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

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 HaskProofToStrong.

  Context {VV:Type} {eqdec_vv:EqDecidable VV} {freshM:FreshMonad VV}.

  Definition fresh := FMT_fresh freshM.
  Definition FreshM := FMT freshM.
  Definition FreshMon := FMT_Monad freshM.
  Existing Instance FreshMon.

  Definition ExprVarResolver Γ := VV -> LeveledHaskType Γ ★.

  Definition judg2exprType (j:Judg) : Type :=
    match j with
      (Γ > Δ > Σ |- τ) => forall (ξ:ExprVarResolver Γ) vars, Σ = mapOptionTree ξ vars ->
        FreshM (ITree _ (fun t => Expr Γ Δ ξ t) τ)
      end.

  Definition justOne Γ Δ ξ τ : ITree _ (fun t => Expr Γ Δ ξ t) [τ] -> Expr Γ Δ ξ τ.
    intros.
    inversion X; auto.
    Defined.

  Definition ileaf `(it:ITree X F [t]) : F t.
    inversion it.
    apply X0.
```

```
  Defined.

Lemma update_branches : forall Γ (ξ:VV -> LeveledHaskType Γ ★) lev l1 l2 q,
  update_ξ ξ lev (app l1 l2) q = update_ξ (update_ξ ξ lev l2) lev l1 q.
  intros.
  induction l1.
    reflexivity.
    simpl.
    destruct a; simpl.
    rewrite IHl1.
    reflexivity.
    Qed.

 Lemma quark {T} (l1:list T) l2 vf :
    (In vf (app l1 l2)) <->
     (In vf l1) \/ (In vf l2).
   induction l1.
   simpl; auto.
   split; intro.
   right; auto.
   inversion H.
   inversion H0.
   auto.
   split.

   destruct IHl1.
   simpl in *.
   intro.
   destruct H1.
   left; left; auto.
   set (H H1) as q.
   destruct q.
   left; right; auto.
   right; auto.
   simpl.

   destruct IHl1.
   simpl in *.
   intro.
   destruct H1.
   destruct H1.
```

```
   left; auto.
   right; apply H0; auto.
   right; apply H0; auto.
 Qed.

Lemma splitter {T} (l1:list T) l2 vf :
    (In vf (app l1 l2) →False)
    -> (In vf l1 →False)  /\ (In vf l2 →False).
  intros.
  split; intros; apply H; rewrite quark.
  auto.
  auto.
  Qed.

Lemma helper
  : forall T Z {eqdt:EqDecidable T}(tl:Tree ??T)(vf:T) ξ (q:Z),
    (In vf (leaves tl) -> False) ->
    mapOptionTree (fun v' => if eqd_dec vf v' then q else ξ v') tl =
    mapOptionTree ξ tl.
  intros.
  induction tl;
    try destruct a;
      simpl in *.
  set (eqd_dec vf t) as x in *.
  destruct x.
  subst.
    assert False.
    apply H.
    left; auto.
    inversion H0.
  auto.
  auto.
  apply splitter in H.
  destruct H.
  rewrite (IHtl1 H).
  rewrite (IHtl2 H0).
  reflexivity.
  Qed.

Lemma fresh_lemma'' Γ
  : forall types ξ lev,
```

```
  FreshM { varstypes : _
    |  mapOptionTree (update_ξ(Γ:=Γ)   ξ lev (leaves varstypes)) (mapOptionTree (@fst _ _) varstypes) = (types @@@ lev)
    /\ distinct (leaves (mapOptionTree (@fst _ _) varstypes)) }.
admit.
Defined.

