```
(*********************************************************************************************************)
(* HaskStrongToProof: convert HaskStrong to HaskProof                                                    *)
(*********************************************************************************************************)

Generalizable All Variables.
Require Import Preamble.
Require Import General.
Require Import NaturalDeduction.
Require Import Coq.Strings.String.
Require Import Coq.Lists.List.
Require Import Coq.Init.Specif.
Require Import HaskKinds.
Require Import HaskStrongTypes.
Require Import HaskStrong.
Require Import HaskProof.

Section HaskStrongToProof.

Definition pivotContext {T} a b c : @Arrange T ((a,,b),,c) ((a,,c),,b) :=
  RComp (RComp (RCossa _ _ _) (RLeft a (RExch c b))) (RAssoc _ _ _).

Definition copyAndPivotContext {T} a b c : @Arrange T ((a,,b),,(c,,b)) ((a,,c),,b).
  eapply RComp; [ idtac | apply (RLeft (a,,c) (RCont b)) ].
  eapply RComp; [ idtac | apply RCossa ].
  eapply RComp; [ idtac | apply (RRight b (pivotContext a b c)) ].
  apply RAssoc.
  Defined.

Context {VV:Type}{eqd_vv:EqDecidable VV}.

(* maintenance of Xi *)
Fixpoint dropVar (lv:list VV)(v:VV) : ??VV :=
  match lv with
    | nil     => Some v
    | v'::lv' => if eqd_dec v v' then None else dropVar lv' v
  end.

Fixpoint mapOptionTree' {a b:Type}(f:a->??b)(t:@Tree ??a) : @Tree ??b :=
  match t with
    | T_Leaf None     => T_Leaf None
    | T_Leaf (Some x) => T_Leaf (f x)
    | T_Branch l r    => T_Branch (mapOptionTree' f l) (mapOptionTree' f r)
  end.

  (* later: use mapOptionTreeAndFlatten *)
  Definition stripOutVars (lv:list VV) : Tree ??VV -> Tree ??VV :=
    mapOptionTree' (dropVar lv).

Lemma In_both : forall T (l1 l2:list T) a, In a l1 -> In a (app l1 l2).
  intros T l1.
  induction l1; intros.
  inversion H.
  simpl.
  inversion H; subst.
  left; auto.
  right.
  apply IHl1.
  apply H0.
  Qed.

Lemma In_both' : forall T (l1 l2:list T) a, In a l2 -> In a (app l1 l2).
  intros T l1.
  induction l1; intros.
  apply H.
  rewrite <- app_comm_cons.
  simpl.
  right.
  apply IHl1.
  auto.
  Qed.

Lemma distinct_app : forall T (l1 l2:list T), distinct (app l1 l2) -> distinct l1 /\ distinct l2.
  intro T.
  intro l1.
  induction l1; intros.
  split; auto.
  apply distinct_nil.
  simpl in H.
  inversion H.
  subst.
  set (IHl1 _ H3) as H3'.
  destruct H3'.
  split; auto.
  apply distinct_cons; auto.
  intro.
  apply H2.
  apply In_both; auto.
  Qed.

Lemma mapOptionTree'_compose : forall T A B (t:Tree ??T) (f:T->??A)(g:A->??B),
  mapOptionTree' g (mapOptionTree' f t)
  =
  mapOptionTree' (fun x => match f x with None => None | Some x => g x end) t.
  intros; induction t.
    destruct a; auto.
    simpl.
    destruct (f t); reflexivity.
    simpl.
    rewrite <- IHt1.
    rewrite <- IHt2.
    reflexivity.
    Qed.

Lemma strip_lemma a x t : stripOutVars (a::x) t = stripOutVars (a::nil) (stripOutVars x t).
  unfold stripOutVars.
  rewrite mapOptionTree'_compose.
  simpl.
  induction t.
  destruct a0.
  simpl.
  induction x.
  reflexivity.
  simpl.
  destruct (eqd_dec v a0).
    destruct (eqd_dec v a); reflexivity.
    apply IHx.
  reflexivity.
  simpl.
  rewrite <- IHt1.
  rewrite <- IHt2.
  reflexivity.
  Qed.
```

```
Lemma strip_twice_lemma x y t : stripOutVars x (stripOutVars y t) = stripOutVars (app y x) t.
(*
  induction x.
  simpl.
  unfold stripOutVars.
  simpl.
  rewrite mapOptionTree'_compose.
  induction t.
  destruct a; try reflexivity.
  simpl.
  destruct (dropVar y v); reflexivity.
  simpl.
  rewrite IHt1.
  rewrite IHt2.
  reflexivity.
  rewrite strip_lemma.
  rewrite IHx.
  rewrite <- strip_lemma.
  rewrite app_comm_cons.
  reflexivity.
*)
  admit.
  Qed.

