```
(*****************************************************************************************************************************)
(* HaskStrongTypes: representation of types and coercions for HaskStrong                                                     *)
(*****************************************************************************************************************************)

Generalizable All Variables.
Require Import Preamble.
Require Import Coq.Strings.String.
Require Import Coq.Lists.List.
Require Import General.
Require Import HaskKinds.
Require Import HaskLiteralsAndTyCons.
Require Import HaskCoreTypes.
Require Import HaskCoreVars.
Require Import HaskWeakTypes.
Require Import HaskWeakVars.
Require Import HaskWeak.
Require Import HaskCoreToWeak.

Variable dataConTyCon      : CoreDataCon -> TyCon.         Extract Inlined Constant dataConTyCon      => "DataCon.dataConTyCon".
Variable dataConExVars_    : CoreDataCon -> list CoreVar.  Extract Inlined Constant dataConExVars_    => "DataCon.dataConExTyVars".
Variable dataConEqTheta_   : CoreDataCon -> list PredType. Extract Inlined Constant dataConEqTheta_   => "DataCon.dataConEqTheta".
Variable dataConOrigArgTys_: CoreDataCon -> list CoreType. Extract Inlined Constant dataConOrigArgTys_=>"DataCon.dataConOrigArgTys".

Definition dataConExTyVars cdc :=
  filter (map (fun x => match coreVarToWeakVar x with WTypeVar v => Some v | _ => None end) (dataConExVars_ cdc)).
  Opaque dataConExTyVars.
Definition dataConCoerKinds cdc :=
  filter (map (fun x => match x with EqPred t1 t2 =>
                          match (
                            coreTypeToWeakType t1 >>= fun t1' =>
                              coreTypeToWeakType t2 >>= fun t2' =>
                                OK (t1',t2'))
                          with OK z => Some z
                            | _ => None
                          end
                          | _ => None
                        end) (dataConEqTheta_ cdc)).
  Opaque dataConCoerKinds.
Definition dataConFieldTypes cdc :=
  filter (map (fun x => match coreTypeToWeakType x with
                          | OK z => Some z
```

```
                        | _ => None
                      end) (dataConOrigArgTys_ cdc)).

Definition tyConNumKinds (tc:TyCon) := length (tyConTyVars tc).
  Coercion tyConNumKinds : TyCon >-> nat.

Inductive DataCon : TyCon -> Type :=
  mkDataCon : forall cdc:CoreDataCon, DataCon (dataConTyCon cdc).
  Definition dataConToCoreDataCon '(dc:DataCon tc) : CoreDataCon := match dc with mkDataCon cdc => cdc end.
  Coercion mkDataCon : CoreDataCon >-> DataCon.
  Coercion dataConToCoreDataCon : DataCon >-> CoreDataCon.
  (*Opaque DataCon.*)

Definition tyConKind' tc := fold_right KindArrow ★(tyConKind tc).

(* types prefixed with "Raw" are NOT binder-polymorphic; they have had their PHOAS parameter instantiated already *)
Section Raw.

  (* TV is the PHOAS type which stands for type variables of System FC *)
  Context {TV:Kind -> Type}.

  (* Figure 7: ρ, σ, τ, ν *)
  Inductive RawHaskType : Kind -> Type :=
  | TVar           : ∀ κ, TV κ                                                    -> RawHaskType κ                      (* a        *)
  | TCon           : ∀ tc,                                                          RawHaskType (tyConKind' tc)       (* T        *)
  | TArrow         :                                                                RawHaskType (★⇉★⇉★)               (* (->)     *)
  | TCoerc         : ∀ κ, RawHaskType κ -> RawHaskType κ -> RawHaskType ★          -> RawHaskType ★                    (* (+>)     *)
  | TApp           : ∀ κ1 κ2, RawHaskType (κ2⇉κ1)        -> RawHaskType κ2         -> RawHaskType κ1                   (* φ φ      *)
  | TAll           : ∀ κ,                          (TV κ -> RawHaskType ★)        -> RawHaskType ★                    (* ∀a:κ.φ   *)
  | TCode          : RawHaskType ★                 -> RawHaskType ★               -> RawHaskType ★                    (* from λ^α *)
  | TyFunApp       : ∀ tf, RawHaskTypeList (fst (tyFunKind tf))                   -> RawHaskType (snd (tyFunKind tf)) (* S_n      *)
  with RawHaskTypeList : list Kind -> Type :=
  | TyFunApp_nil   : RawHaskTypeList nil
  | TyFunApp_cons  : ∀ κ kl, RawHaskType κ -> RawHaskTypeList kl -> RawHaskTypeList (κ::kl).

  (* the "kind" of a coercion is a pair of types *)
  Inductive RawCoercionKind : Type :=
    mkRawCoercionKind : ∀ κ, RawHaskType κ -> RawHaskType κ -> RawCoercionKind.

  (* Figure 7: γ, δ; CV is the PHOAS type which stands for coercion variables of System FC *)
  Inductive RawHaskCoer {CV:Type} : RawCoercionKind -> Prop := .
