```
Require Import List.
Require Import String.
Require Import ZArith.

Open Scope list_scope.
Open Scope string_scope.
Open Scope Z_scope.

Require Import StructTactics.
Require Import ImpSyntax.
Require Import ImpCommon.
Require Import ImpEval.
Require Import ImpSemanticsFacts.
Require Import ImpInterpNock.

Import ListNotations.

Ltac index_simplify :=
  rewrite ? Zlength_correct in *;
  autorewrite with list in *;
  cbn [List.length] in *; zify;
  rewrite ? Z2Nat.id in * by omega.

Ltac index_crush := index_simplify; omega.
Hint Extern 3 (read _ _ = _) =>
match goal with
| [ Hheap : forall z, _ -> read ?h0 z = read _ z |- _ ] =>
  erewrite Hheap by omega
end.

Lemma eval_stmt_while_elim :
  forall (I : store -> heap -> Prop) env s h e p s' h',
    I s h ->
    (forall s0 h0 s1 h1,
        I s0 h0 ->
        eval_e s0 h0 e (Vbool true) ->
        eval_s env s0 h0 p s1 h1 ->
        I s1 h1) ->
    eval_s env s h (Swhile e p) s' h' ->
    I s' h' /\ eval_e s' h' e (Vbool false).
Proof.
  intros.
  prep_induction H1.
  induction H1; intros; subst;
    try discriminate; find_inversion; eauto.
Qed.

Lemma read_write_same :
  forall h h' a x,
    write h a x = Some h' ->
    read h' a = Some x.
Proof.
  induction h; simpl; intros.
  - discriminate.
  - repeat break_match; repeat find_inversion; auto; try congruence.
    all: cbn [read]; auto.
Qed.

Lemma read_write_neq :
  forall h h' a b x,
    write h a x = Some h' ->
    a <> b ->
    read h' b = read h b.
Proof.
  induction h; simpl; intros.
  - discriminate.
  - repeat break_match; repeat find_inversion; try congruence.
    all: cbn[read]; eauto.
    all: eapply IHh; eauto; omega.
```



```
Qed.

Lemma read_write_nop :
  forall h a x,
    read h a = Some x ->
    write h a x = Some h.
Proof.
  induction h; simpl; intros.
  - discriminate.
  - destruct a0.
    + solve_by_inversion.
    + rewrite IHh; auto.
    + rewrite IHh; auto.
Qed.


Fixpoint array_at' (base : Z) (contents : list Z) (h : heap) : Prop :=
  match contents with
  | [] => True
  | z :: contents => read h base = Some (Vint z) /\ array_at' (base + 1) content
s h
  end.

Lemma array_at'_read_nth_error :
  forall contents base h x i,
    array_at' base contents h ->
    0 <= i < Zlength contents ->
    read h (base + i) = Some (Vint x) ->
    nth_error contents (Z.to_nat i) = Some x.
Proof.
  induction contents; simpl; intros.
  - rewrite Zlength_nil in *. omega.
  - rewrite Zlength_cons in *. break_and.
    destruct (Z.to_nat i) eqn:?.
    + simpl. change 0%nat with (Z.to_nat 0) in Heqn.
      apply Z2Nat.inj in Heqn; try omega.
      subst i.
      rewrite Z.add_0_r in *. congruence.
    + simpl. zify.
      rewrite Z2Nat.id in * by auto.
      assert (0 < i) by omega.
      subst i.
      rewrite <- Nat2Z.id with (n := n).
      eapply IHcontents; eauto.
      * omega.
      * rewrite <- H1. f_equal. omega.
Qed.

Definition array_at (base : Z) (contents : list Z) (h : heap) : Prop :=
  match read h base with
  | None => False
  | Some l => l = Vint (Zlength contents) /\
              array_at' (base + 1) contents h
  end.

Lemma array_at_array_at' :
  forall a l h,
    array_at a l h ->
    array_at' (a + 1) l h.
Proof.
  unfold array_at.
  intros.
  break_match; intuition.
Qed.