Lemma strip_distinct a y : (not (In a (leaves y))) -> stripOutVars (a :: nil) y = y.
  intros.
  induction y.
  destruct a0; try reflexivity.
  simpl in *.
  unfold stripOutVars.
  simpl.
  destruct (eqd_dec v a).
  subst.
  assert False.
  apply H.
  left; auto.
  inversion H0.
  auto.
  rewrite <- IHy1 at 2.
  rewrite <- IHy2 at 2.
  reflexivity.
  unfold not; intro.
  apply H.
  eapply In_both' in H0.
  apply H0.
  unfold not; intro.
  apply H.
  eapply In_both in H0.
  apply H0.
  Qed.

Lemma drop_distinct x v : (not (In v x)) -> dropVar x v = Some v.
  intros.
  induction x.
  reflexivity.
  simpl.
  destruct (eqd_dec v a).
  subst.
  assert False. apply H.
  simpl; auto.
  inversion H0.
  apply IHx.
  unfold not.
  intro.
  apply H.
  simpl; auto.
  Qed.

Lemma in3 {T}(a b c:list T) q : In q (app a c) -> In q (app (app a b) c).
  induction a; intros.
  simpl.
  simpl in H.
  apply In_both'.
  auto.
  rewrite <- ass_app.
  rewrite <- app_comm_cons.
  simpl.
  rewrite ass_app.
  rewrite <- app_comm_cons in H.
  inversion H.
  left; auto.
  right.
  apply IHa.
  apply H0.
  Qed.

Lemma distinct3 {T}(a b c:list T) : distinct (app (app a b) c) -> distinct (app a c).
  induction a; intros.
  simpl in *.
  apply distinct_app in H; auto.
  destruct H; auto.
  rewrite <- app_comm_cons.
  apply distinct_cons.
  rewrite <- ass_app in H.
  rewrite <- app_comm_cons in H.
  inversion H.
  subst.
  intro q.
  apply H2.
  rewrite ass_app.
  apply in3.
  auto.
  apply IHa.
  rewrite <- ass_app.
  rewrite <- ass_app in H.
  rewrite <- app_comm_cons in H.
  inversion H.
  subst.
  auto.
  Qed.

Lemma strip_distinct' y : forall x, distinct (app x (leaves y)) -> stripOutVars x y = y.
  induction x; intros.
  simpl in H.
  unfold stripOutVars.
  simpl.
  induction y; try destruct a; auto.
  simpl.
  rewrite IHy1.
  rewrite IHy2.
  reflexivity.
  simpl in H.
  apply distinct_app in H; destruct H; auto.
  apply distinct_app in H; destruct H; auto.
  rewrite <- app_comm_cons in H.
  inversion H; subst.
  set (IHx H3) as qq.
  rewrite strip_lemma.
  rewrite IHx.
  apply strip_distinct.
  unfold not; intros.
  apply H2.
  apply In_both'.
  auto.
  auto.
  Qed.

Lemma updating_stripped_tree_is_inert'
  {Γ} lev
  (ξ:VV -> LeveledHaskType Γ ★)
  lv tree2 :
    mapOptionTree (update_ξ ξ lev lv) (stripOutVars (map (@fst _ _) lv) tree2)
  = mapOptionTree ξ (stripOutVars (map (@fst _ _) lv) tree2).
  induction tree2.
  destruct a.
  simpl.
  induction lv.
  reflexivity.
  simpl.
  destruct a.
  simpl.
  set (eqd_dec v v0) as q.
  destruct q.
  auto.
  set (dropVar (map (@fst _ _) lv) v) as b in *.
  destruct b.
  inversion IHlv.
  admit.
  auto.
  reflexivity.
  simpl.
  unfold stripOutVars in *.
  rewrite <- IHtree2_1.
  rewrite <- IHtree2_2.
  reflexivity.
  Qed.

Lemma update_ξ_lemma `{EQD_VV:EqDecidable VV} : forall Γ ξ (lev:HaskLevel Γ)(varstypes:Tree ??(VV*_)),
  distinct (map (@fst _ _) (leaves varstypes)) ->
  mapOptionTree (update_ξ ξ lev (leaves varstypes)) (mapOptionTree (@fst _ _) varstypes) =
  mapOptionTree (fun t => t@@ lev) (mapOptionTree (@snd _ _) varstypes).
  admit.
  Qed.