```

```
  (*
   *  This has been disabled until we manage to reconcile SystemFC's
   *  coercions with what GHC actually implements (they are not the
   *  same!)
   *
  | CoVar          : CV                                        -> RawHaskCoer (* g      *)
  | CoType         : RawHaskType                               -> RawHaskCoer (* τ      *)
  | CoApp          : RawHaskCoer -> RawHaskCoer                -> RawHaskCoer (* γ γ    *)
  | CoAppT         : RawHaskCoer -> RawHaskType                -> RawHaskCoer (* γ@v    *)
  | CoCFApp        : ∀ n, CoFunConst n -> vec RawHaskCoer n    -> RawHaskCoer (* C   γ8319 *)
  | CoTFApp        : ∀ n, TyFunConst n -> vec RawHaskCoer n    -> RawHaskCoer (* S_n γ8319 *)
  | CoAll          : Kind  -> (TV -> RawHaskCoer)              -> RawHaskCoer (* ∀a:κ.γ *)
  | CoSym          : RawHaskCoer                               -> RawHaskCoer (* sym    *)
  | CoComp         : RawHaskCoer -> RawHaskCoer                -> RawHaskCoer (* ◯      *)
  | CoLeft         : RawHaskCoer                               -> RawHaskCoer (* left   *)
  | CoRight        : RawHaskCoer                               -> RawHaskCoer (* right  *).
  *)
End Raw.

Implicit Arguments TCon   [ [TV] ].
Implicit Arguments TyFunApp [ [TV] ].
Implicit Arguments RawHaskType  [ ].
Implicit Arguments RawHaskCoer  [ ].
Implicit Arguments RawCoercionKind [ ].
Implicit Arguments TVar [ [TV] [κ] ].
Implicit Arguments TCoerc [ [TV] [κ] ].
Implicit Arguments TApp   [ [TV] [κ1] [κ2] ].
Implicit Arguments TAll   [ [TV] ].

Notation "t1 ---> t2"        := (fun TV env => (TApp (TApp TArrow (t1 TV env)) (t2 TV env))).
Notation "φ1 ∼∼ φ2 ⇒φ3"      := (fun TV env => TCoerc (φ1 TV env) (φ2 TV env) (φ3 TV env)).

(* Kind and Coercion Environments *)
(*
 *  In System FC, the environment consists of three components, each of
 *  whose well-formedness depends on all of those prior to it:
 *
 *    1. (TypeEnv)        The list of free type     variables and their kinds
 *    2. (CoercionEnv)    The list of free coercion variables and the pair of types between which it witnesses coercibility
 *    3. (Tree ??CoreVar) The list of free value    variables and the type of each one
 *)
```

```
Definition TypeEnv                                                    := list Kind.
Definition InstantiatedTypeEnv     (TV:Kind->Type)         (Γ:TypeEnv)     := IList _ TV Γ.
Definition HaskCoercionKind                    (Γ:TypeEnv)                := ∀ TV, InstantiatedTypeEnv TV Γ -> @RawCoercionKind TV.
Definition CoercionEnv             (Γ:TypeEnv)                         := list (HaskCoercionKind Γ).
Definition InstantiatedCoercionEnv (TV:Kind->Type) CV       (Γ:TypeEnv)(Δ:CoercionEnv Γ):= vec CV (length Δ).

(* A (HaskXX Γ) is an XX which is valid in environments of shape Γ; they are always PHOAS-uninstantiated *)
Definition HaskTyVar (Γ:TypeEnv) κ :=  forall TV    (env:@InstantiatedTypeEnv TV Γ), TV κ.
Definition HaskCoVar Γ Δ           :=  forall TV CV (env:@InstantiatedTypeEnv TV Γ)(cenv:@InstantiatedCoercionEnv TV CV Γ Δ), CV.
Definition HaskLevel (Γ:TypeEnv)   :=  list (HaskTyVar Γ ★).
Definition HaskType  (Γ:TypeEnv) κ := ∀ TV, @InstantiatedTypeEnv TV Γ -> RawHaskType TV κ.
Definition haskTyVarToType {Γ}{κ}(htv:HaskTyVar Γ κ) : HaskType Γ κ := fun TV ite => TVar (htv TV ite).

Inductive HaskTypeOfSomeKind (Γ:TypeEnv) :=
  haskTypeOfSomeKind : ∀ κ, HaskType Γ κ -> HaskTypeOfSomeKind Γ.
  Implicit Arguments haskTypeOfSomeKind [ [Γ] [κ] ].
  Definition kindOfHaskTypeOfSomeKind {Γ}(htosk:HaskTypeOfSomeKind Γ) :=
    match htosk with
      haskTypeOfSomeKind κ _ => κ
    end.
  Coercion kindOfHaskTypeOfSomeKind : HaskTypeOfSomeKind >-> Kind.
  Definition haskTypeOfSomeKindToHaskType {Γ}(htosk:HaskTypeOfSomeKind Γ) : HaskType Γ htosk :=
    match htosk as H return HaskType Γ H with
      haskTypeOfSomeKind _ ht => ht
      end.