Lemma array_at'_extend_r :
  forall l h a i x,
    array_at' a l h ->
    read h i = Some (Vint x) ->
```

```
    i = a + Zlength l ->
    array_at' a (l ++ [x]) h.
Proof.
  induction l; intros.
  - rewrite Zlength_correct in *.
    simpl in *.
    intuition.
    assert (i = a) by omega.
    subst a.
    auto.
  - rewrite Zlength_correct in *.
    simpl in *. intuition.
    eapply IHl; eauto.
    zify. omega.
Qed.


Lemma array_at'_extend_l :
  forall l h a x,
    array_at' (a + 1) l h ->
    read h a = Some (Vint x) ->
    array_at' a (x :: l) h.
Proof.
  intros.
  simpl.
  intuition.
Qed.

Lemma array_at'_write_preserve :
  forall l h h' a i x,
    array_at' a l h ->
    write h i x = Some h' ->
    ~ a <= i < a + Zlength l ->
    array_at' a l h'.
Proof.
  induction l; intros.
  - simpl. auto.
  - cbn [array_at'] in *. break_and.
    rewrite Zlength_correct in *.
    cbn [Datatypes.length] in *.
    split.
    + erewrite read_write_neq; eauto.
      zify. omega.
    + eapply IHl; eauto.
      zify. omega.
Qed.

Lemma array_at'_write_extend_r :
  forall l h h' a i x,
    array_at' a l h ->
    write h i (Vint x) = Some h' ->
    i = a + Zlength l ->
    array_at' a (l ++ [x]) h'.
Proof.
  intros.
  eapply array_at'_extend_r.
  eapply array_at'_write_preserve.
  eauto.
  eauto.
  omega.
  erewrite read_write_same; eauto.
  auto.
Qed.


Lemma array_at'_write_extend_l :
  forall l h h' a x,
    array_at' (a + 1) l h ->
    write h a (Vint x) = Some h' ->
    array_at' a (x :: l) h'.
Proof.
  intros.
  eapply array_at'_extend_l.
  eapply array_at'_write_preserve.
  eauto.
  eauto.
  omega.
  erewrite read_write_same; eauto.
Qed.

Lemma array_at'_shrink_r :
  forall l a h,
    array_at' a l h ->
    array_at' a (removelast l) h.
Proof.
  induction l; simpl; repeat break_match; intuition.
  cbn [array_at']. intuition.
Qed.

Lemma array_at'_read_nth :
  forall contents base h i d,
    array_at' base contents h ->
    0 <= i < Zlength contents ->
    read h (base + i) = Some (Vint (nth (Z.to_nat i) contents d)).
Proof.
    induction contents; simpl; intros.
  - rewrite Zlength_nil in *. omega.
  - rewrite Zlength_cons in *. break_and.
    destruct (Z.to_nat i) eqn:?.
    + simpl. change 0%nat with (Z.to_nat 0) in Heqn.
      apply Z2Nat.inj in Heqn; try omega.
      subst i.
      rewrite Z.add_0_r in *. congruence.
    + simpl. zify.
      rewrite Z2Nat.id in * by auto.
      assert (0 < i) by omega.
      subst i.
      replace (base + Z.succ (Z.of_nat n)) with ((base + 1) + Z.of_nat n) by ome
ga.
      erewrite IHcontents; eauto; try omega.
      rewrite !Nat2Z.id. eauto.
Qed.

Lemma array_at'_app :
  forall l1 a1 a2 l2 h,
    array_at' a1 l1 h ->
    array_at' a2 l2 h ->
    a2 = a1 + Zlength l1 ->
    array_at' a1 (l1 ++ l2) h.
Proof.
  induction l1; simpl; intuition.
  - rewrite Zlength_nil in *.
    assert (a2 = a1) by omega.
    subst a1.
    auto.
  - rewrite Zlength_correct in *.
    cbn [Datatypes.length] in *.
    eapply IHl1; eauto.
    zify. omega.
Qed.


Lemma lkup_update :
  forall s x1 x2 v,
    lkup (update s x1 v) x2 = if String.string_dec x1 x2 then Some v else lkup s
 x2.
