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.

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.

Lemma bits_to_nat_function : ∀ (b1 b2 : bitstr), b1 = b2 → bits_to_nat b1 = bits_to_nat
b2.

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).

Definition binary := bitstr.

Lemma binary_add_unit_left : ∀ (b : binary), b +.+ [] = b.

Lemma binary_add_unit_right : ∀ (b : binary), [] +.+ b = b.

Lemma binary_add_id_left : ∃ (e : binary), ∀ (b : binary), e +.+ b = b.

Lemma binary_add_id_right : ∃ (e : binary), ∀ (b : binary), b +.+ e = b.

Lemma add_id : ∀ (n : nat), n + 0 = n.

Lemma add_succ_assoc : ∀ (n m : nat), S (n + m) = (S n) + m.

Lemma bits_nat_succ_eq : ∀ (n : nat), bit_inc (nat_to_bits n) = nat_to_bits (S n).

Lemma nat_bits_succ_eq : ∀ (b : bitstr), S (bits_to_nat b) = bits_to_nat (bit_inc b).

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

`Lemma` *binary_add_linearity_1* :

$(\forall\ (b : bitstr),\ bits\_to\_nat\ ([O] \mathbin{+.+} b) = bits\_to\_nat\ b \land bits\_to\_nat\ (b \mathbin{+.+} [O]) = bits\_to\_nat\ b) \to$

$\forall\ (n : nat),\ bits\_to\_nat\ (nat\_to\_bits\ n \mathbin{+.+} [O]) = n + bits\_to\_nat\ [O].$

`Definition` $str1 := nat\_to\_bits\ 2.$

`Definition` $str2 := nat\_to\_bits\ 1.$

`Theorem` $bit\_nat\_correspondance : \forall\ (b\ c : bitstr),\ [[\ b \mathbin{+.+} c\ ]] = [[\ b\ ]] + [[\ c\ ]].$

`Theorem` $nat\_to\_bit\_add\_hom : \forall\ (n\ m : nat),\ nat\_to\_bits\ (n + m) = bit\_add\ (nat\_to\_bits\ n)\ (nat\_to\_bits\ m).$

`Theorem` $bit\_to\_nat\_add\_hom : \forall\ (a\ b : bitstr),\ bits\_to\_nat\ (bit\_add\ a\ b) = (bits\_to\_nat\ a) + (bits\_to\_nat\ b).$

`Theorem` $idempotent\_nat : \forall\ (n : nat),\ bits\_to\_nat\ (nat\_to\_bits\ n) = n.$

`Theorem` $thm\_idk : \forall\ (n : nat)\ (b : bitstr),\ n = bits\_to\_nat\ b \to nat\_to\_bits\ n = b.$

`Theorem` $idempotent\_bits : \forall\ (b : bitstr)\ (n : nat),\ n = bits\_to\_nat\ b \to nat\_to\_bits\ (bits\_to\_nat\ b) = b.$

`Theorem` $bits\_num\_bijection : \forall\ (n : nat),\ \exists\ (b : bitstr),$

`Theorem` $bits\_additive\_identity :$

`Theorem` $list\_sem\_correctness : \forall\ (b : bitstr)\ (n : nat),\ bits\_to\_nat\ b = n \to$ `eval` $(Lit\ b) = n.$

`Theorem` $bit\_expr\_add\_hom : \forall\ (b1\ b2 : bitstr)\ (n\ m : nat),$ `eval` $(Bin\ bin\_add\ b1\ b2) =$ `eval` $()$