Fixpoint expr2antecedent {Γ'}{Δ'}{ξ'}{τ'}(exp:Expr Γ' Δ' ξ' τ') : Tree ??VV :=
  match exp as E in Expr Γ Δ ξ τ with
  | EGlobal  Γ Δ ξ _ _                         => []
  | EVar     Γ Δ ξ ev                          => [ev]
  | ELit     Γ Δ ξ lit lev                     => []
  | EApp     Γ Δ ξ t1 t2 lev e1 e2             => (expr2antecedent e1),,(expr2antecedent e2)
  | ELam     Γ Δ ξ t1 t2 lev v    e            => stripOutVars (v::nil) (expr2antecedent e)
  | ELet     Γ Δ ξ tv t  lev v ev ebody        => ((stripOutVars (v::nil) (expr2antecedent ebody)),,(expr2antecedent ev))
  | EEsc     Γ Δ ξ ec t lev e                  => expr2antecedent e
  | EBrak    Γ Δ ξ ec t lev e                  => expr2antecedent e
  | ECast    Γ Δ ξ γ t1 t2 lev      e          => expr2antecedent e
  | ENote    Γ Δ ξ t n e                       => expr2antecedent e
  | ETyLam   Γ Δ ξ κ σ l e                     => expr2antecedent e
  | ECoLam   Γ Δ κ σ σ₁ σ₂ ξ l             e   => expr2antecedent e
  | ECoApp   Γ Δ κ γ σ₁ σ₂ σ ξ l      e        => expr2antecedent e
  | ETyApp   Γ Δ κ σ τ ξ  l    e               => expr2antecedent e
  | ELetRec  Γ Δ ξ l τ vars branches body      =>
      let branch_context := eLetRecContext branches
   in let all_contexts := (expr2antecedent body),,branch_context
   in     stripOutVars (leaves (mapOptionTree (@fst _ _ ) vars)) all_contexts
  | ECase    Γ Δ ξ l tc tbranches atypes e' alts =>
    ((fix varsfromalts (alts:
               Tree ??{ sac : _ & { scb : StrongCaseBranchWithVVs _ _ tc atypes sac
                              & Expr (sac_Γ sac Γ)
                                     (sac_Δ sac Γ atypes (weakCK'' Δ))
                                     (scbwv_ξ scb ξ l)
                                     (weakLT' (tbranches@@l)) } }
      ): Tree ??VV :=
      match alts with
        | T_Leaf None     => []
        | T_Leaf (Some h) => stripOutVars (vec2list (scbwv_exprvars (projT1 (projT2 h)))) (expr2antecedent (projT2 (projT2 h)))
        | T_Branch b1 b2  => (varsfromalts b1),,(varsfromalts b2)
      end) alts),,(expr2antecedent e')
end
with eLetRecContext {Γ}{Δ}{ξ}{lev}{tree}(elrb:ELetRecBindings Γ Δ ξ lev tree) : Tree ??VV :=
match elrb with
  | ELR_nil    Γ Δ ξ lev  => []
  | ELR_leaf   Γ Δ ξ lev v t e => expr2antecedent e
  | ELR_branch Γ Δ ξ lev t1 t2 b1 b2 => (eLetRecContext b1),,(eLetRecContext b2)
end.

Definition mkProofCaseBranch {Γ}{Δ}{ξ}{l}{tc}{tbranches}{atypes}
(alt : { sac : _ & { scb : StrongCaseBranchWithVVs _ _ tc atypes sac
                            & Expr (sac_Γ sac Γ)
                                   (sac_Δ sac Γ atypes (weakCK'' Δ))
                                   (scbwv_ξ scb ξ l)
                                   (weakLT' (tbranches@@l)) } })
  : { sac : _ & ProofCaseBranch tc Γ Δ l tbranches atypes sac }.
  destruct alt.
  exists x.
  exact
    {| pcb_freevars := mapOptionTree ξ
      (stripOutVars (vec2list (scbwv_exprvars (projT1 s)))
        (expr2antecedent (projT2 s)))
     |}.
     Defined.


Fixpoint eLetRecTypes {Γ}{Δ}{ξ}{lev}{τ}(elrb:ELetRecBindings Γ Δ ξ lev τ) : Tree ??(HaskType Γ ★) :=
  match elrb with
  | ELR_nil    Γ Δ ξ lev  => []
  | ELR_leaf   Γ Δ ξ  lev  v t e => [t]
  | ELR_branch Γ Δ ξ lev t1 t2 b1 b2 => (eLetRecTypes b1),,(eLetRecTypes b2)
  end.

Lemma stripping_nothing_is_inert
  {Γ:TypeEnv}
  (ξ:VV -> LeveledHaskType Γ ★)
  tree :
  stripOutVars nil tree = tree.
  induction tree.
    simpl; destruct a; reflexivity.
    unfold stripOutVars.
    fold stripOutVars.
    simpl.
    fold (stripOutVars nil).
    rewrite <- IHtree1 at 2.
    rewrite <- IHtree2 at 2.
    reflexivity.
    Qed.

Definition arrangeContext
     (Γ:TypeEnv)(Δ:CoercionEnv Γ)
     v      (* variable to be pivoted, if found *)
     ctx    (* initial context *)
     (ξ:VV -> LeveledHaskType Γ ★)
     :

    (* a proof concluding in a context where that variable does not appear *)
     sum (Arrange
          (mapOptionTree ξ                        ctx                        )
          (mapOptionTree ξ (stripOutVars (v::nil) ctx),,[]                   ))

    (* or a proof concluding in a context where that variable appears exactly once in the left branch *)
        (Arrange
          (mapOptionTree ξ                         ctx                       )
          (mapOptionTree ξ ((stripOutVars (v::nil) ctx),,[v])                )).

  induction ctx.

    refine (match a with None => let case_None := tt in _ | Some v' => let case_Some := tt in _ end).