Proof.
  induction s; simpl; intros.
```

```
  - repeat break_if; congruence.
  - repeat (break_match; simpl in * ); try congruence.
    + rewrite IHs. break_if; congruence.
    + rewrite IHs. break_if; congruence.
Qed.

Lemma lkup_update_neq :
  forall s x1 x2 v,
    x1 <> x2 ->
    lkup (update s x1 v) x2 = lkup s x2.
Proof.
  intros.
  rewrite lkup_update.
  break_if; congruence.
Qed.

Lemma lkup_update_same :
  forall s x v,
    lkup (update s x v) x = Some v.
Proof.
  intros.
  rewrite lkup_update.
  break_if; congruence.
Qed.

Lemma array_at_read_nth_error :
  forall a contents h i x,
    array_at a contents h ->
    0 <= i < Zlength contents ->
    read h (Z.succ (a + i)) = Some (Vint x) ->
    nth_error contents (Z.to_nat i) = Some x.
Proof.
  unfold array_at.
  intros.
  break_match; intuition.
  subst v.
  eapply array_at'_read_nth_error. eauto. omega.
  rewrite <- H1. f_equal. omega.
Qed.

Lemma array_at_read_length :
  forall a contents h,
    array_at a contents h ->
    read h a = Some (Vint (Zlength contents)).
Proof.
  unfold array_at.
  intros.
  break_match; intuition.
  congruence.
Qed.
Hint Resolve array_at_read_length.

Lemma length_cons :
  forall A (a : A) l,
    List.length (a :: l) = S (List.length l).
Proof. auto. Qed.

Hint Rewrite length_cons : list.


Hint Constructors eval_e.
Hint Constructors eval_binop.
Hint Extern 3 (lkup (update _ _ _) _ = _) => rewrite lkup_update_neq by discrimi
nate.
Hint Extern 2 (lkup (update _ _ _) _ = _) => rewrite lkup_update_same.

Lemma pred_of_dec_true_elim :
  forall A (B : A -> A -> Prop)
    (dec : (forall a1 a2 : A, {B a1 a2} + {~ B a1 a2})) a1 a2,
```



```
    pred_of_dec dec a1 a2 = true ->
    B a1 a2.
Proof.
  unfold pred_of_dec.
  intros.
  break_if; congruence.
Qed.

Lemma pred_of_dec_false_elim :
  forall A (B : A -> A -> Prop)
    (dec : (forall a1 a2 : A, {B a1 a2} + {~ B a1 a2})) a1 a2,
    pred_of_dec dec a1 a2 = false ->
    ~ B a1 a2.
Proof.
  unfold pred_of_dec.
  intros.
  break_if; congruence.
Qed.


Ltac break_eval_expr :=
  repeat match goal with
  | [ H : eval_unop _ _ ?x |- _ ] => remember x; invc H; [idtac]
  | [ H : eval_binop ?op _ _ ?x |- _ ] => remember x; invc H; [idtac]
  | [ H : eval_e _ _ (Eval _) ?x |- _ ] => remember x; invc H; [idtac]
  | [ H : eval_e _ _ (Evar _) ?x |- _ ] => remember x; invc H; [idtac]
  | [ H : eval_e _ _ (Eop1 _ _) ?x |- _ ] => remember x; invc H; [idtac]
  | [ H : eval_e _ _ (Eop2 _ _ _) ?x |- _ ] => remember x; invc H; [idtac]
  | [ H : eval_e _ _ (Eidx _ _) ?x |- _ ] => remember x; invc H; [idtac]
  | [ H : eval_e _ _ (Eidx ?a ?b) _ |- _ ] =>
    eapply eval_e_idx_a_inv with (e1 := a) (e2 := b) in H; eauto; [idtac]
  | [ H : eval_e _ _ (Elen _) ?x |- _ ] => remember x; invc H; [idtac]
  | [ H : eval_e _ _ (Elen _) ?x |- _ ] =>
    eapply eval_e_len_a_inv in H; eauto using array_at_read_length; [idtac]
  end; repeat find_injection;
  unfold imp_eq, imp_lt, imp_le in *;
  repeat match goal with
  | [ H : pred_of_dec _ _ _ = true |- _ ] => apply pred_of_dec_true_elim in H
  | [ H : pred_of_dec _ _ _ = false |- _ ] => apply pred_of_dec_false_elim in H
  end.

