```
(*****************************************************************************************************************************)
(* HaskWeakToStrong: convert HaskWeak 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 HaskLiteralsAndTyCons.
Require Import HaskWeakTypes.
Require Import HaskWeakVars.
Require Import HaskWeak.
Require Import HaskWeakToCore.
Require Import HaskStrongTypes.
Require Import HaskStrong.
Require Import HaskCoreVars.

Open Scope string_scope.
Definition TyVarResolver Γ   := forall wt:WeakTypeVar, ???(HaskTyVar Γ wt).
Definition CoVarResolver Γ Δ := forall wt:WeakCoerVar, ???(HaskCoVar Γ Δ).

Definition upφ {Γ}(tv:WeakTypeVar)(φ:TyVarResolver Γ) : TyVarResolver ((tv:Kind)::Γ).
  unfold TyVarResolver.
  refine (fun tv' =>
    if eqd_dec tv tv'
    then let fresh := @FreshHaskTyVar Γ tv in OK _
    else φ tv' >>= fun tv'' => OK (fun TV ite => tv'' TV (weakITE ite))).
  rewrite <- _H; apply fresh.
  Defined.

Definition upφ' {Γ}(tvs:list WeakTypeVar)(φ:TyVarResolver Γ)
  : (TyVarResolver (app (map (fun tv:WeakTypeVar => tv:Kind) tvs) Γ)).
  induction tvs.
  apply φ.
  simpl.
  apply upφ.
  apply IHtvs.
```

```
  Defined.

Definition substPhi {Γ:TypeEnv}(κ κ':Kind)(θ:HaskType Γ κ) : HaskType (κ::Γ) κ' -> HaskType Γ κ'.
  intro ht.
  refine (substT _ θ).
  clear θ.
  unfold HaskType in ht.
  intros.
  apply ht.
  apply ICons; [ idtac | apply env ].
  apply X.
  Defined.

Definition substφ {Γ:TypeEnv}(lk:list Kind)(θ:IList _ (fun κ => HaskType Γ κ) lk){κ} : HaskType (app lk Γ) κ -> HaskType Γ κ.
  induction lk.
  intro q; apply q.
  simpl.
  intro q.
  apply IHlk.
  inversion θ; subst; auto.
  inversion θ; subst.
  eapply substPhi.
  eapply weakT'.
  apply X.
  apply q.
  Defined.

(* this is a StrongAltCon plus some stuff we know about StrongAltCons which we've built ourselves *)
Record StrongAltConPlusJunk {tc:TyCon} :=
{ sacpj_sac : @StrongAltCon tc
; sacpj_φ   : forall Γ          (φ:TyVarResolver Γ  ),  (TyVarResolver (sac_Γ sacpj_sac Γ))
; sacpj_ψ   : forall Γ Δ atypes (ψ:CoVarResolver Γ Δ), CoVarResolver _ (sac_Δ sacpj_sac Γ atypes (weakCK'' Δ))
}.
Implicit Arguments StrongAltConPlusJunk [ ].
Coercion sacpj_sac : StrongAltConPlusJunk >-> StrongAltCon.

(* yes, I know, this is really clumsy *)
Variable emptyφ : TyVarResolver nil.
  Extract Inlined Constant emptyφ => "(\x -> Prelude.error ""encountered unbound tyvar!"")".

Definition mkPhi (lv:list WeakTypeVar)
```

```
  : (TyVarResolver (map (fun x:WeakTypeVar => x:Kind) lv)).
  set (upφ'(Γ:=nil) lv emptyφ) as φ'.
  rewrite <- app_nil_end in φ'.
  apply φ'.
  Defined.

Definition dataConExKinds dc := vec_map (fun x:WeakTypeVar => (x:Kind)) (list2vec (dataConExTyVars dc)).
Definition tyConKinds     tc := vec_map (fun x:WeakTypeVar => (x:Kind)) (list2vec (tyConTyVars tc)).

Definition fixkind {κ}(tv:WeakTypeVar) := weakTypeVar tv κ.
Notation " ‘ x " := (@fixkind _ x) (at level 100).

Ltac matchThings T1 T2 S :=
  destruct (eqd_dec T1 T2) as [matchTypeVars_pf|];
   [ idtac | apply (Error (S +++ toString T1 +++ " " +++ toString T2)) ].