Lemma fresh_lemma' Γ
  : forall types vars Σ ξ lev, Σ = mapOptionTree ξ vars ->
  FreshM { varstypes : _
    |  mapOptionTree (update_ξ(Γ:=Γ) ξ lev (leaves varstypes)) vars = Σ
    /\ mapOptionTree (update_ξ       ξ lev (leaves varstypes)) (mapOptionTree (@fst _ _) varstypes) = (types @@@ lev)
    /\ distinct (leaves (mapOptionTree (@fst _ _) varstypes)) }.
  induction types.
    intros; destruct a.
      refine (bind vf = fresh (leaves vars) ; return _).
        apply FreshMon.
        destruct vf as [ vf vf_pf ].
        exists [(vf,h)].
        split; auto.
        simpl.
        set (helper VV _ vars vf ξ (h@@lev) vf_pf) as q.
        rewrite q.
        symmetry; auto.
        simpl.
        destruct (eqd_dec vf vf); [ idtac | set (n (refl_equal _)) as n'; inversion n' ]; auto.
        split; auto.
        apply distinct_cons.
        intro.
        inversion H0.
        apply distinct_nil.
      refine (return _).
        exists []; auto.
        split.
        simpl.
        symmetry; auto.
        split.
        simpl.
        reflexivity.
        simpl.
        apply distinct_nil.
      intros vars Σ ξ lev pf; refine (bind x2 = IHtypes2 vars Σ ξ lev pf; _).
```

```
        apply FreshMon.
        destruct x2 as [vt2 [pf21 [pf22 pfdist]]].
        refine (bind x1 = IHtypes1 (vars,,(mapOptionTree (@fst _ _) vt2)) (Σ,,(types2@@@lev)) (update_ξ ξ lev
          (leaves vt2)) _ _; return _).
        apply FreshMon.
        simpl.
        rewrite pf21.
        rewrite pf22.
        reflexivity.
        clear IHtypes1 IHtypes2.
        destruct x1 as [vt1 [pf11 pf12]].
        exists (vt1,,vt2); split; auto.

        set (update_branches Γ ξ lev (leaves vt1) (leaves vt2)) as q.
        set (mapOptionTree_extensional _ _ q) as q'.
        rewrite q'.
        clear q' q.
        inversion pf11.
        reflexivity.

        simpl.
        set (update_branches Γ ξ lev (leaves vt1) (leaves vt2)) as q.
        set (mapOptionTree_extensional _ _ q) as q'.
        rewrite q'.
        rewrite q'.
        clear q' q.
        rewrite <- mapOptionTree_compose.
        rewrite <- mapOptionTree_compose.
        rewrite <- mapOptionTree_compose in *.
        split.
        destruct pf12.
        rewrite H.
        inversion pf11.
        rewrite <- mapOptionTree_compose.
        reflexivity.

        admit.
      Defined.

Lemma fresh_lemma Γ ξ vars Σ Σ' lev
  : Σ = mapOptionTree ξ vars ->
```

```
  FreshM { vars' : _
    |  mapOptionTree (update_ξ(Γ:=Γ) ξ lev ((vars',Σ')::nil)) vars = Σ
    /\ mapOptionTree (update_ξ ξ lev ((vars',Σ')::nil)) [vars'] = [Σ' @@ lev] }.
  intros.
  set (fresh_lemma' Γ [Σ'] vars Σ ξ lev H) as q.
  refine (q >>>= fun q' => return _).
  apply FreshMon.
  clear q.
  destruct q' as [varstypes [pf1 [pf2 pfdist]]].
  destruct varstypes; try destruct o; try destruct p; simpl in *.
    destruct (eqd_dec v v); [ idtac | set (n (refl_equal _)) as n'; inversion n' ].
    inversion pf2; subst.
    exists v.
    destruct (eqd_dec v v); [ idtac | set (n (refl_equal _)) as n'; inversion n' ].
    split; auto.
    inversion pf2.
    inversion pf2.
  Defined.

 Definition ujudg2exprType Γ (ξ:ExprVarResolver Γ)(Δ:CoercionEnv Γ) Σ τ : Type :=
  forall vars, Σ = mapOptionTree ξ vars -> FreshM (ITree _ (fun t => Expr Γ Δ ξ t) τ).