  Coercion haskTypeOfSomeKindToHaskType : HaskTypeOfSomeKind >-> HaskType.

Definition HaskCoercion Γ Δ (hk:HaskCoercionKind Γ) := forall TV CV (ite:@InstantiatedTypeEnv TV Γ),
    @InstantiatedCoercionEnv TV CV Γ Δ -> @RawHaskCoer TV CV (hk TV ite).
Inductive  LeveledHaskType (Γ:TypeEnv) κ := mkLeveledHaskType : HaskType Γ κ -> HaskLevel Γ -> LeveledHaskType Γ κ.

Definition FreshHaskTyVar {Γ}(κ:Kind) : HaskTyVar (κ::Γ) κ := fun TV env => ilist_head env.
Definition HaskTAll {Γ}(κ:Kind)(σ:forall TV (env:@InstantiatedTypeEnv TV Γ), TV κ -> RawHaskType TV ★) : HaskType Γ ★
  := fun TV env => TAll κ (σ TV env).
Definition HaskTApp {Γ}{κ}(σ:forall TV (env:@InstantiatedTypeEnv TV Γ), TV κ -> RawHaskType TV ★)
  (cv:HaskTyVar Γ κ) : HaskType Γ ★
  := fun TV env => σ TV env (cv TV env).
Definition HaskBrak {Γ}(v:HaskTyVar Γ ★)(t:HaskType Γ ★) : HaskType Γ ★:=
  fun TV env => @TCode TV (TVar (v TV env)) (t TV env).
Definition HaskTCon {Γ}(tc:TyCon) : HaskType Γ (fold_right KindArrow ★(tyConKind tc))
```

```
  := fun TV ite => TCon tc.
Definition HaskAppT {Γ}{κ1}{κ2}(t1:HaskType Γ (κ2 ⇉ κ1))(t2:HaskType Γ κ2) : HaskType Γ κ1 :=
  fun TV ite => TApp (t1 TV ite) (t2 TV ite).
Definition mkHaskCoercionKind {Γ}{κ}(t1:HaskType Γ κ)(t2:HaskType Γ κ) : HaskCoercionKind Γ :=
 fun TV ite => mkRawCoercionKind _ (t1 TV ite) (t2 TV ite).

(* PHOAS substitution on types *)
Definition substT {Γ}{κ1}{κ2}(exp:forall TV (env:@InstantiatedTypeEnv TV Γ), TV κ1 -> RawHaskType TV κ2)(v:@HaskType Γ κ1)
  : @HaskType Γ κ2 :=
fun TV env =>
    (fix flattenT {κ} (exp: RawHaskType (fun k => RawHaskType TV k) κ) : RawHaskType TV κ :=
     match exp with
    | TVar    _  x        => x
    | TAll     _ y        => TAll   _  (fun v => flattenT _ (y (TVar v)))
    | TApp   _ _ x y      => TApp      (flattenT _ x) (flattenT _ y)
    | TCon       tc       => TCon      tc
    | TCoerc _ t1 t2 t    => TCoerc    (flattenT _ t1) (flattenT _ t2)   (flattenT _ t)
    | TArrow              => TArrow
    | TCode      v e      => TCode     (flattenT _ v) (flattenT _ e)
    | TyFunApp       tfc lt => TyFunApp tfc (flattenTyFunApp _ lt)
    end
    with flattenTyFunApp (lk:list Kind)(exp:@RawHaskTypeList (fun k => RawHaskType TV k) lk) : @RawHaskTypeList TV lk :=
    match exp in @RawHaskTypeList _ LK return @RawHaskTypeList TV LK with
    | TyFunApp_nil               => TyFunApp_nil
    | TyFunApp_cons  κ kl t rest => TyFunApp_cons _ _ (flattenT _ t) (flattenTyFunApp _ rest)
    end
    for flattenT) _ (exp (fun k => RawHaskType TV k) (ilmap (fun κ tv => TVar tv) env) (v TV env)).

Notation "t @@  l" := (@mkLeveledHaskType _ _ t l) (at level 20).
Notation "t @@@ l" := (mapOptionTree (fun t' => t' @@ l) t) (at level 20).
Notation "'<[' a '|-' t ']>'" := (@HaskBrak _ a t).

Definition unlev {Γ}{κ}(lht:LeveledHaskType Γ κ) :=
  match lht with t@@l => t end.




(* yeah, things are kind of messy below this point *)
```

```
Definition unAddKindFromInstantiatedTypeEnv {Γ:TypeEnv}{κ:Kind}{TV:Kind->Type}(ite:InstantiatedTypeEnv TV (κ::Γ))
  := ilist_tail ite.
Definition addKindToCoercionEnv (Γ:TypeEnv)(Δ:CoercionEnv Γ)(κ:Kind) : CoercionEnv (κ::Γ) :=
  map (fun f => (fun TV ite => f TV (unAddKindFromInstantiatedTypeEnv ite))) Δ.