Definition mkTAll' {κ}{Γ} : HaskType (κ :: Γ) ★-> (forall TV (ite:InstantiatedTypeEnv TV Γ), TV κ -> RawHaskType TV ★).
  intros.
  unfold InstantiatedTypeEnv in ite.
  apply X.
  apply (X0::::ite).
  Defined.

Definition mkTAll {κ}{Γ} : HaskType (κ :: Γ) ★-> HaskType Γ ★.
  intro.
  unfold HaskType.
  intros.
  apply (TAll κ).
  eapply mkTAll'.
  apply X.
  apply X0.
  Defined.

Definition weakTypeToType : forall {Γ:TypeEnv}(φ:TyVarResolver Γ)(t:WeakType), ???(HaskTypeOfSomeKind Γ).
  refine (fix weakTypeToType {Γ:TypeEnv}(φ:TyVarResolver Γ)(t:WeakType) {struct t} : ???(HaskTypeOfSomeKind Γ) :=
  addErrorMessage ("weakTypeToType " +++ toString t)
  match t with
    | WFunTyCon         => let case_WFunTyCon := tt in OK (haskTypeOfSomeKind (fun TV ite => TArrow))
    | WTyCon      tc    => let case_WTyCon := tt    in _
    | WClassP   c lt    => let case_WClassP := tt   in Error "weakTypeToType: WClassP not implemented"
    | WIParam _ ty      => let case_WIParam := tt   in Error "weakTypeToType: WIParam not implemented"
```

```
  | WAppTy  t1 t2     => let case_WAppTy := tt    in weakTypeToType _ φ t1 >>= fun t1' => weakTypeToType _ φ t2 >>= fun t2' => _
  | WTyVarTy  v       => let case_WTyVarTy := tt  in φ v >>= fun v' => _
  | WForAllTy wtv t   => let case_WForAllTy := tt in weakTypeToType _ (upφ wtv φ) t >>= fun t => _
  | WCodeTy ec tbody  => let case_WCodeTy := tt   in weakTypeToType _ φ tbody >>= fun tbody' => φ (@fixkind ★ec) >>= fun ec' => _
  | WCoFunTy t1 t2 t3 => let case_WCoFunTy := tt  in
    weakTypeToType _ φ t1 >>= fun t1' =>
    weakTypeToType _ φ t2 >>= fun t2' =>
    weakTypeToType _ φ t3 >>= fun t3' => _
  | WTyFunApp   tc lt =>
    ((fix weakTypeListToTypeList (lk:list Kind) (lt:list WeakType)
      { struct lt } : ???(forall TV (ite:InstantiatedTypeEnv TV Γ), @RawHaskTypeList TV lk) :=
      match lt with
        | nil    => match lk as LK return ???(forall TV (ite:InstantiatedTypeEnv TV Γ), @RawHaskTypeList TV LK) with
                      | nil => OK (fun TV _ => TyFunApp_nil)
                      | _   => Error "WTyFunApp not applied to enough types"
                    end
        | tx::lt' => weakTypeToType Γ φ tx >>= fun t' =>
                     match lk as LK return ???(forall TV (ite:InstantiatedTypeEnv TV Γ), @RawHaskTypeList TV LK) with
                       | nil    => Error "WTyFunApp applied to too many types"
                       | k::lk' => weakTypeListToTypeList lk' lt' >>= fun rhtl' =>
                                   let case_weakTypeListToTypeList := tt in _
                     end
      end
    ) (fst (tyFunKind tc)) lt) >>= fun lt' => let case_WTyFunApp := tt in  _
end ); clear weakTypeToType.
apply ConcatenableString.

destruct case_WTyVarTy.
  apply (addErrorMessage "case_WTyVarTy").
  apply OK.
  exact (haskTypeOfSomeKind (fun TV env => TVar (v' TV env))).

destruct case_WAppTy.
  apply (addErrorMessage "case_WAppTy").
  destruct t1' as  [k1' t1'].
  destruct t2' as [k2' t2'].
  set ("tried to apply type "+++toString t1'+++" of kind "+++toString k1'+++" to type "+++
    toString t2'+++" of kind "+++toString k2') as err.
  destruct k1';
    try (matchThings k1'1 k2' "Kind mismatch in WAppTy: ";
      subst; apply OK; apply (haskTypeOfSomeKind (fun TV env => TApp (t1' TV env) (t2' TV env)))).