Ltac step_forward :=
  match goal with
  | [ H : eval_s _ _ _ Snop _ _ |- _ ] => invc H
  | [ H : eval_s _ _ _ (Sset _ _) _ _ |- _ ] => invc H
  | [ H : eval_s _ _ _ (Swrite _ _ _) _ _ |- _ ] => invc H
  | [ H : eval_s _ _ _ (Salloc _ _ _) _ _ |- _ ] => invc H
  | [ H : eval_s _ _ _ (Scall _ _ _) _ _ |- _ ] => invc H
  | [ H : eval_s _ _ _ (Sseq _ _) _ _ |- _ ] => invc H
  | [ H : eval_s _ _ _ (Sifelse _ _ _) _ _ |- _ ] => invc H
  end.

Ltac normalize_Z :=
      repeat rewrite !Z.add_assoc in *;
      repeat
        match goal with
        | [ H : array_at' ?x _ _ |- _ ] => revert H
        | [ H : read _ ?x = _  |- _ ] => revert H
        | [ H : write _ ?x _ = _  |- _ ] => revert H
        end;
      repeat (match goal with
        | [ |- array_at' ?x _ _ -> _ ] => ring_simplify x
        | [ |- read _ ?x = _ -> _ ] => ring_simplify x
        | [ |- write _ ?x _ = _ -> _ ] => ring_simplify x
        end; intro);
      repeat match goal with
      | [ |- array_at' ?x _ _ ] => ring_simplify x
      end;
      repeat match goal with
```

```
      | [ |- read _ ?x = _ ] => ring_simplify x
      end;
      repeat match goal with
      | [ |- write _ ?x _ = _ ] => ring_simplify x
      end.

Definition after env s h p
  (M : Z -> Prop)
  (Q : store -> heap -> Prop) : Prop :=
  exists s' h',
    eval_s env s h p s' h' /\
    (forall z, ~ M z -> read h' z = read h z) /\
    Q s' h'.

Lemma after_seq :
  forall env s h p1 p2 M Q,
    after env s h p1 M
      (fun s' h' => after env s' h' p2 M Q) ->
    after env s h (Sseq p1 p2) M Q.
Proof.
  unfold after.
  intros.
  break_exists.
  break_and.
  break_exists.
  break_and.
  eauto 10 using eval_seq, eq_trans.
Qed.

Lemma after_set :
  forall env s h x e v M (Q : store -> heap -> Prop),
    eval_e s h e v ->
    Q (update s x v) h ->
    after env s h (Sset x e) M Q.
Proof.
  unfold after.
  intros.
  eauto 10 using eval_set.
Qed.

Lemma after_write :
  forall env s h h' x e1 e2 a l i v (M : _ -> Prop) (Q : store -> heap -> Prop),
    lkup s x = Some (Vaddr a) ->
    read h a = Some (Vint l) ->
    eval_e s h e1 (Vint i) ->
    eval_e s h e2 v ->
    0 <= i < l ->
    write h (Z.succ (a + i)) v = Some h' ->
    M (Z.succ (a + i)) ->
    Q s h' ->
    after env s h (Swrite x e1 e2) M Q.
Proof.
  unfold after.
  intros.
  do 2 eexists.
  split.
  econstructor; eauto.
  intuition.
  eapply read_write_neq; eauto.
  congruence.
Qed.

Lemma read_write_success:
  forall h a (v1 v2 : val),
    read h a = Some v1 -> write h a v2 <> None.
Proof.
  induction h; intros.
  + inversion H.
  + simpl in H.
```



```
    break_match.
    simpl. congruence.
    unfold write.
    fold write.
    eapply IHh with (v2 := v2) in H.
    break_match; try congruence.
    unfold write.
    fold write.
    eapply IHh with (v2 := v2) in H.
    break_match; congruence.