 Definition urule2expr  : forall Γ Δ h j t (r:@Arrange _ h j) (ξ:VV -> LeveledHaskType Γ ★),
  ujudg2exprType Γ ξ Δ h t ->
  ujudg2exprType Γ ξ Δ j t
  .
  intros Γ Δ.
    refine (fix urule2expr h j t (r:@Arrange _ h j) ξ {struct r} :
  ujudg2exprType Γ ξ Δ h t ->
  ujudg2exprType Γ ξ Δ j t :=
      match r as R in Arrange H C return
  ujudg2exprType Γ ξ Δ H t ->
  ujudg2exprType Γ ξ Δ C t
with
        | RLeft   h c ctx r => let case_RLeft  := tt in (fun e => _) (urule2expr _ _ _ r)
        | RRight  h c ctx r => let case_RRight := tt in (fun e => _) (urule2expr _ _ _ r)
        | RCanL   a       => let case_RCanL  := tt in _
        | RCanR   a       => let case_RCanR  := tt in _
        | RuCanL  a       => let case_RuCanL := tt in _
        | RuCanR  a       => let case_RuCanR := tt in _
        | RAssoc  a b c   => let case_RAssoc := tt in _
```

```
    | RCossa  a b c   => let case_RCossa := tt in _
    | RExch   a b     => let case_RExch  := tt in _
    | RWeak   a       => let case_RWeak  := tt in _
    | RCont   a       => let case_RCont  := tt in _
    | RComp   a b c f g => let case_RComp  := tt in (fun e1 e2 => _) (urule2expr _ _ _ f) (urule2expr _ _ _ g)
    end); clear urule2expr; intros.

destruct case_RCanL.
  simpl; unfold ujudg2exprType; intros.
  simpl in X.
  apply (X ([],,vars)).
  simpl; rewrite <- H; auto.

destruct case_RCanR.
  simpl; unfold ujudg2exprType; intros.
  simpl in X.
  apply (X (vars,,[])).
  simpl; rewrite <- H; auto.

destruct case_RuCanL.
  simpl; unfold ujudg2exprType; intros.
  destruct vars; try destruct o; inversion H.
  simpl in X.
  apply (X vars2); auto.

destruct case_RuCanR.
  simpl; unfold ujudg2exprType; intros.
  destruct vars; try destruct o; inversion H.
  simpl in X.
  apply (X vars1); auto.

destruct case_RAssoc.
  simpl; unfold ujudg2exprType; intros.
  simpl in X.
  destruct vars; try destruct o; inversion H.
  destruct vars1; try destruct o; inversion H.
  apply (X (vars1_1,,(vars1_2,,vars2))).
  subst; auto.

destruct case_RCossa.
  simpl; unfold ujudg2exprType; intros.
```

```
  simpl in X.
  destruct vars; try destruct o; inversion H.
  destruct vars2; try destruct o; inversion H.
  apply (X ((vars1,,vars2_1),,vars2_2)).
  subst; auto.

destruct case_RExch.
  simpl; unfold ujudg2exprType ; intros.
  simpl in X.
  destruct vars; try destruct o; inversion H.
  apply (X (vars2,,vars1)).
  inversion H; subst; auto.

destruct case_RWeak.
  simpl; unfold ujudg2exprType; intros.
  simpl in X.
  apply (X []).
  auto.

destruct case_RCont.
  simpl; unfold ujudg2exprType ; intros.
  simpl in X.
  apply (X (vars,,vars)).
  simpl.
  rewrite <- H.
  auto.

destruct case_RLeft.
  intro vars; unfold ujudg2exprType; intro H.
  destruct vars; try destruct o; inversion H.
  apply (fun q => e ξ q vars2 H2).
  clear r0 e H2.
    simpl in X.
    simpl.
    unfold ujudg2exprType.
    intros.
    apply X with (vars:=vars1,,vars).
    rewrite H0.
    rewrite H1.
    simpl.
    reflexivity.
```

```
    destruct case_RRight.
      intro vars; unfold ujudg2exprType; intro H.
      destruct vars; try destruct o; inversion H.
      apply (fun q => e ξ q vars1 H1).
      clear r0 e H2.
        simpl in X.
        simpl.
        unfold ujudg2exprType.
        intros.
        apply X with (vars:=vars,,vars2).
        rewrite H0.
        inversion H.
        simpl.
        reflexivity.

    destruct case_RComp.
      apply e2.
      apply e1.
      apply X.
      Defined.

