bits_to_nat (b +.+ [O]) =
bits_to_nat b) →
       ∀ (n : nat), bits_to_nat (nat_to_bits n +.+ [O]) = n + bits_to_nat [O].
Proof.
  intros.
  simpl.
  induction n.
  simpl. reflexivity.
  simpl. pose proof (bits_nat_succ_eq n) as H2. rewrite → H2.
  simpl in IHn. pose proof (bits_nat_succ_eq n) as H3. simpl in H3.
  rewrite → Nat.add_comm in IHn. rewrite → Nat.add_0_l in IHn.
  rewrite → Nat.add_comm. rewrite → Nat.add_0_l.
  simpl.
```

```
  rewrite H2.
  rewrite ← IHn.
  pose proof (nat_bits_succ_eq (nat_to_bits n +.+ [O])) as H4.
  rewrite → H4.
  pose proof (H (nat_to_bits n)) as H5.
  destruct H5.
  simpl in H1.
  simpl.
  rewrite ← H4.
  rewrite → H1.
  pose proof (H (nat_to_bits (S (bits_to_nat (nat_to_bits n))))) as H6.
  destruct H6.
  rewrite → H6.
  rewrite ← H5.
  rewrite H5.
  pose proof H6 as H7.
  simpl in H6.
  simpl.
  rewrite ← H1.
  rewrite IHn.
  simpl.
  rewrite H2.
  simpl.
  pose proof (nat_bits_succ_eq (nat_to_bits n)) as H8.
  rewrite ← H8.
  rewrite ← H1.
  rewrite → IHn.
  reflexivity.
Qed.

Definition str1 := nat_to_bits 2.
Definition str2 := nat_to_bits 1.
Compute (bit_add str1 str2).

Theorem bit_nat_correspondance : ∀ (b c : bitstr), [[ b +.+ c ]] = [[ b ]] + [[ c ]].
Proof.
  intros.
  induction b.
  induction c.
  simpl. reflexivity.
  destruct a.
  all: simpl.
  repeat (rewrite → Nat.add_assoc || rewrite → Nat.add_0_l || rewrite → Nat.add_comm).
  reflexivity.
```

```
  reflexivity.
  destruct a.
  auto.
  simpl.
  rewrite -> Nat.add_assoc.
  unfold bit_add.
  auto.
  simpl.
  auto.
  unfold bit_add.
  unfold bit_inc.
  auto.
  simpl.
  unfold bits_to_nat.
  unfold bits_to_nat.
  unfold bit_add.
  simpl.
  auto.
  induction c.
  simpl. reflexivity.
  auto.
  omega.
  rewrite <- IHb.
  rewrite -> Nat.add_comm.
  rewrite -> IHb.
  auto.
  simpl
Lemma bit_add_succ : ∀ (a b : bitstr), [[a]] = [[b]] -> [[ (bit_inc a) +.+ b ]] = [[ bit_inc (a
+.+ b) ]].
Proof.
  intros.
  unfold bits_to_nat.
  rewrite H.
  induction b.
  unfold bit_inc. simpl.
  reflexivity.
   unfold bit_add.
   simpl.
   destruct b.
   unfold bit_inc.
  reflexivity.
```

```
  destruct b.
  unfold bit_inc.
  all: simpl.
  induction a.
    simpl.
    destruct b.
Theorem nat_to_bit_add_hom : ∀ (n m : nat), nat_to_bits (n + m) = bit_add (nat_to_bits
n) (nat_to_bits m).
Proof.
  intros.
  induction n.
  - simpl. reflexivity.
  - simpl. rewrite IHn.
    induction m.

Theorem bit_to_nat_add_hom : ∀ (a b : bitstr), bits_to_nat (bit_add a b) = (bits_to_nat a)
+ (bits_to_nat b).
Proof.

