```
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 ImpExprTransf.
Require Import ImpInterpNock.
Require Import ImpConstFold.

Require Import ImpEval.
Require Import ImpStep.
Require Import ImpSemanticsFacts.
Require Import ImpInterpProof.
Require Import ImpInterpNockProof.

Section PROOF.

Variable transf : expr -> expr.

Variable transf_fwd :
  forall s h e v,
    eval_e s h e v ->
    eval_e s h (transf e) v.

Variable transf_bwd :
  forall s h e v,
    eval_e s h (transf e) v ->
    eval_e s h e v \/
    (forall v', ~ eval_e s h e v').

Lemma transf_e_fwd :
  forall s h e v,
    eval_e s h e v ->
    eval_e s h (transf_e transf e) v.
Proof.
  induction e; simpl; intros;
    apply transf_fwd; auto.
  - on (eval_e _ _ _ _), invc.
    find_copy_apply_hyp_hyp. ee.
  - on (eval_e _ _ _ _), invc.
    do 2 find_copy_apply_hyp_hyp. ee.
  - on (eval_e _ _ _ _), invc.
    + find_copy_apply_hyp_hyp. ee.
    + find_copy_apply_hyp_hyp.
      eapply eval_len_s; eauto.
  - on (eval_e _ _ _ _), invc.
    + find_copy_apply_hyp_hyp. ee.
    + find_copy_apply_hyp_hyp.
      eapply eval_idx_s; eauto.
Qed.

Lemma transf_e_bwd :
  forall s h e v,
    eval_e s h (transf_e transf e) v ->
    eval_e s h e v \/
    (forall v', ~ eval_e s h e v').
Proof.
  induction e; simpl; intros;
    find_apply_hyp_hyp;
    on (or _ _), invc; auto.
  - on (eval_e _ _ _ _), inv.
    find_apply_hyp_hyp.
    on (or _ _), invc; auto.
```



```
    + left; ee.
    + right; unfold not in *; intros.
      on (eval_e _ _ _ _), inv. firstorder.
  - right; unfold not in *; intros.
    on (eval_e _ _ _ _), inv.
    eapply H0; eauto. ee.
    eapply transf_e_fwd; eauto.
  - on (eval_e _ _ _ _), invc.
    repeat find_apply_hyp_hyp.
    repeat on (or _ _), invc; auto.
    + left; ee.
    + right; unfold not in *; intros.
      on (eval_e _ _ _ _), inv. firstorder.
    + right; unfold not in *; intros.
      on (eval_e _ _ _ _), inv. firstorder.
    + right; unfold not in *; intros.
      on (eval_e _ _ _ _), inv. firstorder.
  - right; unfold not in *; intros.
    on (eval_e _ _ _ _), inv.
    eapply H0; eauto. ee.
    + eapply transf_e_fwd; eauto.
    + eapply transf_e_fwd; eauto.
  (** len and idx are copy paste *)
  - on (eval_e _ _ _ _), invc.
    + find_apply_hyp_hyp.
      on (or _ _), invc; auto.
      * left; ee.
      * right; unfold not in *; intros.
        on (eval_e _ _ _ _), inv.
        firstorder. firstorder.
    + find_apply_hyp_hyp.
      on (or _ _), invc; auto.
      * left; eapply eval_len_s; eauto.
      * right; unfold not in *; intros.
        on (eval_e _ _ _ _), inv.
        firstorder. firstorder.
  - right; unfold not in *; intros.
    on (eval_e _ _ _ _), inv.
    + eapply H0; eauto. ee.
      eapply transf_e_fwd; eauto.
    + eapply H0; eauto.
      eapply eval_len_s; eauto.
      eapply transf_e_fwd; eauto.
  - on (eval_e _ _ _ _), invc.
    + repeat find_apply_hyp_hyp.
      repeat on (or _ _), invc; auto.
      * left; ee.
      * right; unfold not in *; intros.
        on (eval_e _ _ _ _), inv.
        firstorder. firstorder.
      * right; unfold not in *; intros.
        on (eval_e _ _ _ _), inv.
        firstorder. firstorder.
      * right; unfold not in *; intros.
        on (eval_e _ _ _ _), inv.
        firstorder. firstorder.
    + repeat find_apply_hyp_hyp.
      repeat on (or _ _), invc; auto.
      * left; eapply eval_idx_s; eauto.
      * right; unfold not in *; intros.
        on (eval_e _ _ _ _), inv.
        firstorder. firstorder.
      * right; unfold not in *; intros.
        on (eval_e _ _ _ _), inv.
        firstorder. firstorder.
      * right; unfold not in *; intros.
        on (eval_e _ _ _ _), inv.
        firstorder. firstorder.