Definition addKindToInstantiatedTypeEnv {Γ:TypeEnv}{TV:Kind->Type}(env:InstantiatedTypeEnv TV Γ)(κ:Kind)(tv:TV κ)
  : InstantiatedTypeEnv TV (κ::Γ) := tv::::env.
Definition addKindToInstantiatedCoercionEnv {Γ:TypeEnv}{Δ}{TV:Kind->Type}{CV:Type}
  (env:InstantiatedCoercionEnv TV CV Γ Δ)(κ:Kind)(tv:TV κ)
  : InstantiatedCoercionEnv TV CV (κ::Γ) (addKindToCoercionEnv Γ Δ κ).
    simpl.
    unfold InstantiatedCoercionEnv.
    unfold addKindToCoercionEnv.
    simpl.
    rewrite <- map_preserves_length.
    apply env.
    Defined.
Definition coercionEnvContainsCoercion {Γ}{Δ}{TV:Kind->Type}{CV:Type}(ite:InstantiatedTypeEnv TV Γ)
  (ice:InstantiatedCoercionEnv TV CV Γ Δ)(cv:CV)(ck:RawCoercionKind TV)
  := @vec_In _ _ (cv,ck) (vec_zip ice (vec_map (fun f => f TV ite) (list2vec Δ))).
Definition addCoercionToCoercionEnv {Γ}(Δ:CoercionEnv Γ)(κ:HaskCoercionKind Γ) : CoercionEnv Γ :=
  κ::Δ.
Definition addCoercionToInstantiatedCoercionEnv {Γ}{Δ}{κ}{TV CV}(ice:InstantiatedCoercionEnv TV CV Γ Δ)(cv:CV)
  : InstantiatedCoercionEnv TV CV Γ (addCoercionToCoercionEnv Δ κ).
  simpl.
  unfold addCoercionToCoercionEnv; simpl.
  unfold InstantiatedCoercionEnv; simpl.
  apply vec_cons; auto.
  Defined.
(* the various "weak" functions turn a HaskXX-in-Γ into a HaskXX-in-(κ::Γ) *)
Definition weakITE  {Γ:TypeEnv}{κ}{TV}(ite:InstantiatedTypeEnv TV (κ::Γ)) : InstantiatedTypeEnv TV Γ
  := ilist_tail ite.
Definition weakITE' {Γ:TypeEnv}{κ}{TV}(ite:InstantiatedTypeEnv TV (app κ Γ)) : InstantiatedTypeEnv TV Γ.
  induction κ; auto. apply IHκ. inversion ite; subst. apply X0. Defined.
Definition weakCE   {Γ:TypeEnv}{κ}(Δ:CoercionEnv Γ) : CoercionEnv (κ::Γ)
  := map (fun x => (fun tv ite => x tv (weakITE ite))) Δ.
Definition weakV  {Γ:TypeEnv}{κ}{κv}(cv':HaskTyVar Γ κv) : HaskTyVar (κ::Γ) κv
  := fun TV ite => (cv' TV (weakITE ite)).
Definition weakV' {Γ:TypeEnv}{κ}{κv}(cv':HaskTyVar Γ κv) : HaskTyVar (app κ Γ) κv.
  induction κ; auto. apply weakV; auto. Defined.
```

```
Definition weakT {Γ:TypeEnv}{κ}{κ₂}(lt:HaskType Γ κ₂) : HaskType (κ::Γ) κ₂
  := fun TV ite => lt TV (weakITE ite).
Definition weakL  {Γ}{κ}(lt:HaskLevel Γ) : HaskLevel (κ::Γ)
  := map weakV lt.
Definition weakT' {Γ}{κ}{κ₂}(lt:HaskType Γ κ₂) : HaskType (app κ Γ) κ₂.
  induction κ; auto. apply weakT; auto. Defined.
Definition weakT'' {Γ}{κ}{κ₂}(lt:HaskType Γ κ₂) : HaskType (app Γ κ) κ₂.
  unfold HaskType in *.
  unfold InstantiatedTypeEnv in *.
  intros.
  apply ilist_chop in X.
  apply lt.
  apply X.
  Defined.
Definition lamer {a}{b}{c}{κ}(lt:HaskType (app (app a  b) c) κ) : HaskType (app a (app b c)) κ.
  rewrite <- ass_app in lt.
  exact lt.
  Defined.
Definition weakL' {Γ}{κ}(lev:HaskLevel Γ) : HaskLevel (app κ Γ).
  induction κ; auto. apply weakL; auto. Defined.
Definition weakLT {Γ}{κ}{κ₂}(lt:LeveledHaskType Γ κ₂) : LeveledHaskType (κ::Γ) κ₂
  := match lt with t @@ l => weakT t @@ weakL l end.
Definition weakLT' {Γ}{κ}{κ₂}(lt:LeveledHaskType Γ κ₂) : LeveledHaskType (app κ Γ) κ₂
  := match lt with t @@ l => weakT' t @@ weakL' l end.