        (* nonempty leaf *)
        destruct case_Some.
          unfold stripOutVars in *; simpl.
          unfold dropVar.
          unfold mapOptionTree in *; simpl; fold (mapOptionTree ξ) in *.

          destruct (eqd_dec v' v); subst.
        (* where the leaf is v *)
        apply inr.
          subst.
          apply RuCanL.

        (* where the leaf is NOT v *)
        apply inl.
          apply RuCanR.

    (* empty leaf *)
    destruct case_None.
      apply inl; simpl in *.
      apply RuCanR.

  (* branch *)
  refine (
    match IHctx1 with
      | inr lpf =>
        match IHctx2 with
          | inr rpf => let case_Both := tt in _
          | inl rpf => let case_Left := tt in _
        end
      | inl lpf =>
        match IHctx2 with
          | inr rpf => let case_Right   := tt in _
          | inl rpf => let case_Neither := tt in _
        end
    end); clear IHctx1; clear IHctx2.

destruct case_Neither.
  apply inl.
  eapply RComp; [idtac | apply RuCanR ].
    exact (RComp
      (* order will not matter because these are central as morphisms *)
      (RRight _ (RComp lpf (RCanR _)))
      (RLeft  _ (RComp rpf (RCanR _)))).


destruct case_Right.
  apply inr.
```

```
      fold (stripOutVars (v::nil)).
      set (RRight (mapOptionTree ξ ctx2)  (RComp lpf ((RCanR _)))) as q.
      simpl in *.
      eapply RComp.
      apply q.
      clear q.
      clear lpf.
      unfold mapOptionTree in *; simpl; fold (mapOptionTree ξ) in *.
      eapply RComp; [ idtac | apply RAssoc ].
      apply RLeft.
      apply rpf.

    destruct case_Left.
      apply inr.
      unfold mapOptionTree in *; simpl; fold (mapOptionTree ξ) in *.
      fold (stripOutVars (v::nil)).
      eapply RComp; [ idtac | eapply pivotContext ].
      set (RComp rpf (RCanR _ )) as rpf'.
      set (RLeft ((mapOptionTree ξ (stripOutVars (v :: nil) ctx1),, [ξ v])) rpf') as qq.
      simpl in *.
      eapply RComp; [ idtac | apply qq ].
      clear qq rpf' rpf.
      apply (RRight (mapOptionTree ξ ctx2)).
      apply lpf.

    destruct case_Both.
      apply inr.
      unfold mapOptionTree in *; simpl; fold (mapOptionTree ξ) in *.
      fold (stripOutVars (v::nil)).
      eapply RComp; [ idtac | eapply copyAndPivotContext ].
        (* order will not matter because these are central as morphisms *)
        exact (RComp (RRight _ lpf) (RLeft _ rpf)).

    Defined.

(* same as before, but use RWeak if necessary *)
Definition arrangeContextAndWeaken
     (Γ:TypeEnv)(Δ:CoercionEnv Γ)
     v      (* variable to be pivoted, if found *)
     ctx    (* initial context *)
     (ξ:VV -> LeveledHaskType Γ ★) :
       Arrange
          (mapOptionTree ξ                         ctx                  )
          (mapOptionTree ξ ((stripOutVars (v::nil) ctx),,[v])           ).
  set (arrangeContext Γ Δ v ctx ξ) as q.
  destruct q; auto.
  eapply RComp; [ apply a | idtac ].
  refine (RLeft _ (RWeak _)).
  Defined.

Lemma cheat : forall {T}(a b:list T), distinct (app a b) -> distinct (app b a).
  admit.
  Qed.

Definition arrangeContextAndWeaken''
     (Γ:TypeEnv)(Δ:CoercionEnv Γ)
     v      (* variable to be pivoted, if found *)
     (ξ:VV -> LeveledHaskType Γ ★) : forall ctx,
  distinct (leaves v) ->
  Arrange
    ((mapOptionTree ξ ctx)                                       )
    ((mapOptionTree ξ (stripOutVars (leaves v) ctx)),,(mapOptionTree ξ v)).

  induction v; intros.
    destruct a.
    unfold mapOptionTree in *.
    simpl in *.
    fold (mapOptionTree ξ) in *.
    intros.
    apply arrangeContextAndWeaken.
    apply Δ.

  unfold mapOptionTree; simpl in *.
    intros.
    rewrite (@stripping_nothing_is_inert Γ); auto.
    apply RuCanR.
    intros.
    unfold mapOptionTree in *.
    simpl in *.
    fold (mapOptionTree ξ) in *.
    set (mapOptionTree ξ) as X in *.
    set (IHv2 ((stripOutVars (leaves v1) ctx),, v1)) as IHv2'.
    unfold stripOutVars in IHv2'.
    simpl in IHv2'.
    fold (stripOutVars (leaves v2)) in IHv2'.
    fold (stripOutVars (leaves v1)) in IHv2'.
    unfold X in IHv2'.
    unfold mapOptionTree in IHv2'.
    simpl in IHv2'.
    fold (mapOptionTree ξ) in IHv2'.
    fold X in IHv2'.
    set (distinct_app _ _ _ H) as H'.
    destruct H' as [H1 H2].
    set (RComp (IHv1 _ H1) (IHv2' H2)) as qq.
    eapply RComp.
      apply qq.
      clear qq IHv2' IHv2 IHv1.
      rewrite strip_twice_lemma.