Definition letrec_helper Γ Δ l (varstypes:Tree ??(VV * HaskType Γ ★)) ξ' :
  ITree (LeveledHaskType Γ ★)
       (fun t : LeveledHaskType Γ ★=> Expr Γ Δ ξ' t)
       (mapOptionTree (ξ' ○ (@fst _ _)) varstypes)
       -> ELetRecBindings Γ Δ ξ' l varstypes.
  intros.
  induction varstypes.
  destruct a; simpl in *.
  destruct p.
  simpl.
  apply ileaf in X. simpl in X.
    apply ELR_leaf.
    rename h into τ.
    destruct (eqd_dec (unlev (ξ' v)) τ).
    rewrite <- e.
    destruct (ξ' v).
    simpl.
    destruct (eqd_dec h0 l).
      rewrite <- e0.
```

```
    apply X.
  apply (Prelude_error "level mismatch; should never happen").
  apply (Prelude_error "letrec type mismatch; should never happen").

apply ELR_nil.
apply ELR_branch.
  apply IHvarstypes1; inversion X; auto.
  apply IHvarstypes2; inversion X; auto.
Defined.

Definition unindex_tree {V}{F} : forall {t:Tree ??V}, ITree V F t -> Tree ??{ v:V & F v }.
  refine (fix rec t it := match it as IT return Tree ??{ v:V & F v } with
    | INone => T_Leaf None
    | ILeaf x y => T_Leaf (Some _)
    | IBranch _ _ b1 b2 => (rec _ b1),,(rec _ b2)
        end).
  exists x; auto.
  Defined.

Definition fix_indexing X (F:X->Type)(J:X->Type)(t:Tree ??{ x:X & F x })
  :  ITree { x:X & F x } (fun x => J (projT1 x))                                t
  -> ITree X             (fun x:X => J x)   (mapOptionTree (@projT1 _ _) t).
  intro it.
  induction it; simpl in *.
  apply INone.
  apply ILeaf.
  apply f.
  simpl; apply IBranch; auto.
  Defined.

Definition fix2 {X}{F} : Tree ??{ x:X & FreshM (F x) } -> Tree ??(FreshM { x:X & F x }).
  refine (fix rec t := match t with
    | T_Leaf None => T_Leaf None
    | T_Leaf (Some x) => T_Leaf (Some _)
    | T_Branch b1 b2 => T_Branch (rec b1) (rec b2)
        end).
  destruct x as [x fx].
  refine (bind fx' = fx ; return _).
  apply FreshMon.
  exists x.
  apply fx'.
```

```
  Defined.

Definition case_helper tc Γ Δ lev tbranches avars ξ :
  forall pcb:{sac : StrongAltCon & ProofCaseBranch tc Γ Δ lev tbranches avars sac},
   prod (judg2exprType (pcb_judg (projT2 pcb))) {vars' : Tree ??VV & pcb_freevars (projT2 pcb) = mapOptionTree ξ vars'} ->
   ((fun sac => FreshM
     { scb : StrongCaseBranchWithVVs VV eqdec_vv tc avars sac
       & Expr (sac_Γ sac Γ) (sac_Δ sac Γ avars (weakCK'' Δ)) (scbwv_ξ scb ξ lev) (weakLT' (tbranches @@  lev)) }) (projT1 pcb)).
  intro pcb.
  intro X.
  simpl in X.
  simpl.
  destruct pcb as [sac pcb].
  simpl in *.

  destruct X.
  destruct s as [vars vars_pf].

  refine (bind localvars = fresh_lemma' _ (unleaves  (vec2list (sac_types sac _ avars))) vars
    (mapOptionTree weakLT' (pcb_freevars pcb)) (weakLT' ○ ξ) (weakL' lev) _  ; _).
    apply FreshMon.
    rewrite vars_pf.
    rewrite <- mapOptionTree_compose.
    reflexivity.
    destruct localvars as [localvars [localvars_pf1 [localvars_pf2 localvars_dist ]]].
    set (mapOptionTree (@fst _ _) localvars) as localvars'.