Definition weakCE' {Γ:TypeEnv}{κ}(Δ:CoercionEnv Γ) : CoercionEnv (app κ Γ).
  induction κ; auto. apply weakCE; auto. Defined.
Definition weakICE  {Γ:TypeEnv}{κ}{Δ:CoercionEnv Γ}{TV}{CV}(ice:InstantiatedCoercionEnv TV CV (κ::Γ) (weakCE Δ))
  : InstantiatedCoercionEnv TV CV Γ Δ.
  intros.
  unfold InstantiatedCoercionEnv; intros.
  unfold InstantiatedCoercionEnv in ice.
  unfold weakCE in ice.
  simpl in ice.
  rewrite <- map_preserves_length in ice.
  apply ice.
  Defined.
Definition weakCK {Γ}{κ}(hck:HaskCoercionKind Γ) : HaskCoercionKind (κ::Γ).
  unfold HaskCoercionKind in *.
  intros.
  apply hck; clear hck.
  inversion X; subst; auto.
```

```
  Defined.
Definition weakCK' {Γ}{κ}(hck:HaskCoercionKind Γ) : HaskCoercionKind (app κ Γ).
  induction κ; auto.
  apply weakCK.
  apply IHκ.
  Defined.
Definition weakCK'' {Γ}{κ}(hck:list (HaskCoercionKind Γ)) : list (HaskCoercionKind (app κ Γ)) :=
  map weakCK' hck.
Definition weakCV {Γ}{Δ}{κ}(cv':HaskCoVar Γ Δ) : HaskCoVar (κ::Γ) (weakCE Δ) :=
  fun TV CV ite ice => (cv' TV CV (weakITE ite) (weakICE ice)).
Definition weakF {Γ:TypeEnv}{κ}{κ2}(f:forall TV (env:@InstantiatedTypeEnv TV Γ), TV κ -> RawHaskType TV κ2) :
  forall TV (env:@InstantiatedTypeEnv TV (κ::Γ)), TV κ -> RawHaskType TV κ2
  := fun TV ite tv => (f TV (weakITE ite) tv).

Fixpoint caseType0 {Γ}(lk:list Kind) :
  IList _ (HaskType Γ) lk ->
  HaskType Γ (fold_right KindArrow ★lk) ->
  HaskType Γ ★:=
  match lk as LK return
    IList _ (HaskType Γ) LK ->
    HaskType Γ (fold_right KindArrow ★LK) ->
    HaskType Γ ★
  with
  | nil    => fun _     ht => ht
  | k::lk' => fun tlist ht => caseType0 lk' (ilist_tail tlist) (fun TV env => TApp (ht TV env) (ilist_head tlist TV env))
  end.

Definition caseType {Γ}(tc:TyCon)(atypes:IList _ (HaskType Γ) (tyConKind tc)) : HaskType Γ ★:=
  caseType0 (tyConKind tc) atypes (fun TV env => TCon tc).

(* like a GHC DataCon, but using PHOAS representation for types and coercions *)
Record StrongAltCon {tc:TyCon} :=
{ sac_tc          := tc
; sac_altcon      :  WeakAltCon
; sac_numExTyVars :  nat
; sac_numCoerVars :  nat
; sac_numExprVars :  nat
; sac_ekinds      :  vec Kind sac_numExTyVars
; sac_kinds       := app (tyConKind tc) (vec2list sac_ekinds)
; sac_Γ           := fun Γ => app (vec2list sac_ekinds) Γ
; sac_coercions   :  forall Γ (atypes:IList _ (HaskType Γ) (tyConKind tc)), vec (HaskCoercionKind (sac_Γ Γ)) sac_numCoerVars
```

```
; sac_types       :  forall Γ (atypes:IList _ (HaskType Γ) (tyConKind tc)), vec (HaskType (sac_Γ Γ) ★) sac_numExprVars
; sac_Δ           := fun    Γ (atypes:IList _ (HaskType Γ) (tyConKind tc)) Δ => app (vec2list (sac_coercions Γ atypes)) Δ
}.
Coercion sac_tc     : StrongAltCon >-> TyCon.
Coercion sac_altcon : StrongAltCon >-> WeakAltCon.


Definition kindOfType {Γ}{κ}(ht:@HaskType Γ κ) : ???Kind := OK κ.

Axiom literal_tycons_are_of_ordinary_kind : forall lit, tyConKind (haskLiteralToTyCon lit) = nil.

Definition literalType (lit:HaskLiteral){Γ} : HaskType Γ ★.
  set (fun TV (ite:InstantiatedTypeEnv TV Γ) => @TCon TV (haskLiteralToTyCon lit)) as z.
  unfold tyConKind' in z.
  rewrite literal_tycons_are_of_ordinary_kind in z.
  unfold HaskType.
  apply z.
  Defined.

Notation "a ~~~ b" := (@mkHaskCoercionKind _ _ a b) (at level 18).