```

```
    apply (Error ("Kind mismatch in WAppTy: "+++err)).

destruct case_weakTypeListToTypeList.
  apply (addErrorMessage "case_weakTypeListToTypeList").
  destruct t' as [ k' t' ].
  matchThings k k' "Kind mismatch in weakTypeListToTypeList".
  subst.
  apply (OK (fun TV ite => TyFunApp_cons _ _ (t' TV ite) (rhtl' TV ite))).

destruct case_WTyFunApp.
  apply (addErrorMessage "case_WTyFunApp").
  apply OK.
  eapply haskTypeOfSomeKind.
  unfold HaskType; intros.
  apply TyFunApp.
  apply lt'.
  apply X.

destruct case_WTyCon.
  apply (addErrorMessage "case_WTyCon").
  apply OK.
  eapply haskTypeOfSomeKind.
  unfold HaskType; intros.
  apply (TCon tc).

destruct case_WCodeTy.
  apply (addErrorMessage "case_WCodeTy").
  destruct tbody'.
  matchThings κ ★"Kind mismatch in WCodeTy: ".
  apply OK.
  eapply haskTypeOfSomeKind.
  unfold HaskType; intros.
  apply TCode.
  apply (TVar (ec' TV X)).
  subst.
  apply h.
  apply X.

destruct case_WCoFunTy.
  apply (addErrorMessage "case_WCoFunTy").
  destruct t1' as [ k1' t1' ].
```

```
    destruct t2' as [ k2' t2' ].
    destruct t3' as [ k3' t3' ].
    matchThings k1' k2' "Kind mismatch in arguments of WCoFunTy".
    subst.
    matchThings k3' ★"Kind mismatch in result of WCoFunTy".
    subst.
    apply OK.
    apply (haskTypeOfSomeKind (t1' ⇝ t2' ⇒t3')).

  destruct case_WForAllTy.
    apply (addErrorMessage "case_WForAllTy").
    destruct t1.
    matchThings ★ κ "Kind mismatch in WForAllTy: ".
    subst.
    apply OK.
    apply (@haskTypeOfSomeKind _ ★).
    apply (@mkTAll wtv).
    apply h.
    Defined.

(* information about a datacon/literal/default which is common to all instances of a branch with that tag *)
Section StrongAltCon.
  Context (tc : TyCon)(dc:DataCon tc).

Definition weakTypeToType' {Γ} : IList Kind (HaskType Γ) (vec2list (tyConKinds tc))
 -> WeakType →???(HaskType (app (vec2list (dataConExKinds dc)) Γ) ★.
  intro avars.
  intro ct.
  apply (addErrorMessage "weakTypeToType'").
  set (ilmap (@weakT' _ (vec2list (dataConExKinds dc))) avars) as avars'.
  set (@substφ _ _ avars') as q.
  set (upφ' (tyConTyVars tc)  (mkPhi (dataConExTyVars dc))) as φ'.
  set (@weakTypeToType _ φ' ct) as t.
  destruct t as [| t]; try apply (Error error_message).
  destruct t as [tk t].
  matchThings tk ★"weakTypeToType'".
  subst.
  apply OK.
  set (@weakT'' _ Γ _ t) as t'.
  set (@lamer _ _ _ _ t') as t''.
  fold (tyConKinds tc) in t''.
```

```
  fold (dataConExKinds dc) in t''.
  apply q.
  clear q.
  unfold tyConKinds.
  unfold dataConExKinds.
  rewrite <- vec2list_map_list2vec.
  rewrite <- vec2list_map_list2vec.
  rewrite vec2list_list2vec.
  rewrite vec2list_list2vec.
  apply t''.
  Defined.

Definition mkStrongAltCon : @StrongAltCon tc.
  refine
   {| sac_altcon      := WeakDataAlt dc
    ; sac_numCoerVars := length (dataConCoerKinds dc)
    ; sac_numExprVars := length (dataConFieldTypes dc)
    ; sac_ekinds      := dataConExKinds dc
    ; sac_coercions   := fun Γ avars => let case_sac_coercions := tt in _
    ; sac_types       := fun Γ avars => let case_sac_types := tt in _
    |}.