  set (list2vec (leaves localvars')) as localvars''.
  cut (length (leaves localvars') = sac_numExprVars sac). intro H''.
    rewrite H'' in localvars''.
  cut (distinct (vec2list localvars'')). intro H'''.
  set (@Build_StrongCaseBranchWithVVs _ _ _ _ avars sac localvars'' H''') as scb.

  refine (bind q = (f (scbwv_ξ scb ξ lev) (vars,,(unleaves (vec2list (scbwv_exprvars scb)))) _) ; return _).
    apply FreshMon.
    simpl.
    unfold scbwv_ξ.
    rewrite vars_pf.
    rewrite <- mapOptionTree_compose.
    clear localvars_pf1.
    simpl.
```

```
    rewrite mapleaves'.

  admit.

  exists scb.
  apply ileaf in q.
  apply q.

  admit.
  admit.
  Defined.

Definition gather_branch_variables
  Γ Δ (ξ:VV -> LeveledHaskType Γ ★) tc avars tbranches lev (alts:Tree ?? {sac : StrongAltCon &
            ProofCaseBranch tc Γ Δ lev tbranches avars sac})
  :
  forall vars,
  mapOptionTreeAndFlatten (fun x => pcb_freevars(Γ:=Γ) (projT2 x)) alts = mapOptionTree ξ vars
  -> ITree Judg judg2exprType (mapOptionTree (fun x => pcb_judg (projT2 x)) alts)
  -> ITree _ (fun q => prod (judg2exprType (pcb_judg (projT2 q)))
    { vars' : _ & pcb_freevars (projT2 q) = mapOptionTree ξ vars' })
alts.
  induction alts;
  intro vars;
  intro pf;
  intro source.
  destruct a; [ idtac | apply INone ].
  simpl in *.
  apply ileaf in source.
  apply ILeaf.
  destruct s as [sac pcb].
  simpl in *.
  split.
  intros.
  eapply source.
  apply H.
  clear source.

  exists vars.
  auto.
```

```
  simpl in pf.
  destruct vars; try destruct o; simpl in pf; inversion pf.
  simpl in source.
  inversion source.
  subst.
  apply IBranch.
  apply (IHalts1 vars1 H0 X); auto.
  apply (IHalts2 vars2 H1 X0); auto.

  Defined.


Definition rule2expr : forall h j (r:Rule h j), ITree _ judg2exprType h -> ITree _ judg2exprType j.

  intros h j r.

    refine (match r as R in Rule H C return ITree _ judg2exprType H -> ITree _ judg2exprType C with
    | RArrange a b c d e  r          => let case_RURule := tt        in _
    | RNote   Γ Δ Σ τ l n            => let case_RNote := tt         in _
    | RLit    Γ Δ l     _            => let case_RLit := tt          in _
    | RVar    Γ Δ σ         p        => let case_RVar := tt          in _
    | RGlobal Γ Δ σ l wev            => let case_RGlobal := tt       in _
    | RLam    Γ Δ Σ tx te     x      => let case_RLam := tt          in _
    | RCast   Γ Δ Σ σ τ γ     x      => let case_RCast := tt         in _
    | RAbsT   Γ Δ Σ κ σ a            => let case_RAbsT := tt         in _
    | RAppT   Γ Δ Σ κ σ τ     y      => let case_RAppT := tt         in _
    | RAppCo  Γ Δ Σ κ σ₁ σ₂ γ σ l    => let case_RAppCo := tt        in _
    | RAbsCo  Γ Δ Σ κ σ  σ₁ σ₂  y    => let case_RAbsCo := tt        in _
    | RApp    Γ Δ Σ₁ Σ₂ tx te p      => let case_RApp := tt          in _
    | RLet    Γ Δ Σ₁ Σ₂ σ₁ σ₂ p      => let case_RLet := tt          in _
    | RJoin Γ p lri m x q   => let case_RJoin := tt in _
    | RVoid _ _                => let case_RVoid := tt   in _
    | RBrak   Σ a b c n m            => let case_RBrak := tt         in _
    | REsc    Σ a b c n m            => let case_REsc := tt          in _
    | RCase   Γ Δ lev tc Σ avars tbranches alts => let case_RCase := tt         in _
    | RLetRec Γ Δ lri x y t          => let case_RLetRec := tt       in _
    end); intro X_; try apply ileaf in X_; simpl in X_.