Fixpoint update_ξ
  `{EQD_VV:EqDecidable VV}{Γ}
   (ξ:VV -> LeveledHaskType Γ ★)
   (lev:HaskLevel Γ)
   (vt:list (VV * HaskType Γ ★))
   : VV -> LeveledHaskType Γ ★ :=
  match vt with
    | nil => ξ
    | (v,τ)::tl => fun v' => if eqd_dec v v' then τ @@ lev else (update_ξ ξ lev tl) v'
  end.

Lemma update_ξ_lemma0 `{EQD_VV:EqDecidable VV} : forall Γ ξ (lev:HaskLevel Γ)(varstypes:list (VV*_)) v,
  not (In v (map (@fst _ _) varstypes)) ->
  (update_ξ ξ lev varstypes) v = ξ v.
  intros.
  induction varstypes.
  reflexivity.
  simpl.
  destruct a.
  destruct (eqd_dec v0 v).
```

```
  subst.
  simpl in  H.
  assert False.
  apply H.
  auto.
  inversion H0.
  apply IHvarstypes.
  unfold not; intros.
  apply H.
  simpl.
  auto.
  Defined.


(*************************************************************************************************)
(* Well-Formedness of Types and Coercions                                                        *)
(* also represents production "S_n:κ" of Γ because these can only wind up in Γ via rule (Type) *)
Inductive TypeFunctionDecl (tfc:TyCon)(vk:vec Kind tfc) : Type :=
  mkTFD : Kind -> TypeFunctionDecl tfc vk.

(*
Section WFCo.
  Context {TV:Kind->Type}.
  Context {CV:Type}.

  (* local notations *)
  Notation "ienv '⊢ty' σ : κ"              := (@WellKinded_RawHaskType TV _ ienv σ κ).
  Notation "env  ∋  cv : t1 ∼ t2 : Γ : t"  := (@coercionEnvContainsCoercion Γ _ TV CV t env cv (@mkRawCoercionKind _ t1 t2))
                (at level 20, t1 at level 99, t2 at level 99, t at level 99).
  Reserved Notation "ice '⊢co' γ : a '∼' b : Δ : Γ : ite"
        (at level 20, γ at level 99, b at level 99, Δ at level 99, ite at level 99, Γ at level 99).

  (* Figure 8, lower half *)
  Inductive WFCoercion:forall Γ (Δ:CoercionEnv Γ),
    @InstantiatedTypeEnv TV Γ ->
    @InstantiatedCoercionEnv TV CV Γ Δ ->
    @RawHaskCoer TV CV -> @RawCoercionKind TV -> Prop :=
  | CoTVar':∀ Γ Δ t e c σ τ,
    (@coercionEnvContainsCoercion Γ _ TV CV t e c (@mkRawCoercionKind _ σ τ)) -> e⊢co CoVar c : σ ∼ τ  : Δ : Γ : t
  | CoRefl :∀ Γ Δ t e   τ κ,                                    t ⊢ty τ :κ    -> e⊢co CoType τ    :        τ ∼ τ  : Δ :Γ: t
  | Sym    :∀ Γ Δ t e γ σ τ,                          (e⊢co γ : σ ∼ τ : Δ : Γ:t)  -> e⊢co CoSym  γ    :        τ ∼ σ  : Δ :Γ: t
```

```
  | Trans  :∀ Γ Δ t e γ₁ γ₂ σ₁ σ₂ σ₃,(e⊢co γ₁:σ₁∼σ₂:Δ:Γ:t) -> (e⊢co γ₂:σ₂∼σ₃:Δ:Γ:t) -> e⊢co CoComp γ₁ γ₂:        σ₁ ∼ σ₃ : Δ :Γ: t
  | Left   :∀ Γ Δ t e γ σ₁ σ₂ τ₁ τ₂,(e⊢co γ : TApp σ₁ σ₂ ∼ TApp τ₁ τ₂ :Δ:Γ:t    ) -> e⊢co CoLeft  γ   :        σ₁ ∼ τ₁ : Δ :Γ: t
  | Right  :∀ Γ Δ t e γ σ₁ σ₂ τ₁ τ₂,(e⊢co γ : TApp σ₁ σ₂ ∼ TApp τ₁ τ₂ :Δ:Γ:t   ) -> e⊢co CoRight γ   :        σ₂ ∼ τ₂ : Δ :Γ: t
  (*
  | SComp  :∀ Γ Δ t e γ n S σ τ κ,
           ListWFCo Γ Δ t e γ σ τ -> t ⊢Ty TyFunApp(n:=n) S σ : κ  -> e⊢co CoTFApp S γ : TyFunApp S σ∼TyFunApp S τ : Δ : Γ : t
  | CoAx   :∀ Γ Δ t e n C κ γ, forall (σ₁:vec TV n) (σ₂:vec TV n), forall (ax:@AxiomDecl n C κ TV),
    ListWFCo                              Γ Δ t e γ (map TVar (vec2list σ₁)) (map TVar (vec2list σ₂)) ->
    ListWellKinded_RawHaskType TV Γ t   (map TVar (vec2list σ₁))           (vec2list κ)  ->
    ListWellKinded_RawHaskType TV Γ t   (map TVar (vec2list σ₂))           (vec2list κ)  ->