  destruct case_sac_coercions.
    refine (vec_map _ (list2vec (dataConCoerKinds dc))).
    intro.
    destruct X.
    unfold tyConKind in avars.
    set (@weakTypeToType' Γ) as q.
    unfold tyConKinds in q.
    rewrite <- vec2list_map_list2vec in q.
    rewrite vec2list_list2vec in q.
    apply (
      match
        q avars w >>= fun t1 =>
        q avars w0 >>= fun t2 =>
          OK (mkHaskCoercionKind t1 t2)
      with
        | Error s => Prelude_error s
        | OK y => y
      end).
```

```
  destruct case_sac_types.
    refine (vec_map _ (list2vec (dataConFieldTypes dc))).
    intro X.
    unfold tyConKind in avars.
    set (@weakTypeToType' Γ) as q.
    unfold tyConKinds in q.
    rewrite <- vec2list_map_list2vec in q.
    rewrite vec2list_list2vec in q.
    set (q avars X) as y.
    apply (match y with
             | Error s =>Prelude_error s
             | OK y' => y'
           end).
    Defined.


Lemma weakCV' : forall {Γ}{Δ} Γ',
   HaskCoVar Γ Δ
   -> HaskCoVar (app Γ' Γ) (weakCK'' Δ).
  intros.
  unfold HaskCoVar in *.
  intros; apply (X TV CV).
  apply ilist_chop' in env; auto.
  unfold InstantiatedCoercionEnv in *.
  unfold weakCK'' in cenv.
  destruct Γ'.
  rewrite <- map_preserves_length in cenv.
  apply cenv.
  rewrite <- map_preserves_length in cenv.
  apply cenv.
  Defined.

Definition mkStrongAltConPlusJunk : StrongAltConPlusJunk tc.
    refine
     {| sacpj_sac     := mkStrongAltCon
      ; sacpj_φ       := fun Γ φ => (fun htv => φ htv >>= fun htv' => OK (weakV' htv'))
      ; sacpj_ψ       :=
      fun Γ Δ avars ψ => (fun htv => ψ htv >>= fun htv' => OK (_ (weakCV' (vec2list (sac_ekinds mkStrongAltCon)) htv')))
      |}.
    intro.
    unfold sac_Γ.
```

```
    unfold HaskCoVar in *.
    intros.
    apply (x TV CV env).
    simpl in cenv.
    unfold sac_Δ in *.
    unfold InstantiatedCoercionEnv in *.
    apply vec_chop' in cenv.
    apply cenv.
    Defined.

  Lemma weakCK'_nil_inert : forall Γ Δ, (@weakCK'' Γ (@nil Kind)) Δ = Δ.
    intros.
    induction Δ.
    reflexivity.
    simpl.
    rewrite IHΔ.
    reflexivity.
    Qed.

End StrongAltCon.

Definition mkStrongAltConPlusJunk' (tc : TyCon)(alt:WeakAltCon) : ???(@StrongAltConPlusJunk tc).
  destruct alt.
  set (c:DataCon _) as dc.
  set ((dataConTyCon c):TyCon) as tc' in *.
  set (eqd_dec tc tc') as eqpf; destruct eqpf;
    [ idtac
      | apply (Error ("in a case of tycon "+++toString tc+++", found a branch with datacon "+++toString (dc:CoreDataCon))) ]; subst.
  apply OK.
  eapply mkStrongAltConPlusJunk.
  simpl in *.
  apply dc.

  apply OK; refine {| sacpj_sac := {|
                      sac_ekinds  := vec_nil ; sac_coercions := fun _ _ => vec_nil ; sac_types := fun _ _ => vec_nil
                    ; sac_altcon := WeakLitAlt h
                    |} |}.
           intro; intro φ; apply φ.
           intro; intro; intro; intro ψ. simpl. unfold sac_Γ; simpl. unfold sac_Δ; simpl.
           rewrite weakCK'_nil_inert. apply ψ.
  apply OK; refine {| sacpj_sac := {|
```

```
                    sac_ekinds := vec_nil ; sac_coercions := fun _ _ => vec_nil ; sac_types := fun _ _ => vec_nil
                     ; sac_altcon := WeakDEFAULT |} |}.
            intro; intro φ; apply φ.
            intro; intro; intro; intro ψ. simpl. unfold sac_Γ; simpl. unfold sac_Δ; simpl.
            rewrite weakCK'_nil_inert. apply ψ.