  destruct case_RURule.
    apply ILeaf. simpl. intros.
    set (@urule2expr a b _ _ e r0 ξ) as q.
```

```
    set (fun z => q z) as q'.
    simpl in q'.
    apply q' with (vars:=vars).
    clear q' q.
    unfold ujudg2exprType.
    intros.
    apply X_ with (vars:=vars0).
    auto.
    auto.

destruct case_RBrak.
  apply ILeaf; simpl; intros; refine (X_ ξ vars H >>>= fun X => return ILeaf _ _). apply FreshMon.
  apply EBrak.
  apply (ileaf X).

destruct case_REsc.
  apply ILeaf; simpl; intros; refine (X_ ξ vars H >>>= fun X => return ILeaf _ _). apply FreshMon.
  apply EEsc.
  apply (ileaf X).

destruct case_RNote.
  apply ILeaf; simpl; intros; refine (X_ ξ vars H >>>= fun X => return ILeaf _ _). apply FreshMon.
  apply ENote; auto.
  apply (ileaf X).

destruct case_RLit.
  apply ILeaf; simpl; intros; refine (return ILeaf _ _).
  apply ELit.

destruct case_RVar.
  apply ILeaf; simpl; intros; refine (return ILeaf _ _).
  destruct vars; simpl in H; inversion H; destruct o. inversion H1. rewrite H2.
  apply EVar.
  inversion H.

destruct case_RGlobal.
  apply ILeaf; simpl; intros; refine (return ILeaf _ _).
  apply EGlobal.
  apply wev.

destruct case_RLam.
```

```
  apply ILeaf.
  simpl in *; intros.
  refine (fresh_lemma _ ξ vars _ tx x H >>>= (fun pf => _)).
  apply FreshMon.
  destruct pf as [ vnew [ pf1 pf2 ]].
  set (update_ξ ξ x (((vnew, tx  )) :: nil)) as ξ' in *.
  refine (X_ ξ' (vars,,[vnew]) _ >>>= _).
  apply FreshMon.
  simpl.
  rewrite pf1.
  rewrite <- pf2.
  simpl.
  reflexivity.
  intro hyp.
  refine (return _).
  apply ILeaf.
  apply ELam with (ev:=vnew).
  apply ileaf in hyp.
  simpl in hyp.
  unfold ξ' in hyp.
  apply hyp.

destruct case_RCast.
  apply ILeaf; simpl; intros; refine (X_ ξ vars H >>>= fun X => return ILeaf _ _). apply FreshMon.
  eapply ECast.
  apply x.
  apply ileaf in X. simpl in X.
  apply X.

destruct case_RJoin.
  apply ILeaf; simpl; intros.
  inversion X_.
  apply ileaf in X.
  apply ileaf in X0.
  simpl in *.
  destruct vars; inversion H.
  destruct o; inversion H3.
  refine (X ξ vars1 H3 >>>= fun X' => X0 ξ vars2 H4 >>>= fun X0' => return _).
  apply FreshMon.
  apply FreshMon.
  apply IBranch; auto.
```

```
destruct case_RApp.
  apply ILeaf.
  inversion X_.
  inversion X.
  inversion X0.
  simpl in *.
  intros.
  destruct vars. try destruct o; inversion H.
  simpl in H.
  inversion H.
  set (X1 ξ vars1 H5) as q1.
  set (X2 ξ vars2 H6) as q2.
  refine (q1 >>>= fun q1' => q2 >>>= fun q2' => return _).
  apply FreshMon.
  apply FreshMon.
  apply ILeaf.
  apply ileaf in q1'.
  apply ileaf in q2'.
  simpl in *.
  apply (EApp _ _ _ _ _ _ q1' q2').