    e⊢co CoCFApp C γ : axd_σ _ _ _ ax σ₁ ∼ axd_τ _ _ _ ax σ₂ : Δ : Γ : t
  *)
  | WFCoAll  : forall Γ Δ κ (t:InstantiatedTypeEnv TV Γ) (e:InstantiatedCoercionEnv TV CV (κ::Γ) (weakCE Δ)) γ σ τ    ,
      (∀ a,          e ⊢co (       γ a) : (       σ a) ∼ (       τ a) : _ : _ : (t + a : κ))
      ->    weakICE e ⊢co (CoAll κ γ  ) : (TAll κ σ  ) ∼ (TAll κ τ  ) : Δ : Γ :  t
  | Comp   :forall Γ Δ t e γ₁ γ₂ σ₁ σ₂ τ₁ τ₂ κ,
           (t ⊢Ty TApp σ₁ σ₂:κ)->
           (e⊢co γ₁:σ₁∼τ₁:Δ:Γ:t)->
           (e⊢co γ₂:σ₂∼τ₂:Δ:Γ:t) ->
           e⊢co (CoApp γ₁ γ₂) : (TApp σ₁ σ₂) ∼ (TApp τ₁ τ₂) : Δ:Γ:t
  | CoInst :forall Γ Δ t e σ τ κ γ (v:∀ TV, InstantiatedTypeEnv TV Γ -> RawHaskType TV),
        t ⊢Ty v TV t : κ  ->
          (e⊢co γ:HaskTAll κ σ _ t ∼ HaskTAll κ τ _ t:Δ:Γ:t) ->
          e⊢co CoAppT γ (v TV t) : substT σ v TV t ∼substT τ v TV t : Δ : Γ : t
  with ListWFCo  : forall Γ (Δ:CoercionEnv Γ),
     @InstantiatedTypeEnv TV Γ ->
     InstantiatedCoercionEnv TV CV Γ Δ ->
     list (RawHaskCoer TV CV) -> list (RawHaskType TV) -> list (RawHaskType TV) -> Prop :=
  | LWFCo_nil  : ∀ Γ Δ t e ,                                                    ListWFCo Γ Δ t e nil     nil     nil
  | LWFCo_cons : ∀ Γ Δ t e a b c la lb lc, (e⊢co a : b∼c : Δ : Γ : t )->
    ListWFCo Γ Δ t e la lb lc -> ListWFCo Γ Δ t e (a::la) (b::lb) (c::lc)
  where "ice '⊢co' γ : a '∼' b : Δ : Γ : ite" := (@WFCoercion Γ Δ ite ice γ (@mkRawCoercionKind _ a b)).
End WFCo.

Definition WFCCo (Γ:TypeEnv)(Δ:CoercionEnv Γ)(γ:HaskCoercion Γ Δ)(a b:HaskType Γ) :=
  forall {TV CV:Type}(env:@InstantiatedTypeEnv TV Γ)(cenv:InstantiatedCoercionEnv TV CV Γ Δ),
    @WFCoercion _ _ Γ Δ env cenv (γ TV CV env cenv) (@mkRawCoercionKind _ (a TV env) (b TV env)).
    Notation "Δ '⊢co' γ : a '∼' b" := (@WFCCo _ Δ γ a b).
*)
```

```
(* Decidable equality on PHOAS types *)
Fixpoint compareT (n:nat){κ1}(t1:@RawHaskType (fun _ => nat) κ1){κ2}(t2:@RawHaskType (fun _ => nat) κ2) : bool :=
match t1 with
| TVar    _  x     => match t2 with TVar _ x' => if eqd_dec x x' then true else false | _ => false end
| TAll     _ y     => match t2 with TAll _ y' => compareT (S n) (y n) (y' n)          | _ => false end
| TApp   _ _ x y   => match t2 with TApp _ _ x' y' => if compareT n x x' then compareT n y y' else false | _ => false end
| TCon       tc    => match t2 with TCon tc' => if eqd_dec tc tc' then true else false | _ => false end
| TArrow           => match t2 with TArrow => true | _ => false end
| TCode      ec t  => match t2 with TCode ec' t' => if compareT n ec ec' then compareT n t t' else false | _ => false end
| TCoerc _ t1 t2 t => match t2 with TCoerc _ t1' t2' t' => compareT n t1 t1' && compareT n t2 t2' && compareT n t t' | _ =>false end
| TyFunApp tfc lt  => match t2 with TyFunApp tfc' lt' => eqd_dec tfc tfc' && compareTL n lt lt' | _ => false end
end
with compareTL (n:nat){κ1}(t1:@RawHaskTypeList (fun _ => nat) κ1){κ2}(t2:@RawHaskTypeList (fun _ => nat) κ2) : bool :=
match t1 with
| TyFunApp_nil              => match t2 with TyFunApp_nil => true | _ => false end
| TyFunApp_cons κ kl t r => match t2 with | TyFunApp_cons κ' kl' t' r' => compareT n t t' && compareTL n r r' | _ => false end
end.