Defined.

Definition weakExprVarToWeakType : WeakExprVar -> WeakType :=
  fun wev => match wev with weakExprVar _ t => t end.
  Coercion weakExprVarToWeakType : WeakExprVar >-> WeakType.

Variable weakCoercionToHaskCoercion : forall Γ Δ κ, WeakCoercion -> HaskCoercion Γ Δ κ.

Definition weakψ {Γ}{Δ:CoercionEnv Γ} {κ}(ψ:WeakCoerVar -> ???(HaskCoVar Γ Δ)) :
  WeakCoerVar -> ???(HaskCoVar Γ (κ::Δ)).
  intros.
  refine (ψ X >>= _).
  unfold HaskCoVar.
  intros.
  apply OK.
  intros.
  inversion cenv; auto.
  Defined.

(* attempt to "cast" an expression by simply checking if it already had the desired type, and failing otherwise *)
Definition castExpr (we:WeakExpr)(err_msg:string) {Γ} {Δ} {ξ} {τ} τ' (e:@Expr _ CoreVarEqDecidable Γ Δ ξ τ)
  : ???(@Expr _ CoreVarEqDecidable Γ Δ ξ τ').
  apply (addErrorMessage ("castExpr " +++ err_msg)).
  intros.
  destruct τ  as [τ  l].
  destruct τ' as [τ' l'].
  destruct (eqd_dec l l'); [ idtac
    | apply (Error ("level mismatch in castExpr, invoked by "+++err_msg+++eol+++
                    "  got: " +++(fold_left (fun x y => y+++","+++y) (map (toString ∘ haskTyVarToType) l) "")+++eol+++
                    "  wanted: "+++(fold_left (fun x y => x+++","+++y) (map (toString ∘ haskTyVarToType) l') "")
    )) ].
  destruct (eqd_dec τ τ'); [ idtac
    | apply (Error ("type mismatch in castExpr, invoked by "+++err_msg+++eol+++
                    "  got: " +++toString τ+++eol+++
                    "  wanted: "+++toString τ'
    )) ].
```

```
  subst.
  apply OK.
  apply e.
  Defined.

Definition coVarKind (wcv:WeakCoerVar) : Kind :=
  match wcv with weakCoerVar _ κ _ _ => κ end.
  Coercion coVarKind : WeakCoerVar >-> Kind.

Definition weakTypeToTypeOfKind : forall {Γ:TypeEnv}(φ:TyVarResolver Γ)(t:WeakType)(κ:Kind), ???(HaskType Γ κ).
  intros.
  set (weakTypeToType φ t) as wt.
  destruct wt; try apply (Error error_message).
  destruct h.
  matchThings κ κ0 ("Kind mismatch in weakTypeToTypeOfKind in ").
  subst.
  apply OK.
  apply h.
  Defined.

Fixpoint varsTypes {Γ}(t:Tree ??(WeakExprVar * WeakExpr))(φ:TyVarResolver Γ) : Tree ??(CoreVar * HaskType Γ ★) :=
  match t with
    | T_Leaf None            => []
    | T_Leaf (Some (wev,e))  => match weakTypeToTypeOfKind φ wev ★with
                                  | OK    t' => [((wev:CoreVar),t')]
                                  | _        => []
                                end
    | T_Branch b1 b2         => (varsTypes b1 φ),,(varsTypes b2 φ)
  end.

Fixpoint mkAvars {Γ}(wtl:list WeakType)(lk:list Kind)(φ:TyVarResolver Γ) : ???(IList Kind (HaskType Γ) lk) :=
match lk as LK return ???(IList Kind (HaskType Γ) LK) with
  | nil => match wtl with
             | nil => OK INil
             | _   => Error "length mismatch in mkAvars"
           end
  | k::lk' => match wtl with
                | nil => Error "length mismatch in mkAvars"
                | wt::wtl' =>
                  weakTypeToTypeOfKind φ wt k >>= fun t =>
                    mkAvars wtl' lk' φ >>= fun rest =>
```

```
                    OK (ICons _ _ t rest)
              end
end.