      rewrite (strip_distinct' v1 (leaves v2)).
        apply RCossa.
        apply cheat.
        auto.
    Defined.

Lemma updating_stripped_tree_is_inert {Γ} (ξ:VV -> LeveledHaskType Γ ★) v tree t lev :
      mapOptionTree (update_ξ ξ lev ((v,t)::nil)) (stripOutVars (v :: nil) tree)
    = mapOptionTree ξ (stripOutVars (v :: nil) tree).
  set (@updating_stripped_tree_is_inert' Γ lev ξ ((v,t)::nil)) as p.
  rewrite p.
  simpl.
  reflexivity.
  Qed.

(* TODO: use multi-conclusion proofs instead *)
Inductive LetRecSubproofs Γ Δ ξ lev : forall tree, ELetRecBindings Γ Δ ξ lev tree -> Type :=
  | lrsp_nil  : LetRecSubproofs Γ Δ ξ lev [] (ELR_nil _ _ _ _)
  | lrsp_leaf : forall v t e ,
    (ND Rule [] [Γ > Δ > mapOptionTree ξ (expr2antecedent e) |- [t@@lev]]) ->
    LetRecSubproofs Γ Δ ξ lev [(v, t)] (ELR_leaf _ _ _ _ _ t e)
  | lrsp_cons : forall t1 t2 b1 b2,
    LetRecSubproofs Γ Δ ξ lev t1 b1 ->
```

```
    LetRecSubproofs Γ Δ ξ lev t2 b2 ->
    LetRecSubproofs Γ Δ ξ lev (t1,,t2) (ELR_branch _ _ _ _ _ _ b1 b2).

Lemma letRecSubproofsToND Γ Δ ξ lev tree branches :
  LetRecSubproofs Γ Δ ξ lev tree branches ->
    ND Rule [] [ Γ > Δ > mapOptionTree ξ (eLetRecContext branches)
      |- (mapOptionTree (@snd _ _) tree) @@@ lev ].
  intro X; induction X; intros; simpl in *.
    apply nd_rule.
      apply RVoid.
    set (ξ v) as q in *.
      destruct q.
      simpl in *.
      apply n.
    eapply nd_comp; [ idtac | eapply nd_rule; apply RJoin ].
    eapply nd_comp; [ apply nd_llecnac | idtac ].
    apply nd_prod; auto.
  Defined.

Lemma letRecSubproofsToND' Γ Δ ξ lev τ tree  :
    forall branches body,
    distinct (leaves (mapOptionTree (@fst _ _) tree)) ->
    ND Rule [] [Γ > Δ > mapOptionTree (update_ξ ξ lev (leaves tree)) (expr2antecedent body) |- [τ @@ lev]] ->
    LetRecSubproofs Γ Δ (update_ξ ξ lev (leaves tree)) lev tree branches ->
    ND Rule [] [Γ > Δ > mapOptionTree ξ (expr2antecedent (@ELetRec VV _ Γ Δ ξ lev τ tree branches body)) |- [τ @@ lev]].

  (* NOTE: how we interpret stuff here affects the order-of-side-effects *)
  intro branches.
  intro body.
  intro disti.
  intro pf.
  intro lrsp.

  rewrite mapleaves in disti.
  set (@update_ξ_lemma _ Γ ξ lev tree disti) as ξlemma.
    rewrite <- mapOptionTree_compose in ξlemma.

  set ((update_ξ ξ lev (leaves tree))) as ξ' in *.
  set ((stripOutVars (leaves (mapOptionTree (@fst _ _) tree)) (eLetRecContext branches))) as ctx.
  set (mapOptionTree (@fst _ _) tree) as pctx.
  set (mapOptionTree ξ' pctx) as passback.
  set (fun a b => @RLetRec Γ Δ a b (mapOptionTree (@snd _ _) tree)) as z.
  eapply nd_comp; [ idtac | eapply nd_rule; apply z ].
  clear z.

  set (@arrangeContextAndWeaken''  Γ Δ  pctx ξ' (expr2antecedent body,,eLetRecContext branches)) as q'.
  unfold passback in *; clear passback.
  unfold pctx in *; clear pctx.
  rewrite <- mapleaves in disti.
  set (q' disti) as q''.

  unfold ξ' in *.
  set (@updating_stripped_tree_is_inert' Γ lev ξ (leaves tree)) as zz.
  rewrite <- mapleaves in zz.
  rewrite zz in q''.
  clear zz.
  clear ξ'.
  Opaque stripOutVars.
  simpl.
  rewrite <- mapOptionTree_compose in q''.
  rewrite <- ξlemma.
  eapply nd_comp; [ idtac | eapply nd_rule; apply (RArrange _ _ _ _ _ q'') ].
  clear q'.
  clear q''.
  simpl.

