```
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 ImpInterp.
Require Import ImpInterpProof.
Require ImpInterpNock.

Module I := ImpInterp.
Module F := ImpInterpNock.

Lemma nock_op1_ok :
  forall op v v',
    I.interp_op1 op v = Some v' ->
    F.interp_op1 op v = v'.
Proof.
  unfold I.interp_op1; intros.
  repeat break_match;
    solve_by_inversion.
Qed.

Lemma nock_op2_ok :
  forall op v1 v2 v',
    I.interp_op2 op v1 v2 = Some v' ->
    F.interp_op2 op v1 v2 = v'.
Proof.
  unfold I.interp_op2; intros.
  repeat break_match;
    solve_by_inversion.
Qed.

Lemma nock_e_ok :
  forall s h e v,
    I.interp_e s h e = Some v ->
    F.interp_e s h e = v.
Proof.
  induction e; simpl; intros.
  - invc H; auto.
  - unfold F.lkup'. find_rewrite; auto.
  - break_match; try discriminate.
    erewrite IHe; eauto.
    apply nock_op1_ok; auto.
  - repeat break_match; try discriminate.
    erewrite IHe1; eauto.
    erewrite IHe2; eauto.
    apply nock_op2_ok; auto.
  - break_match_hyp; try discriminate.
    erewrite IHe; eauto.
    break_match_hyp; try discriminate.
    + find_inversion. auto.
    + break_match_hyp; try discriminate.
      break_match_hyp; try discriminate.
      find_inversion. auto.
  - break_match_hyp; try discriminate.
    erewrite IHe1; eauto.
    break_match_hyp; try discriminate.
    + break_match_hyp; try discriminate.
      erewrite IHe2; eauto.
      break_match_hyp; try discriminate.
      break_match_hyp; try discriminate.
```



```
      unfold F.strget'.
      break_match_hyp; try discriminate.
      find_inversion; auto.
    + unfold F.read'.
      repeat break_match_hyp; try discriminate.
      erewrite IHe2; eauto.
      simpl. find_rewrite; auto.
Qed.

Lemma nock_e_ok' :
  forall s h e v,
    F.interp_e s h e = v ->
    I.interp_e s h e = Some v \/
    (forall v', ~ eval_e s h e v').
Proof.
  intros; subst.
  destruct (I.interp_e s h e) eqn:?.
  + find_apply_lem_hyp nock_e_ok.
    subst; auto.
  + right; unfold not; intros.
    find_apply_lem_hyp eval_e_interp_e.
    congruence.
Qed.

Lemma nocks_e_ok :
  forall s h es vs,
    I.interps_e s h es = Some vs ->
    F.interps_e s h es = vs.
Proof.
  induction es; simpl; intros.
  - congruence.
  - repeat break_match; subst; try discriminate.
    find_inversion. f_equal; auto.
    apply nock_e_ok; auto.
Qed.

Lemma nock_updates_ok :
  forall s xs vs s',
  updates s xs vs = Some s' ->
  F.updates' s xs vs = s'.
Proof.
  intros s xs; revert s.
  induction xs; simpl; intros.
  - break_match; try discriminate.
    find_inversion; auto.
  - break_match; try discriminate.
    auto.
Qed.

Lemma nock_s_ok :
  forall env s h p s' h' p',
    I.interp_s env s h p = Some (s', h', p') ->
    F.interp_s env s h p = (s', h', p').
Proof.
  induction p; simpl; intros.
  - discriminate.
  - break_match; try discriminate.
    erewrite nock_e_ok; eauto.
    inv H; auto.
  - repeat break_match; try discriminate.
    repeat (erewrite nock_e_ok; eauto).
    invc H; auto.
  - repeat break_match_hyp; try discriminate.
    unfold F.write', F.lkup'. repeat find_rewrite.
    repeat (erewrite nock_e_ok; eauto).
    invc H. repeat find_rewrite; auto.
  - repeat break_match_hyp; try discriminate.
    + find_inversion.
      erewrite nocks_e_ok; eauto.
```

```
      erewrite nock_updates_ok; eauto.
    + find_apply_lem_hyp nocks_e_ok.
      repeat find_rewrite.
      find_inversion; auto.
  - break_match_hyp; try discriminate.
    find_apply_lem_hyp nock_e_ok.
    find_rewrite.
    break_match; try discriminate.
    break_match; find_inversion; auto.
  - break_match_hyp; try discriminate.
    find_apply_lem_hyp nock_e_ok.
    find_rewrite.
    break_match; try discriminate.
    break_match; find_inversion; auto.
  - repeat break_match; subst; try discriminate.
    + find_inversion; auto.
    + find_inversion; auto.
      specialize (IHp1 s' h' s2).
      forward IHp1; auto.
      find_apply_hyp_hyp. find_inversion; auto.
  - repeat break_match; subst; try discriminate.
    + find_inversion; auto.
      erewrite nock_e_ok; eauto.
    + find_inversion; auto.
      specialize (IHp s' h' s4).
      forward IHp; auto.
      find_apply_hyp_hyp. find_inversion; auto.
Qed.

Lemma nocks_s_done_ok :
  forall env n s h p ret h' v,
    I.interps_p n env s h p ret = Done h' v ->
    F.interps_p n env s h p ret = Done h' v.
Proof.
  induction n; simpl; intros.
  - discriminate.
  - repeat break_match_hyp; try discriminate.
    + find_inversion.
      erewrite nock_e_ok; eauto.
    + erewrite nock_s_ok; eauto.
Qed.

Lemma nocks_s_timeout_ok :
  forall env n s h p ret s' h' p',
    I.interps_p n env s h p ret = Timeout s' h' p' ret ->
    F.interps_p n env s h p ret = Timeout s' h' p' ret.
Proof.
  induction n; simpl; intros.
  - find_inversion; auto.
  - repeat break_match_hyp; try discriminate.
    erewrite nock_s_ok; eauto.
Qed.

Lemma nock_p_done_ok :
  forall n funcs main ret h' v,
    I.interp_p n (Prog funcs main ret) = Done h' v ->
    F.interp_p n (Prog funcs main ret) = Done h' v.
Proof.
  unfold I.interp_p, F.interp_p; intros.
  apply nocks_s_done_ok; auto.
Qed.

Lemma nock_p_timeout_ok :
  forall n funcs main ret s' h' p',
    I.interp_p n (Prog funcs main ret) = Timeout s' h' p' ret ->
    F.interp_p n (Prog funcs main ret) = Timeout s' h' p' ret.
Proof.
  unfold I.interp_p, F.interp_p; intros.
  apply nocks_s_timeout_ok; auto.
```



```
Qed.
```