Fixpoint update_ig (ig:CoreVar -> bool) (vars:list CoreVar) : CoreVar -> bool :=
  match vars with
    | nil => ig
    | v::vars' =>
      fun v' =>
        if eqd_dec v v'
          then false
            else update_ig ig vars' v'
  end.

(* does the specified variable occur free in the expression? *)
Fixpoint doesWeakVarOccur (wev:WeakExprVar)(me:WeakExpr) : bool :=
  match me with
    | WELit    _        => false
    | WEVar    cv       => if eqd_dec (wev:CoreVar) (cv:CoreVar) then true else false
    | WECast   e co     =>                              doesWeakVarOccur wev e
    | WENote   n e      =>                              doesWeakVarOccur wev e
    | WETyApp  e t      =>                              doesWeakVarOccur wev e
    | WECoApp  e co     =>                              doesWeakVarOccur wev e
    | WEBrak _ ec e _   =>                              doesWeakVarOccur wev e
    | WEEsc  _ ec e _   =>                              doesWeakVarOccur wev e
    | WECSP  _ ec e _   =>                              doesWeakVarOccur wev e
    | WELet    cv e1 e2 => doesWeakVarOccur wev e1 || (if eqd_dec (wev:CoreVar) (cv:CoreVar)then false else doesWeakVarOccur wev e2)
    | WEApp    e1 e2    => doesWeakVarOccur wev e1 || doesWeakVarOccur wev e2
    | WELam    cv e     => if eqd_dec (wev:CoreVar) (cv:CoreVar) then false else doesWeakVarOccur wev e
    | WETyLam  cv e     => doesWeakVarOccur wev e
    | WECoLam  cv e     => doesWeakVarOccur wev e
    | WECase vscrut escrut tbranches tc avars alts =>
      doesWeakVarOccur wev escrut ||
      if eqd_dec (wev:CoreVar) (vscrut:CoreVar) then false else
        ((fix doesWeakVarOccurAlts alts {struct alts} : bool :=
          match alts with
            | T_Leaf  None                                          => false
            | T_Leaf (Some (WeakDEFAULT,_,_,_,e))                       => doesWeakVarOccur wev e
            | T_Leaf (Some (WeakLitAlt lit,_,_,_,e))                    => doesWeakVarOccur wev e
            | T_Leaf (Some ((WeakDataAlt dc), tvars, cvars, evars,e))  => doesWeakVarOccur wev e  (* FIXME!!! *)
            | T_Branch b1 b2                                        => doesWeakVarOccurAlts b1 || doesWeakVarOccurAlts b2
```

```
          end) alts)
    | WELetRec mlr e =>
      doesWeakVarOccur wev e ||
      (fix doesWeakVarOccurLetRec (mlr:Tree ??(WeakExprVar * WeakExpr)) : bool :=
      match mlr with
        | T_Leaf None          => false
        | T_Leaf (Some (cv,e)) => if eqd_dec (wev:CoreVar) (cv:CoreVar) then false else doesWeakVarOccur wev e
        | T_Branch b1 b2       => doesWeakVarOccurLetRec b1 || doesWeakVarOccurLetRec b2
      end) mlr
  end.
Fixpoint doesWeakVarOccurAlts (wev:WeakExprVar)
  (alts:Tree ??(WeakAltCon * list WeakTypeVar * list WeakCoerVar * list WeakExprVar * WeakExpr)) : bool :=
  match alts with
    | T_Leaf  None                                          => false
    | T_Leaf (Some (WeakDEFAULT,_,_,_,e))                   => doesWeakVarOccur wev e
    | T_Leaf (Some (WeakLitAlt lit,_,_,_,e))                => doesWeakVarOccur wev e
    | T_Leaf (Some ((WeakDataAlt dc), tvars, cvars, evars,e))  => doesWeakVarOccur wev e  (* FIXME!!! *)
    | T_Branch b1 b2                                        => doesWeakVarOccurAlts wev b1 || doesWeakVarOccurAlts wev b2
  end.