  set (letRecSubproofsToND _ _ _ _ _ branches lrsp) as q.
    eapply nd_comp; [ idtac | eapply nd_rule; apply RJoin ].
    eapply nd_comp; [ apply nd_llecnac | idtac ].
    apply nd_prod; auto.
    rewrite ξlemma.
    apply q.
    Defined.

Lemma scbwv_coherent {tc}{Γ}{atypes:IList _ (HaskType Γ) _}{sac} :
  forall scb:StrongCaseBranchWithVVs _ _ tc atypes sac,
    forall l ξ,
      vec2list (vec_map (scbwv_ξ scb ξ l) (scbwv_exprvars scb)) =
      vec2list (vec_map (fun t => t @@ weakL' l) (sac_types sac _ atypes)).
  intros.
  unfold scbwv_ξ.
  unfold scbwv_varstypes.
  set (@update_ξ_lemma _ _ (weakLT' ○ ξ) (weakL' l)
    (unleaves (vec2list (vec_zip (scbwv_exprvars scb) (sac_types sac Γ atypes))))
    ) as q.
  rewrite <- mapleaves' in q.
  rewrite <- mapleaves' in q.
  rewrite <- mapleaves' in q.
  rewrite <- mapleaves' in q.
  set (fun z => unleaves_injective _ _ _ (q z)) as q'.
  rewrite vec2list_map_list2vec in q'.
  rewrite fst_zip in q'.
  rewrite vec2list_map_list2vec in q'.
  rewrite vec2list_map_list2vec in q'.
  rewrite snd_zip in q'.
  rewrite leaves_unleaves in q'.
  rewrite vec2list_map_list2vec in q'.
  rewrite vec2list_map_list2vec in q'.
  apply q'.
  rewrite fst_zip.
  apply scbwv_exprvars_distinct.
  Qed.


Lemma case_lemma : forall Γ Δ ξ l tc tbranches atypes e
   (alts':Tree
           ??{sac : StrongAltCon &
             {scb : StrongCaseBranchWithVVs VV eqd_vv tc atypes sac &
             Expr (sac_Γ sac Γ) (sac_Δ sac Γ atypes (weakCK'' Δ))
               (scbwv_ξ scb ξ l) (weakLT' (tbranches @@  l))}}),

      (mapOptionTreeAndFlatten (fun x => pcb_freevars (projT2 x))
        (mapOptionTree mkProofCaseBranch alts'))
    ,,
    mapOptionTree ξ  (expr2antecedent e) =
  mapOptionTree ξ
        (expr2antecedent (ECase Γ Δ ξ l tc tbranches atypes e alts')).
  intros.
  simpl.
  Ltac hack := match goal with [ |- ?A,,?B = ?C,,?D ] => assert (A=C) end.
  hack.
  induction alts'.
  destruct a; simpl.
  destruct s; simpl.
  unfold mkProofCaseBranch.
  reflexivity.
  reflexivity.
  simpl.
  rewrite IHalts'1.
  rewrite IHalts'2.
  reflexivity.
  rewrite H.
  reflexivity.
  Qed.

Definition expr2proof  :
  forall Γ Δ ξ τ (e:Expr Γ Δ ξ τ),
    ND Rule [] [Γ > Δ > mapOptionTree ξ (expr2antecedent e) |- [τ]].

  refine (fix expr2proof Γ' Δ' ξ' τ' (exp:Expr Γ' Δ' ξ' τ') {struct exp}
    : ND Rule [] [Γ' > Δ' > mapOptionTree ξ' (expr2antecedent exp) |- [τ']] :=
    match exp as E in Expr Γ Δ ξ τ with
    | EGlobal  Γ Δ ξ t wev                          => let case_EGlobal := tt in _
    | EVar     Γ Δ ξ ev                             => let case_EVar := tt in _
    | ELit     Γ Δ ξ lit lev                        => let case_ELit := tt in _
    | EApp     Γ Δ ξ t1 t2 lev e1 e2                => let case_EApp := tt in
                                                        (fun e1' e2' => _) (expr2proof _ _ _ _ e1) (expr2proof _ _ _ _ e2)
    | ELam     Γ Δ ξ t1 t2 lev v    e               => let case_ELam := tt in (fun e' => _) (expr2proof _ _ _ _ e)
    | ELet     Γ Δ ξ tv t      v lev ev ebody       => let case_ELet := tt in
                                                       (fun pf_let pf_body => _) (expr2proof _ _ _ _ ev) (expr2proof _ _ _ _ ebody)
    | ELetRec  Γ Δ ξ lev t tree branches ebody      =>
      let ξ' := update_ξ ξ lev (leaves tree) in
      let case_ELetRec := tt in  (fun e' subproofs => _) (expr2proof _ _ _ _ ebody)
        ((fix subproofs Γ'' Δ'' ξ'' lev'' (tree':Tree ??(VV * HaskType Γ'' ★))
        (branches':ELetRecBindings Γ'' Δ'' ξ'' lev'' tree')
        : LetRecSubproofs Γ'' Δ'' ξ'' lev'' tree' branches' :=