Theorem idempotent_nat : ∀ (n : nat), bits_to_nat (nat_to_bits n) = n.
Proof.
  intros n.
  induction n.
  simpl. reflexivity.
  simpl.
  pose proof (bits_nat_succ_eq n) as H1.
  pose proof (nat_bits_succ_eq (nat_to_bits n)) as H2.
  pose proof (nat_bits_succ_eq (nat_to_bits n)) as H3. rewrite → IHn in H3.
  rewrite H3.
  reflexivity.
Qed.

Theorem thm_idk : ∀ (n : nat) (b : bitstr), n = bits_to_nat b → nat_to_bits n = b.
Proof.
  intros n b H.
  induction b.
    induction n.
    rewrite H. simpl. reflexivity.
    rewrite H. simpl. reflexivity.
    pose proof (idempotent_nat n) as H2.
    pose proof (idempotent_nat n) as H3.
    rewrite → H in H3.
    rewrite H.
    destruct a.
      simpl.
```

```
Theorem idempotent_bits : ∀ (b : bitstr) (n : nat), n = bits_to_nat b → nat_to_bits
(bits_to_nat b) = b.
Proof.
  intros.
  pose proof (idempotent_nat n) as H2.
  pose proof (idempotent_nat n) as H3. rewrite → H in H3.
  pose proof H3 as H4. rewrite ← H in H4.
  rewrite ← H.
  simpl.
  induction b.
    - simpl. rewrite → H. simpl. reflexivity.
    - simpl. rewrite → H. destruct a.
    all: simpl.
    all: repeat (rewrite → Nat.add_assoc || rewrite → Nat.add_0_l || rewrite →
Nat.add_comm).
    pose proof (bits_nat_succ_eq (bits_to_nat b + bits_to_nat b)) as H5.
    pose proof (bits_nat_succ_eq (bits_to_nat b + bits_to_nat b)) as H6.
    pose proof H as H7. simpl in H7. repeat (rewrite → Nat.add_assoc in H7 ||
rewrite → Nat.add_0_l in H7 || rewrite → Nat.add_comm in H7).
    simpl.
    rewrite → H5.
    pose proof (idempotent_nat (bits_to_nat b + bits_to_nat b)) as H8.
    pose proof H7 as H10.
    rewrite ← H8 in H10.
    pose proof H10 as H11.
    rewrite → H7 in H11.
    pose proof H11 as H12.
    pose proof (bits_nat_succ_eq (bits_to_nat b + bits_to_nat b)) as H13.
    rewrite ← H7.
    pose proof (nat_bits_succ_eq (I :: b)) as H14. simpl in H14.
    unfold nat_to_bits. simpl.
    rewrite ← H13. simpl. unfold bit_inc. simpl. simpl in H13.
    rewrite → H11.
    simpl.
    all: simpl.
    all: try rewrite Nat.add_comm.
    all: replace (bits_to_nat b + 0 + bits_to_nat b) with (bits_to_nat b + bits_to_nat b).
    cbv.
    rewrite ← H. rewrite ← H2.
    induction n.
    all: rewrite ← H. simpl.

    destruct a.
```

```
    all: simpl.
  induction n.

  induction n.
  rewrite ← H. simpl.
  intros b.
  pose proof (idempotent_nat (bits_to_nat b)) as H.
  pose proof (nat_bits_succ_eq b) as H2.
  pose proof (bits_to_nat b) as n.
  replace (nat_to_bits (bits_to_nat b) = b) with (bits_to_nat (nat_to_bits (bits_to_nat b))
= bits_to_nat b).
  rewrite ← H.
  simpl.
Theorem bits_num_bijection : ∀ (n : nat), ∃ (b : bitstr),

Theorem bits_additive_identity :

Theorem list_sem_correctness : ∀ (b : bitstr) (n : nat), bits_to_nat b = n → eval (Lit b)
= n.
Proof.
Theorem bit_expr_add_hom : ∀ (b1 b2 : bitstr) (n m : nat), eval (Bin bin_add b1 b2) =
eval ()
```