Qed.

Lemma array_at'_write_success :
  forall l h a v i,
    array_at' a l h ->
    0 <= i < Zlength l ->
    write h (a + i) v <> None.
Proof.
  induction l; intros.
  - destruct H0.
    rewrite Zlength_correct in H1.
    simpl in H1.
    omega.
  - simpl.
    destruct H.
    generalize H0.
    eapply natlike_ind with (x := i); intros.
    * replace (a0 + 0) with a0 by ring.
      eapply read_write_success; eauto.
    * replace (a0 + Z.succ x) with ((a0 + 1) + x) by ring.
      eapply IHl; eauto.
      rewrite Zlength_cons in H4.
      omega.
   * omega.
Qed.

Lemma array_at_write_success :
  forall h a l v i,
    array_at a l h ->
    0 <= i < Zlength l ->
    write h (Z.succ (a + i)) v <> None.
Proof.
  unfold array_at.
  intros.
  break_match; intuition.
  rewrite <- Z.add_succ_l in *.
  eapply array_at'_write_success; eauto.
Qed.

Lemma after_write' :
  forall env s h x e1 e2 a l i v (M : _ -> Prop) (Q : store -> heap -> Prop),
    lkup s x = Some (Vaddr a) ->
    array_at a l h ->
    eval_e s h e1 (Vint i) ->
    eval_e s h e2 v ->
    0 <= i < Zlength l ->
    M (Z.succ (a + i)) ->
    Q s (write' h (Vaddr (Z.succ (a + i))) v) ->
    after env s h (Swrite x e1 e2) M Q.
Proof.
  intros.
  eapply after_write; eauto.
  eauto using array_at_read_length.
  unfold write'.
  break_match; auto.
  pose proof array_at_write_success h a l v i.
  repeat concludes. congruence.
Qed.
```

```
Lemma eval_stmt_while_intro :
  forall (I : store -> heap -> Prop) (f : store -> nat) env s h e p,
    I s h ->
    (forall s0 h0,
        I s0 h0 ->
        exists b,
          eval_e s0 h0 e (Vbool b)) ->
    (forall s0 h0,
        I s0 h0 ->
        eval_e s0 h0 e (Vbool true) ->
        exists s1 h1,
          eval_s env s0 h0 p s1 h1 /\
          I s1 h1 /\
          (f s1 < f s0)%nat) ->
    exists s' h',
      eval_s env s h (Swhile e p) s' h' /\
      I s' h' /\
      eval_e s' h' e (Vbool false).
Proof.
  intros.
  assert (forall n s h, (f s < n)%nat -> I s h ->
                  exists s1 h1,
                    eval_s env s h (Swhile e p) s1 h1 /\
                    I s1 h1 /\
                    eval_e s1 h1 e (Vbool false) /\
                    (f s1 <= f s)%nat).
  {
    induction n; intros.
    - omega.
    - copy_apply H0 H3.
      break_exists.
      destruct b.
      + copy_eapply H1 H4; eauto.
        break_exists. break_and.
        eapply IHn in H6; [|omega].
        break_exists_exists.
        intuition.
        eauto using eval_while_t.
      + eauto 10 using eval_while_f.
  }

  apply H2 with (n := S (f s)) in H; [|omega].
  break_exists_exists.
  intuition.
Qed.

Lemma after_while :
  forall env s h e p M (I Q : store -> heap -> Prop) (f : store -> nat),
    (forall s0 h0, I s0 h0 -> exists b, eval_e s0 h0 e (Vbool b)) ->
    I s h ->
    (forall s0 h0, I s0 h0 -> eval_e s0 h0 e (Vbool true) ->
              (forall z, ~ M z -> read h0 z = read h z) ->
              after env s0 h0 p M (fun s1 h1 => I s1 h1 /\ (f s1 < f s0)%nat)) -
>
    (forall s0 h0, I s0 h0 -> eval_e s0 h0 e (Vbool false) ->
              (forall z, ~ M z -> read h0 z = read h z) ->
              Q s0 h0) ->
    after env s h (Swhile e p) M Q.