  - right; unfold not in *; intros.
```

```
    on (eval_e _ _ _ _), inv.
    + eapply H0; eauto. ee.
      * eapply transf_e_fwd; eauto.
      * eapply transf_e_fwd; eauto.
    + eapply H0; eauto.
      eapply eval_idx_s; eauto.
      * eapply transf_e_fwd; eauto.
      * eapply transf_e_fwd; eauto.
Qed.

Lemma transfs_e_fwd :
  forall s h es vs,
    evals_e s h es vs ->
    evals_e s h (List.map transf es) vs.
Proof.
  induction es; simpl; intros.
  - auto.
  - on (evals_e _ _ _ _), invc.
    find_apply_hyp_hyp. repeat ee.
Qed.

Lemma transfs_e_bwd :
  forall s h es vs,
    evals_e s h (List.map transf es) vs ->
    evals_e s h es vs \/
    (forall vs', ~ evals_e s h es vs').
Proof.
  induction es; simpl; intros.
  - auto.
  - on (evals_e _ _ _ _), invc.
    find_apply_hyp_hyp.
    on (or _ _), invc; auto.
    + find_apply_lem_hyp transf_bwd.
      on (or _ _), invc; auto.
      * left; ee.
      * right; unfold not in *; intros.
        on (evals_e _ _ _ _), invc.
        eapply H0; eauto.
    + right; unfold not in *; intros.
      on (evals_e _ _ _ _), invc.
      eapply H; eauto.
Qed.

Lemma locate_some_transf :
  forall env x f,
    locate env x = Some f ->
    locate (transf_env transf env) x =
      Some (transf_f transf f).
Proof.
  induction env; simpl; intros.
  - discriminate.
  - repeat break_match; subst.
    + congruence.
    + simpl in *. find_inversion. congruence.
    + simpl in *. find_inversion. congruence.
    + auto.
Qed.

Lemma locate_none_transf :
  forall env x,
    locate env x = None ->
    locate (transf_env transf env) x = None.
Proof.
  induction env; simpl; intros.
  - auto.
  - repeat break_match; subst.
    + congruence.
    + simpl in *. find_inversion. congruence.
    + simpl in *. find_inversion. congruence.
```



```
    + auto.
Qed.

Lemma transf_locate_some :
  forall env x f',
    locate (transf_env transf env) x = Some f' ->
    exists f,
      locate env x = Some f /\
      transf_f transf f = f'.
Proof.
  induction env; simpl; intros.
  - discriminate.
  - repeat break_match; subst.
    + find_inversion; repeat ee.
    + simpl in *; find_inversion; congruence.
    + simpl in *; find_inversion; congruence.
    + auto.
Qed.

Lemma transf_locate_none :
  forall env x,
    locate (transf_env transf env) x = None ->
    locate env x = None.
Proof.
  induction env; simpl; intros.
  - auto.
  - repeat break_match; subst.
    + congruence.
    + simpl in *. find_inversion. congruence.
    + simpl in *. find_inversion. congruence.
    + auto.
Qed.

Lemma transf_s_fwd :
  forall env s1 h1 p1 s2 h2 p2,
    step env
      s1 h1 p1
      s2 h2 p2 ->
    step (transf_env transf env)
      s1 h1 (transf_s transf p1)
      s2 h2 (transf_s transf p2).
Proof.
  induction 1; simpl; intros.
  - repeat ee; apply transf_e_fwd; auto.
  - repeat ee; apply transf_e_fwd; auto.
  - repeat ee; apply transf_e_fwd; auto.
  - repeat ee.
    + find_apply_lem_hyp locate_some_transf; auto.
    + apply transfs_e_fwd; auto.
  - repeat ee.
    + find_apply_lem_hyp locate_none_transf; auto.
    + eapply transfs_e_fwd; eauto.
  - repeat ee; apply transf_e_fwd; auto.
  - repeat ee; apply transf_e_fwd; auto.
  - repeat ee; apply transf_e_fwd; auto.
  - repeat ee; apply transf_e_fwd; auto.
  - repeat ee; apply transf_e_fwd; auto.
  - repeat ee; apply transf_e_fwd; auto.
  - repeat ee; apply transf_e_fwd; auto.
  - repeat ee; apply transf_e_fwd; auto.
Qed.

(** Need to slightly strengthen IH for env locate,
   and add lame equalities b/c prep_induction
   does not work well with sections(?). *)
Lemma transf_s_bwd' :
  forall env' s1 h1 p1' s2 h2 p2,
    step env'
      s1 h1 p1'
```

```
      s2 h2 p2 ->
  forall env p1,
    env' = transf_env transf env ->
    p1' = transf_s transf p1 ->
    (exists p,
      step env
        s1 h1 p1
        s2 h2 p
      /\ transf_s transf p = p2) \/
    (forall s2 h2 p,
      p1 <> Snop /\
      ~ step env
        s1 h1 p1
        s2 h2 p).