        match branches' as B in ELetRecBindings G D X L T return LetRecSubproofs G D X L T B with
          | ELR_nil    Γ Δ ξ lev  => lrsp_nil _ _ _ _
          | ELR_leaf   Γ Δ ξ l v t e => lrsp_leaf Γ Δ ξ l v t e (expr2proof _ _ _ _ e)
          | ELR_branch Γ Δ ξ lev t1 t2 b1 b2 => lrsp_cons _ _ _ _ _ _ _ _ _ (subproofs _ _ _ _ _ b1) (subproofs _ _ _ _ _ b2)
        end
        ) _ _ _ _ tree branches)
    | EEsc     Γ Δ ξ ec t lev e                     => let case_EEsc    := tt in (fun e' => _) (expr2proof _ _ _ _ e)
    | EBrak    Γ Δ ξ ec t lev e                     => let case_EBrak   := tt in (fun e' => _) (expr2proof _ _ _ _ e)
  | ECast    Γ Δ ξ γ t1 t2 lev      e             => let case_ECast   := tt in (fun e' => _) (expr2proof _ _ _ _ e)
  | ENote    Γ Δ ξ t n e                          => let case_ENote   := tt in (fun e' => _) (expr2proof _ _ _ _ e)
  | ETyLam   Γ Δ ξ κ σ l e                        => let case_ETyLam  := tt in (fun e' => _) (expr2proof _ _ _ _ e)
  | ECoLam   Γ Δ κ σ σ₁ σ₂ ξ l              e      => let case_ECoLam  := tt in (fun e' => _) (expr2proof _ _ _ _ e)
  | ECoApp   Γ Δ κ σ₁ σ₂ σ γ ξ l      e           => let case_ECoApp  := tt in (fun e' => _) (expr2proof _ _ _ _ e)
  | ETyApp   Γ Δ κ σ τ ξ  l    e                  => let case_ETyApp  := tt in (fun e' => _) (expr2proof _ _ _ _ e)
  | ECase    Γ Δ ξ l tc tbranches atypes e alts' =>
    let dcsp :=
      ((fix mkdcsp (alts:
             Tree ??{ sac : _ & { scb : StrongCaseBranchWithVVs _ _ tc atypes sac
                    & Expr (sac_Γ sac Γ)
                           (sac_Δ sac Γ atypes (weakCK'' Δ))
                           (scbwv_ξ scb ξ l)
                           (weakLT' (tbranches@@l)) } })
        : ND Rule [] (mapOptionTree (fun x => pcb_judg (projT2 (mkProofCaseBranch x))) alts) :=
      match alts as ALTS return ND Rule []
        (mapOptionTree (fun x => pcb_judg (projT2 (mkProofCaseBranch x))) ALTS) with
        | T_Leaf None      => let case_nil := tt in _
        | T_Branch b1 b2   => let case_branch := tt in (fun b1' b2' => _) (mkdcsp b1) (mkdcsp b2)
        | T_Leaf (Some x)  =>
          match x as X return ND Rule [] [pcb_judg (projT2 (mkProofCaseBranch X))] with
          existT sac (existT scbx ex) =>
          (fun e' => let case_leaf := tt in _) (expr2proof _ _ _ _ ex)
      end
      end) alts')
      in let case_ECase := tt in (fun e' => _) (expr2proof _ _ _ _ e)
  end
); clear exp ξ' τ' Γ' Δ' expr2proof; try clear mkdcsp.

  destruct case_EGlobal.
    apply nd_rule.
    simpl.
    destruct t as [t lev].
    apply (RGlobal _ _ _ _ wev).

  destruct case_EVar.
    apply nd_rule.
    unfold mapOptionTree; simpl.
    destruct (ξ ev).
    apply RVar.
```

```
destruct case_ELit.
  apply nd_rule.
  apply RLit.

destruct case_EApp.
  unfold mapOptionTree; simpl; fold (mapOptionTree ξ).
  eapply nd_comp; [ idtac | eapply nd_rule; apply RApp ].
  eapply nd_comp; [ apply nd_llecnac | idtac ].
  apply nd_prod; auto.
  apply e1'.
  apply e2'.

destruct case_ELam; intros.
  unfold mapOptionTree; simpl; fold (mapOptionTree ξ).
  eapply nd_comp; [ idtac | eapply nd_rule; apply RLam ].
  set (update_ξ ξ lev ((v,t1)::nil)) as ξ'.
  set (arrangeContextAndWeaken Γ Δ v (expr2antecedent e) ξ') as pfx.
    eapply RArrange in pfx.
    unfold mapOptionTree in pfx; simpl in pfx.
    unfold ξ' in pfx.
    rewrite updating_stripped_tree_is_inert in pfx.
    unfold update_ξ in pfx.
    destruct (eqd_dec v v).
    eapply nd_comp; [ idtac | apply (nd_rule pfx) ].
    clear pfx.
    apply e'.
    assert False.
    apply n.
    auto.
    inversion H.