Proof.
  unfold after.
  intros.
  pose proof eval_stmt_while_intro (fun s0 h0 => I s0 h0 /\ (forall z, ~ M z ->
read h0 z = read h z))
       f env s h e p.

  repeat conclude_using intuition.
  forwards.
  intuition.
  eapply H1 in H5; auto.
```



```
  break_exists_exists.
  intuition.
  eauto using eq_trans.
  concludes.
  firstorder.
Qed.

Lemma after_if :
  forall env s h e b p1 p2 M (Q : store -> heap -> Prop),
    eval_e s h e (Vbool b) ->
    (eval_e s h e (Vbool true) -> after env s h p1 M Q) ->
    (eval_e s h e (Vbool false) -> after env s h p2 M Q) ->
    after env s h (Sifelse e p1 p2) M Q.
Proof.
  unfold after.
  intros.
  destruct b; concludes; break_exists; break_and.
  all: do 2 eexists; intuition eauto using eval_ifelse_t, eval_ifelse_f.
Qed.


Lemma nth_error_nth :
  forall A (l : list A) n x d,
    nth_error l n = Some x ->
    nth n l d = x.
Proof.
  induction l; destruct n; simpl; intros; try discriminate.
  - congruence.
  - eauto.
Qed.

Lemma nth_nth_error :
  forall A (l : list A) n d,
    (0 <= n < Datatypes.length l)%nat ->
    nth_error l n = Some (nth n l d).
Proof.
  induction l; destruct n; simpl; intros.
  - omega.
  - omega.
  - auto.
  - apply IHl. omega.
Qed.

Lemma array_at'_nth_error_read :
  forall contents base h x i,
    array_at' base contents h ->
    0 <= i < Zlength contents ->
    nth_error contents (Z.to_nat i) = Some x ->
    read h (base + i) = Some (Vint x).
Proof.
  induction contents; simpl; intros.
  - rewrite Zlength_nil in *. omega.
  - break_and.
    generalize dependent i.
    refine (natlike_ind _ _ _); intros.
    + simpl in *. find_inversion. rewrite Z.add_0_r. auto.
    + rewrite Z2Nat.inj_succ in * by auto.
      simpl in *.
      rewrite <- Z.add_succ_comm.
      apply IHcontents; auto.
      rewrite Zlength_cons in *. omega.
Qed.

Lemma array_at'_nth_read :
  forall contents base h i,
    array_at' base contents h ->
    0 <= i < Zlength contents ->
    read h (base + i) = Some (Vint (nth (Z.to_nat i) contents 0)).
Proof.
```

```
  intros.
  assert (exists x, nth_error contents (Z.to_nat i) = Some x).
  {
    destruct (nth_error contents (Z.to_nat i)) eqn:?; eauto.
    exfalso.
    find_apply_lem_hyp nth_error_None.
    rewrite Zlength_correct in *.
    zify. rewrite Z2Nat.id in * by omega. omega. }
  break_exists.
  eapply array_at'_nth_error_read; eauto.
  erewrite nth_error_nth; eauto.
Qed.

Lemma array_at_read_nth :
  forall contents base h i,
    array_at base contents h ->
    0 <= i < Zlength contents ->
    read h (Z.succ (base + i)) = Some (Vint (nth (Z.to_nat i) contents 0)).
Proof.
  intros.
  rewrite <- Z.add_succ_l.
  eapply array_at'_nth_read.
  eauto using array_at_array_at'.
  auto.
Qed.


Definition Znth {A} (z : Z) (l : list A) (d : A) : A :=
  nth (Z.to_nat z) l d.

Lemma array_at_read_Znth :
  forall contents base h i,
    array_at base contents h ->
    0 <= i < Zlength contents ->
    read h (Z.succ (base + i)) = Some (Vint (Znth i contents 0)).
Proof.
  unfold Znth.
  exact array_at_read_nth.
Qed.

Ltac rewrite_lkup_update :=
      repeat (rewrite ? lkup_update_same in *; rewrite ? lkup_update_neq in * by
 discriminate).