destruct case_RLet.
  apply ILeaf.
  simpl in *; intros.
  destruct vars; try destruct o; inversion H.
  refine (fresh_lemma _ ξ vars1 _ σ2 p H1 >>>= (fun pf => _)).
  apply FreshMon.
  destruct pf as [ vnew [ pf1 pf2 ]].
  set (update_ξ ξ p (((vnew, σ2 )) :: nil)) as ξ' in *.
  inversion X_.
  apply ileaf in X.
  apply ileaf in X0.
  simpl in *.
  refine (X ξ  vars2 _ >>>= fun X0' => _).
  apply FreshMon.
  auto.

  refine (X0 ξ' (vars1,,[vnew]) _ >>>= fun X1' => _).
  apply FreshMon.
  rewrite H1.
```

```
  simpl.
  rewrite pf2.
  rewrite pf1.
  rewrite H1.
  reflexivity.

  refine (return _).
  apply ILeaf.
  apply ileaf in X0'.
  apply ileaf in X1'.
  simpl in *.
  apply ELet with (ev:=vnew)(tv:=σ2).
  apply X0'.
  apply X1'.

destruct case_RVoid.
  apply ILeaf; simpl; intros.
  refine (return _).
  apply INone.

destruct case_RAppT.
  apply ILeaf; simpl; intros; refine (X_ ξ vars H >>>= fun X => return ILeaf _ _). apply FreshMon.
  apply ETyApp.
  apply (ileaf X).

destruct case_RAbsT.
  apply ILeaf; simpl; intros; refine (X_ (weakLT ○ ξ) vars _ >>>= fun X => return ILeaf _ _). apply FreshMon.
  rewrite mapOptionTree_compose.
  rewrite <- H.
  reflexivity.
  apply ileaf in X. simpl in *.
  apply ETyLam.
  apply X.

destruct case_RAppCo.
  apply ILeaf; simpl; intros; refine (X_ ξ vars _ >>>= fun X => return ILeaf _ _). apply FreshMon.
  auto.
  eapply ECoApp.
  apply γ.
  apply (ileaf X).
```

```
destruct case_RAbsCo.
  apply ILeaf; simpl; intros; refine (X_ ξ vars _ >>>= fun X => return ILeaf _ _). apply FreshMon.
  auto.
  eapply ECoLam.
  apply (ileaf X).

destruct case_RLetRec.
  apply ILeaf; simpl; intros.
  refine (bind ξvars = fresh_lemma' _ y _ _ _ t H; _). apply FreshMon.
  destruct ξvars as [ varstypes [ pf1[ pf2 pfdist]]].
  refine (X_ ((update_ξ ξ t (leaves varstypes)))
    (vars,,(mapOptionTree (@fst _ _) varstypes)) _ >>>= fun X => return _); clear X_.  apply FreshMon.
  simpl.
  rewrite pf2.
  rewrite pf1.
  auto.
  apply ILeaf.
  inversion X; subst; clear X.

  apply (@ELetRec _ _ _ _ _ _ _ varstypes).
  apply (@letrec_helper Γ Δ t varstypes).
  rewrite <- pf2 in X1.
  rewrite mapOptionTree_compose.
  apply X1.
  apply ileaf in X0.
  apply X0.

destruct case_RCase.
  apply ILeaf; simpl; intros.
  inversion X_.
  clear X_.
  subst.
  apply ileaf in X0.
  simpl in X0.