Fixpoint count' (lk:list Kind)(n:nat) : IList _ (fun _ => nat) lk :=
match lk as LK return IList _ _ LK with
  | nil    => INil
  | h::t   => n::::(count' t (S n))
end.

Definition compareHT Γ κ (ht1 ht2:HaskType Γ κ) :=
  compareT (length Γ) (ht1 (fun _ => nat) (count' Γ O)) (ht2 (fun _ => nat) (count' Γ O)).

(* The PHOAS axioms
 *
 * This is not provable in Coq's logic because the Coq function space
 * is "too big" - although its only definable inhabitants are Coq
 * functions, it is not provable in Coq that all function space
 * inhabitants are definable (i.e. there are no "exotic" inhabitants).
 * This is actually an important feature of Coq: it lets us reason
 * about properties of non-computable (non-recursive) functions since
 * any property proven to hold for the entire function space will hold
 * even for those functions.  However when representing binding
 * structure using functions we would actually prefer the smaller
 * function-space of *definable* functions only.  These two axioms
```

```
 * assert that. *)
Axiom compareHT_decides : forall Γ κ (ht1 ht2:HaskType Γ κ),
  if compareHT Γ κ ht1 ht2
  then ht1=ht2
  else ht1≠ht2.
Axiom compareVars : forall Γ κ (htv1 htv2:HaskTyVar Γ κ),
  if compareHT _ _ (haskTyVarToType htv1) (haskTyVarToType htv2)
  then htv1=htv2
  else htv1≠htv2.

(* using the axioms, we can now create an EqDecidable instance for HaskType, HaskTyVar, and HaskLevel *)
Instance haskTypeEqDecidable Γ κ : EqDecidable (HaskType Γ κ).
  apply Build_EqDecidable.
  intros.
  set (compareHT_decides _ _ v1 v2) as z.
  set (compareHT Γ κ v1 v2) as q.
  destruct q as [ ] _eqn; unfold q in *; rewrite Heqb in *.
    left; auto.
    right; auto.
    Defined.

Instance haskTyVarEqDecidable Γ κ : EqDecidable (HaskTyVar Γ κ).
  apply Build_EqDecidable.
  intros.
  set (compareVars _ _ v1 v2) as z.
  set (compareHT Γ κ (haskTyVarToType v1) (haskTyVarToType v2)) as q.
  destruct q as [ ] _eqn; unfold q in *; rewrite Heqb in *.
    left; auto.
    right; auto.
    Defined.

Instance haskLevelEqDecidable Γ : EqDecidable (HaskLevel Γ).
  apply Build_EqDecidable.
  intros.
  unfold HaskLevel in *.
  apply (eqd_dec v1 v2).
  Defined.
```

```
(* ToString instance for PHOAS types *)
Fixpoint typeToString' (needparens:bool)(n:nat){κ}(t:RawHaskType (fun _ => nat) κ) {struct t} : string :=
    match t with
    | TVar    _ v          => "tv" +++ toString v
    | TCon    tc           => toString tc
    | TCoerc _ t1 t2   t   => "("+++typeToString' false n t1+++"~"
                                  +++typeToString' false n t2+++")=>"
                                  +++typeToString' needparens n t
    | TApp  _ _  t1 t2     =>
      match t1 with
        | TApp _ _ TArrow t1 =>
                    if needparens
                    then "("+++(typeToString' true n t1)+++"->"+++(typeToString' true n t2)+++")"
                    else (typeToString' true n t1)+++"->"+++(typeToString' true n t2)
        | _ =>
                    if needparens
                    then "("+++(typeToString' true n t1)+++" "+++(typeToString' false n t2)+++")"
                    else (typeToString' true n t1)+++" "+++(typeToString' false n t2)
      end
    | TArrow => "(->)"
    | TAll   k f           => let alpha := "tv"+++ toString n
                              in "(forall "+++ alpha +++ ":"+++ toString k +++")"+++
                                   typeToString' false (S n) (f n)
    | TCode  ec t          => "<["+++(typeToString' true n t)+++"]>@"+++(typeToString' false n ec)
    | TyFunApp   tfc lt    => toString tfc+++ "_" +++ toString n+++" ["+++
      (fold_left (fun x y => " \  "+++x+++y) (typeList2string false n lt) "")+++"]"
  end
  with typeList2string (needparens:bool)(n:nat){κ}(t:RawHaskTypeList κ) {struct t} : list string :=
  match t with
  | TyFunApp_nil                => nil
  | TyFunApp_cons  κ kl rhk rhkl => (typeToString' needparens n rhk)::(typeList2string needparens n rhkl)
  end.

Definition typeToString {Γ}{κ}(ht:HaskType Γ κ) : string :=
  typeToString' false (length Γ) (ht (fun _ => nat) (count' Γ 0)).

Instance TypeToStringInstance {Γ} {κ} : ToString (HaskType Γ κ) :=
  { toString := typeToString }.
```