Lemma Znth_app_2 :
  forall A (l1 l2 : list A) i d,
    Zlength l1 <= i ->
    Znth i (l1 ++ l2) d = Znth (i - Zlength l1) l2 d.
Proof.
  intros.
  unfold Znth.
  rewrite Zlength_correct.
  rewrite app_nth2 by index_crush.
  f_equal.
  index_crush.
Qed.

Lemma Znth_app_Zlength :
  forall A (l1 l2 : list A) i n d,
    n = Zlength l1 ->
    0 <= i ->
    Znth (n + i) (l1 ++ l2) d = Znth i l2 d.
Proof.
  intros.
  subst.
  rewrite Znth_app_2 by index_crush.
  f_equal.
  index_crush.
Qed.
```



```
Fixpoint nth_set {A} (l : list A) (n : nat) (v : A) : list A :=
  match l with
  | [] => []
  | x :: l => match n with
              | 0%nat => v :: l
              | S n => x :: nth_set l n v
              end
  end.

Lemma length_nth_set :
  forall A (l : list A) n v,
    List.length (nth_set l n v) = List.length l.
Proof.
  induction l; simpl; intros.
  - auto.
  - break_match.
    + auto.
    + simpl. auto using f_equal.
Qed.

Definition Znth_set {A} (l : list A) (i : Z) (v : A) : list A :=
  nth_set l (Z.to_nat i) v.


Lemma length_Znth_set :
  forall A (l : list A) i v,
    List.length (Znth_set l i v) = List.length l.
Proof.
  unfold Znth_set.
  intros.
  now rewrite length_nth_set.
Qed.
Hint Rewrite length_Znth_set : list.

Lemma Zlength_Znth_set :
  forall A (l : list A) i v,
    Zlength (Znth_set l i v) = Zlength l.
Proof.
  intros.
  index_crush.
Qed.

Lemma read_write'_neq :
  forall h a b x,
    a <> b ->
    read (write' h (Vaddr a) x) b = read h b.
Admitted.

Lemma read_write'_eq :
  forall h a b x,
    a = b ->
    read (write' h (Vaddr a) x) b = Some x.
Admitted.

Lemma read_write'_nop :
  forall h a x,
    read h a = Some x ->
    write' h (Vaddr a) x = h.
Admitted.

Lemma read_write'_same :
  forall h a x,
    read (write' h (Vaddr a) x) a = Some x.
Admitted.

Lemma Znth_set_unroll :
  forall A (a : A) l p v,
    0 <= p ->
```

```
    Znth_set (a :: l) p v =
    if Z.eq_dec p 0 then v :: l else a :: Znth_set l (Z.pred p) v.
Proof.
Admitted.

Lemma array_at'_write'_preserve :
  forall l h a i x,
    array_at' a l h ->
    ~ a <= i < a + Zlength l ->
    array_at' a l (write' h (Vaddr i) x).
Admitted.

Lemma array_at'_write' :
  forall l a_val h p v,
  array_at' a_val l h ->
  0 <= p < Zlength l ->
  array_at' a_val (Znth_set l p v) (write' h (Vaddr (a_val + p)) (Vint v)).
Proof.
  induction l; intros.
  - index_crush.
  - simpl in H. break_and.
    rewrite Znth_set_unroll by omega.
    break_if.
    + subst. cbn [array_at'].
      rewrite read_write'_eq by index_crush.
      intuition.
      apply array_at'_write'_preserve; auto.
      index_crush.
    + cbn [array_at'].
      rewrite read_write'_neq by index_crush.
      intuition.
      replace (a_val + p) with ((a_val + 1) + Z.pred p) by index_crush.
      apply IHl; auto.
      index_crush.
Qed.

Lemma array_at_write' :
  forall a_val l h p v,
  array_at a_val l h ->
  0 <= p < Zlength l ->
  array_at a_val (Znth_set l p v) (write' h (Vaddr (Z.succ (a_val + p))) (Vint v
)).