  (* body_freevars and alts_freevars are the types of variables in the body and alternatives (respectively) which are free
   * from the viewpoint just outside the case block -- i.e. not bound by any of the branches *)
  rename Σ into body_freevars_types.
  rename vars into all_freevars.
  rename X0 into body_expr.
  rename X  into alts_exprs.
```

```
  destruct all_freevars; try destruct o; inversion H.
  rename all_freevars2 into body_freevars.
  rename all_freevars1 into alts_freevars.

  set (gather_branch_variables _ _ _ _ _ _ _ _ _ _ H1 alts_exprs) as q.
  set (itmap (fun pcb alt_expr => case_helper tc Γ Δ lev tbranches avars ξ pcb alt_expr) q) as alts_exprs'.
  apply fix_indexing in alts_exprs'.
  simpl in alts_exprs'.
  apply unindex_tree in alts_exprs'.
  simpl in alts_exprs'.
  apply fix2 in alts_exprs'.
  apply treeM in alts_exprs'.

  refine ( alts_exprs' >>>= fun Y =>
    body_expr ξ _ _
    >>>= fun X => return ILeaf _ (@ECase _ _ _ _ _ _ _ _ _ _ (ileaf X) Y)); auto.
    apply FreshMon.
    apply FreshMon.
    apply H2.
  Defined.

Definition closed2expr : forall c (pn:@ClosedSIND _ Rule c), ITree _ judg2exprType c.
  refine ((
    fix closed2expr' j (pn:@ClosedSIND _ Rule j) {struct pn} : ITree _ judg2exprType j :=
    match pn in @ClosedSIND _ _ J return ITree _ judg2exprType J with
    | cnd_weak             => let case_nil    := tt in INone _ _
    | cnd_rule h c cnd' r  => let case_rule   := tt in rule2expr _ _ r (closed2expr' _ cnd')
    | cnd_branch _ _ c1 c2 => let case_branch := tt in IBranch _ _ (closed2expr' _ c1) (closed2expr' _ c2)
    end)); clear closed2expr'; intros; subst.
      Defined.

Lemma manyFresh : forall Γ Σ (ξ0:VV -> LeveledHaskType Γ ★),
  FreshM { vars : _ & { ξ : VV -> LeveledHaskType Γ ★ & Σ = mapOptionTree ξ vars } }.
  intros Γ Σ.
  induction Σ; intro ξ.
  destruct a.
  destruct l as [τ l].
  set (fresh_lemma' Γ [τ] [] [] ξ l (refl_equal _)) as q.
  refine (q >>>= fun q' => return _).
  apply FreshMon.
```

```
  clear q.
  destruct q' as [varstypes [pf1 [pf2 distpf]]].
  exists (mapOptionTree (@fst _ _) varstypes).
  exists (update_ξ ξ l (leaves varstypes)).
  symmetry; auto.
  refine (return _).
  exists [].
  exists ξ; auto.
  refine (bind f1 = IHΣ1 ξ ; _).
  apply FreshMon.
  destruct f1 as [vars1 [ξ1 pf1]].
  refine (bind f2 = IHΣ2 ξ1 ; _).
  apply FreshMon.
  destruct f2 as [vars2 [ξ2 pf22]].
  refine (return _).
  exists (vars1,,vars2).
  exists ξ2.
  simpl.
  rewrite pf22.
  rewrite pf1.
  admit.
  Defined.

Definition proof2expr Γ Δ τ Σ (ξ0: VV -> LeveledHaskType Γ ★)
  {zz:ToString VV} : ND Rule [] [Γ > Δ > Σ |- [τ]] ->
  FreshM (???{ ξ : _ & Expr Γ Δ ξ τ}).
  intro pf.
  set (closedFromSIND _ _ (mkSIND systemfc_all_rules_one_conclusion _ _ _ pf (scnd_weak [])) cnd_weak) as cnd.
  apply closed2expr in cnd.
  apply ileaf in cnd.
  simpl in *.
  clear pf.
  refine (bind ξvars = manyFresh _ Σ ξ0; _).
  apply FreshMon.
  destruct ξvars as [vars ξpf].
  destruct ξpf as [ξ pf].
  refine (cnd ξ vars _ >>>= fun it => _).
  apply FreshMon.
  auto.
  refine (return OK _).
  exists ξ.
```

```
    apply (ileaf it).
    Defined.

End HaskProofToStrong.
```