Proof.
  induction 1; intros; subst.
  - destruct p1; simpl in *; try discriminate.
    repeat find_inversion.
    find_apply_lem_hyp transf_bwd.
    on (or _ _), invc; [left | right].
    + repeat ee.
    + unfold not in *; intros.
      split; [congruence | intros].
      on (step _ _ _ _ _ _ _), invc.
      firstorder.
  - destruct p1; simpl in *; try discriminate.
    repeat find_inversion.
    repeat find_apply_lem_hyp transf_bwd.
    on (or _ _), invc; [|right].
    on (or _ _), invc; [left | right].
    + repeat ee.
    + unfold not in *; intros.
      split; [congruence | intros].
      on (step _ _ _ _ _ _ _), invc.
      firstorder.
    + unfold not in *; intros.
      split; [congruence | intros].
      on (step _ _ _ _ _ _ _), invc.
      firstorder.
  - destruct p1; simpl in *; try discriminate.
    repeat find_inversion.
    repeat find_apply_lem_hyp transf_bwd.
    on (or _ _), invc; [|right].
    on (or _ _), invc; [left | right].
    + repeat ee.
    + unfold not in *; intros.
      split; [congruence | intros].
      on (step _ _ _ _ _ _ _), invc.
      firstorder.
    + unfold not in *; intros.
      split; [congruence | intros].
      on (step _ _ _ _ _ _ _), invc.
      firstorder.
  - destruct p1; simpl in *; try discriminate.
    find_apply_lem_hyp transf_locate_some.
    break_exists; break_and.
    repeat find_inversion.
    repeat find_apply_lem_hyp transfs_e_bwd.
    on (or _ _), invc; [left |right].
    + destruct f0; simpl in *.
      repeat find_inversion. repeat ee.
    + unfold not in *; intros.
      split; [congruence | intros].
      on (step _ _ _ _ _ _ _), invc.
      * firstorder.
      * firstorder.
  - destruct p1; simpl in *; try discriminate.
    find_apply_lem_hyp transf_locate_none.
    repeat find_inversion.
```



```
    repeat find_apply_lem_hyp transfs_e_bwd.
    on (or _ _), invc; [left |right].
    + repeat ee.
    + unfold not in *; intros.
      split; [congruence | intros].
      on (step _ _ _ _ _ _ _), invc.
      * firstorder.
      * firstorder.
  - destruct p0; simpl in *; try discriminate.
    repeat find_inversion.
    find_apply_lem_hyp transf_bwd.
    on (or _ _), invc; [left | right].
    + repeat ee.
    + unfold not in *; intros.
      split; [congruence | intros].
      on (step _ _ _ _ _ _ _), invc.
      * firstorder.
      * firstorder.
  - destruct p0; simpl in *; try discriminate.
    repeat find_inversion.
    find_apply_lem_hyp transf_bwd.
    on (or _ _), invc; [left | right].
    + repeat ee.
    + unfold not in *; intros.
      split; [congruence | intros].
      on (step _ _ _ _ _ _ _), invc.
      * firstorder.
      * firstorder.
  - destruct p1; simpl in *; try discriminate.
    repeat find_inversion.
    find_apply_lem_hyp transf_bwd.
    on (or _ _), invc; [left | right].
    + repeat ee.
    + unfold not in *; intros.
      split; [congruence | intros].
      on (step _ _ _ _ _ _ _), invc.
      * firstorder.
      * firstorder.
  - destruct p1; simpl in *; try discriminate.
    repeat find_inversion.
    find_apply_lem_hyp transf_bwd.
    on (or _ _), invc; [left | right].
    + exists Snop; repeat ee.
    + unfold not in *; intros.
      split; [congruence | intros].
      on (step _ _ _ _ _ _ _), invc.
      * firstorder.
      * firstorder.
  - destruct p1; simpl in *; try discriminate.
    repeat find_inversion.
    destruct p1_1; simpl in *; try discriminate.
    left; repeat ee.
  - destruct p0; simpl in *; try discriminate.
    repeat find_inversion.
    edestruct IHstep; eauto.
    + break_exists; break_and.
      left; repeat ee.
      subst; auto.
    + right; unfold not in *; intros.
      split; [congruence | intros].
      on (step _ _ _ _ _ _ _), inv; simpl in *.
      * inversion H.
      * eapply H0; eauto.