Proof.
  unfold array_at.
  intros.
  break_match_hyp; intuition.
  subst.
  rewrite read_write'_neq by index_crush.
  find_rewrite.
  intuition.
  - index_simplify. reflexivity.
  - eapply eq_rect.
    eapply array_at'_write'.
    eauto.
    omega.
    f_equal.
    f_equal.
    omega.
Qed.

Lemma nth_ext :
  forall A (l1 l2 : list A),
    List.length l1 = List.length l2 ->
    (forall n d, (0 <= n < List.length l1)%nat -> nth n l1 d = nth n l2 d) ->
    l1 = l2.
Proof.
  induction l1; simpl; intros.
  - symmetry. rewrite <- length_zero_iff_nil. auto.
  - destruct l2; try index_crush.
```



```
    simpl in *.
    f_equal.
    + specialize (H0 0%nat a).
      concludes.
      simpl in *.
      auto.
    + apply IHl1.
      * omega.
      * intros.
        specialize (H0 (S n) d).
        conclude_using omega.
        simpl in *.
        auto.
Qed.

Lemma Znth_ext :
  forall A (l1 l2 : list A),
    Zlength l1 = Zlength l2 ->
    (forall i d, 0 <= i < Zlength l1 -> Znth i l1 d = Znth i l2 d) ->
    l1 = l2.
Proof.
  intros.
  unfold Znth in *.
  rewrite !Zlength_correct in *.
  apply nth_ext.
  - index_crush.
  - intros.
    specialize (H0 (Z.of_nat n) d).
    concludes.
    rewrite Nat2Z.id in *.
    auto.
Qed.

Lemma nth_nth_set :
  forall A (l : list A) n1 n2 v d,
    (n1 < List.length l)%nat ->
    nth n2 (nth_set l n1 v) d = if Nat.eq_dec n1 n2 then v else nth n2 l d.
Proof.
  induction l; simpl; intros.
  - omega.
  - destruct n1.
    + break_if.
      * subst. auto.
      * simpl. destruct n2; auto. omega.
    + cbn [nth].
      destruct n2.
      * break_if; try omega.
        auto.
      * rewrite IHl by omega.
        repeat break_if; try omega; auto.
Qed.

Lemma Znth_Znth_set :
  forall A (l : list A) i1 i2 v d,
    0 <= i1 < Zlength l ->
    0 <= i2 ->
    Znth i2 (Znth_set l i1 v) d = if Z.eq_dec i1 i2 then v else Znth i2 l d.
Proof.
  intros.
  unfold Znth, Znth_set.
  rewrite nth_nth_set by index_crush.
  destruct (Z.eq_dec i1 i2).
  + subst. break_if; congruence.
  + break_if; try congruence.
    find_apply_lem_hyp Z2Nat.inj; try index_crush.
Qed.

Lemma Zlength_rev :
  forall A (l : list A),
```

```
    Zlength (rev l) = Zlength l.
Proof. intros. index_crush. Qed.

Lemma Znth_succ_cons :
  forall A a (l : list A) i d,
    0 <= i ->
    Znth (Z.succ i) (a :: l) d = Znth i l d.
Admitted.

Lemma Znth_app :
  forall A (l1 l2 : list A) i d,
    Znth i (l1 ++ l2) d =
    if Z_lt_dec i (Zlength l1)
    then Znth i l1 d
    else Znth (i - Zlength l1) l2 d.
Admitted.

Lemma app_cons_single :
  forall A (a : A) l,
    a :: l = ([a] ++ l)%list.
Proof. auto. Qed.

Lemma Znth_cons :
  forall A a (l : list A) i d,
    Znth i (a :: l) d =
    if Z.eq_dec i 0 then a else Znth (Z.pred i) l d.
Admitted.

Lemma Znth_cons_middle :
  forall A (l1 l2 : list A) a i d,
    i = Zlength l1 ->
    Znth i (l1 ++ a :: l2) d = a.
Admitted.

Lemma Znth_cons_0 :
  forall A (a : A) l d,
    Znth 0 (a :: l) d = a.
Proof. auto. Qed.
```