destruct case_ELet; intros; simpl in *.
  eapply nd_comp; [ idtac | eapply nd_rule; eapply RLet ].
  eapply nd_comp; [ apply nd_llecnac | idtac ].
  apply nd_prod.
    apply pf_let.
    clear pf_let.
    eapply nd_comp; [ apply pf_body | idtac ].
    clear pf_body.
  fold (@mapOptionTree VV).
  fold (mapOptionTree ξ).
  set (update_ξ ξ v ((lev,tv)::nil)) as ξ'.
  set (arrangeContextAndWeaken Γ Δ lev (expr2antecedent ebody) ξ') as n.
  unfold mapOptionTree in n; simpl in n; fold (mapOptionTree ξ') in n.
  unfold ξ' in n.
  rewrite updating_stripped_tree_is_inert in n.
  unfold update_ξ in n.
  destruct (eqd_dec lev lev).
  unfold ξ'.
  unfold update_ξ.
  eapply RArrange in n.
  apply (nd_rule n).
  assert False. apply n0; auto. inversion H.

destruct case_EEsc.
  eapply nd_comp; [ idtac | eapply nd_rule; apply REsc ].
  apply e'.

destruct case_EBrak; intros.
  eapply nd_comp; [ idtac | eapply nd_rule; apply RBrak ].
  apply e'.

destruct case_ECast.
  eapply nd_comp; [ idtac | eapply nd_rule; eapply RCast ].
  apply e'.
  auto.

destruct case_ENote.
  destruct t.
  eapply nd_comp; [ idtac | eapply nd_rule; apply RNote ].
  apply e'.
  auto.

destruct case_ETyApp; simpl in *; intros.
  eapply nd_comp; [ idtac | eapply nd_rule; apply RAppT ].
  apply e'.
  auto.

destruct case_ECoLam; simpl in *; intros.
  eapply nd_comp; [ idtac | eapply nd_rule; apply RAbsCo with (κ:=κ) ].
  apply e'.
```

```
destruct case_ECoApp; simpl in *; intros.
  eapply nd_comp; [ idtac | eapply nd_rule; apply (@RAppCo _ _ (mapOptionTree ξ (expr2antecedent e)) _ σ₁ σ₂ σ γ l) ].
  apply e'.
  auto.

destruct case_ETyLam; intros.
  eapply nd_comp; [ idtac | eapply nd_rule; apply RAbsT ].
  unfold mapOptionTree in e'.
  rewrite mapOptionTree_compose in e'.
  unfold mapOptionTree.
  apply e'.

destruct case_leaf.
  clear o x alts alts' e.
  eapply nd_comp; [ apply e' | idtac ].
  clear e'.
  apply nd_rule.
  apply RArrange.
  simpl.
  rewrite mapleaves'.
  simpl.
  rewrite <- mapOptionTree_compose.
  unfold scbwv_ξ.
  rewrite <- mapleaves'.
  rewrite vec2list_map_list2vec.
  unfold sac_Γ.
  rewrite <- (scbwv_coherent scbx l ξ).
  rewrite <- vec2list_map_list2vec.
  rewrite mapleaves'.
  set (@arrangeContextAndWeaken'') as q.
  unfold scbwv_ξ.
  set (@updating_stripped_tree_is_inert' _ (weakL' l) (weakLT' ○ ξ) (vec2list (scbwv_varstypes scbx))) as z.
  unfold scbwv_varstypes in z.
  rewrite vec2list_map_list2vec in z.
  rewrite fst_zip in z.
  rewrite <- z.
  clear z.
  replace (stripOutVars (vec2list (scbwv_exprvars scbx))) with
    (stripOutVars (leaves (unleaves (vec2list (scbwv_exprvars scbx))))).
  apply q.
  apply (sac_Δ sac Γ atypes (weakCK'' Δ)).
```

```
      rewrite leaves_unleaves.
      apply (scbwv_exprvars_distinct scbx).
      rewrite leaves_unleaves.
      reflexivity.

    destruct case_nil.
      apply nd_id0.

    destruct case_branch.
      simpl; eapply nd_comp; [ apply nd_llecnac | idtac ].
      apply nd_prod.
      apply b1'.
      apply b2'.

    destruct case_ECase.
    set (@RCase Γ Δ l tc) as q.
      rewrite <- case_lemma.
      eapply nd_comp; [ idtac | eapply nd_rule; eapply RCase ].
      eapply nd_comp; [ apply nd_llecnac | idtac ]; apply nd_prod.
      rewrite <- mapOptionTree_compose.
      apply dcsp.
      apply e'.

    destruct case_ELetRec; intros.
      unfold ξ'0 in *.
      clear ξ'0.
      unfold ξ'1 in *.
      clear ξ'1.
      apply letRecSubproofsToND'.
      admit.
      apply e'.
      apply subproofs.

  Defined.

End HaskStrongToProof.
```