  - destruct p1; simpl in *; try discriminate.
    repeat find_inversion.
    destruct p1; simpl in *; try discriminate.
    find_apply_lem_hyp transf_bwd.
    on (or _ _), invc; [left | right].
    + exists Snop; repeat ee.
```

```
    + unfold not in *; intros.
      split; [congruence | intros].
      on (step _ _ _ _ _ _ _), invc.
      * firstorder.
      * inv H11.
  - destruct p1; simpl in *; try discriminate.
    repeat find_inversion.
    edestruct IHstep; eauto.
    + break_exists; break_and.
      left; repeat ee.
      subst; auto.
    + right; unfold not in *; intros.
      split; [congruence | intros].
      on (step _ _ _ _ _ _ _), inv; simpl in *.
      * inversion H.
      * eapply H0; eauto.
Qed.

Lemma transf_s_bwd :
  forall env s1 h1 p1 s2 h2 p2,
    step (transf_env transf env)
      s1 h1 (transf_s transf p1)
      s2 h2 p2 ->
    (exists p,
      step env
        s1 h1 p1
        s2 h2 p
      /\ transf_s transf p = p2) \/
    (forall s2 h2 p,
      p1 <> Snop /\
      ~ step env
        s1 h1 p1
        s2 h2 p).
Proof.
  intros.
  eapply transf_s_bwd'; eauto.
Qed.

Lemma transfs_s_fwd :
  forall env s1 h1 p1 s2 h2 p2,
    step_star env
      s1 h1 p1
      s2 h2 p2 ->
    step_star (transf_env transf env)
      s1 h1 (transf_s transf p1)
      s2 h2 (transf_s transf p2).
Proof.
  induction 1.
  - repeat ee.
  - find_apply_lem_hyp transf_s_fwd.
    repeat ee.
Qed.

Inductive can_get_stuck :
  env -> store -> heap -> stmt -> Prop :=
| cgs_stuck :
    forall env s1 h1 p1,
      p1 <> Snop ->
      (forall s2 h2 p2,
        ~ step env
            s1 h1 p1
            s2 h2 p2) ->
      can_get_stuck env s1 h1 p1
| cgs_step :
    forall env s1 h1 p1 s2 h2 p2,
      step env
        s1 h1 p1
        s2 h2 p2 ->
      can_get_stuck env s2 h2 p2 ->
      can_get_stuck env s1 h1 p1.
```



```
Lemma transfs_s_bwd :
  forall env' s1 h1 p1' s2 h2 p2,
    step_star env'
      s1 h1 p1'
      s2 h2 p2 ->
  forall env p1,
    env' = transf_env transf env ->
    p1' = transf_s transf p1 ->
    (exists p,
      step_star env
        s1 h1 p1
        s2 h2 p
      /\ transf_s transf p = p2) \/
    can_get_stuck env s1 h1 p1.
Proof.
  induction 1; intros; subst.
  - left; repeat ee.
  - find_apply_lem_hyp transf_s_bwd.
    on (or _ _), invc.
    + break_exists; break_and.
      edestruct IHstep_star; eauto.
      * break_exists; break_and.
        left; repeat ee.
      * right. eapply cgs_step; eauto.
    + right. ee.
      * firstorder.
      * firstorder.
Qed.

Lemma transf_p_fwd :
  forall p v,
    steps_p p v ->
    steps_p (transf_p transf p) v.
Proof.
  destruct p; intros.
  on (steps_p _ _), invc. ee.
  change Snop
    with (transf_s transf Snop).
  eapply transfs_s_fwd; eauto.
Qed.

Inductive can_get_stuck_prog : prog -> Prop :=
| cgsp_body :
    forall funcs main ret,
      can_get_stuck funcs store_0 heap_0 main ->
      can_get_stuck_prog (Prog funcs main ret)
| cgsp_ret :
    forall funcs main ret s2 h2,
      step_star funcs
        store_0 heap_0 main
        s2 h2 Snop ->
      (forall v,
        ~ eval_e s2 h2 ret v) ->
      can_get_stuck_prog (Prog funcs main ret).

Lemma transf_p_bwd :
  forall p v,
    steps_p (transf_p transf p) v ->
    steps_p p v \/ can_get_stuck_prog p.
Proof.
  destruct p; intros.
  on (steps_p _ _), invc.
  find_eapply_lem_hyp transfs_s_bwd; eauto.
  on (or _ _), invc.
  - break_exists; break_and.
    destruct p; simpl in *; try discriminate.
    find_apply_lem_hyp transf_bwd.
```

```
    on (or _ _), invc.
    + left; ee.
    + right. eapply cgsp_ret; eauto.
  - right; ee.
Qed.

End PROOF.
```