(*Definition ensureCaseBindersAreNotUsed (we:WeakExpr) : UniqM WeakExpr := FIXME *)

Definition weakExprToStrongExpr : forall
    (Γ:TypeEnv)
    (Δ:CoercionEnv Γ)
    (φ:TyVarResolver Γ)
    (ψ:CoVarResolver Γ Δ)
    (ξ:CoreVar -> LeveledHaskType Γ ★)
    (ig:CoreVar -> bool)
    (τ:HaskType Γ ★)
    (lev:HaskLevel Γ),
    WeakExpr -> ???(@Expr _ CoreVarEqDecidable Γ Δ ξ (τ @@ lev) ).
  refine ((
    fix weakExprToStrongExpr
    (Γ:TypeEnv)
    (Δ:CoercionEnv Γ)
    (φ:TyVarResolver Γ)
    (ψ:CoVarResolver Γ Δ)
    (ξ:CoreVar -> LeveledHaskType Γ ★)
    (ig:CoreVar -> bool)
    (τ:HaskType Γ ★)
```

```
(lev:HaskLevel Γ)
(we:WeakExpr) : ???(@Expr _ CoreVarEqDecidable Γ Δ ξ (τ @@ lev) )  :=
addErrorMessage ("in weakExprToStrongExpr " +++ toString we)
match we with

| WEVar   v                        => if ig v
                                       then OK (EGlobal Γ Δ ξ (τ@@lev) v)
                                       else castExpr we ("WEVar "+++toString (v:CoreVar)) (τ @@ lev) (EVar Γ Δ ξ v)

| WELit   lit                      => castExpr we ("WELit "+++toString lit) (τ @@ lev) (ELit Γ Δ ξ lit lev)

| WELam   ev ebody                 => weakTypeToTypeOfKind φ ev ★>>= fun tv =>
                                     weakTypeOfWeakExpr ebody >>= fun tbody =>
                                       weakTypeToTypeOfKind φ tbody ★>>= fun tbody' =>
                                         let ξ' := update_ξ ξ lev (((ev:CoreVar),tv)::nil) in
                                         let ig' := update_ig ig ((ev:CoreVar)::nil) in
                                           weakExprToStrongExpr Γ Δ φ ψ ξ' ig' tbody' lev ebody >>= fun ebody' =>
                                             castExpr we "WELam" (τ@@lev) (ELam Γ Δ ξ tv tbody' lev ev ebody')

| WEBrak  _ ec e tbody             => φ (‘ec) >>= fun ec' =>
                                     weakTypeToTypeOfKind φ tbody ★>>= fun tbody' =>
                                       weakExprToStrongExpr Γ Δ φ ψ ξ ig tbody' ((ec')::lev) e >>= fun e' =>
                                         castExpr we "WEBrak" (τ@@lev) (EBrak Γ Δ ξ ec' tbody' lev e')

| WEEsc   _ ec e tbody             => φ ec >>= fun ec'' =>
                                    weakTypeToTypeOfKind φ tbody ★>>= fun tbody' =>
                                    match lev with
                                      | nil       => Error "ill-leveled escapification"
                                      | ec'::lev' => weakExprToStrongExpr Γ Δ φ ψ ξ ig (<[ ec' |- tbody' ]>) lev' e
                                        >>= fun e' => castExpr we "WEEsc" (τ@@lev) (EEsc Γ Δ ξ ec' tbody' lev' e')
                                    end

| WECSP   _ ec e tbody             => Error "FIXME: CSP not supported beyond HaskWeak stage"

| WENote  n e                      => weakExprToStrongExpr Γ Δ φ ψ ξ ig τ lev e >>= fun e' => OK (ENote _ _ _ _ n e')

| WELet   v ve  ebody              => weakTypeToTypeOfKind φ v ★ >>= fun tv =>
                                     weakExprToStrongExpr Γ Δ φ ψ ξ ig tv lev ve >>= fun ve' =>
                                       weakExprToStrongExpr Γ Δ φ ψ (update_ξ ξ lev (((v:CoreVar),tv)::nil))
                                            (update_ig ig ((v:CoreVar)::nil)) τ lev ebody
                                       >>= fun ebody' =>
```

```
                                        OK (ELet _ _ _ tv _ lev (v:CoreVar) ve' ebody')

| WEApp   e1 e2                      => weakTypeOfWeakExpr e2 >>= fun t2 =>
                                        weakTypeToTypeOfKind φ t2 ★>>= fun t2' =>
                                          weakExprToStrongExpr Γ Δ φ ψ ξ ig t2' lev e2 >>= fun e2' =>
                                            weakExprToStrongExpr Γ Δ φ ψ ξ ig (t2'--->τ) lev e1 >>= fun e1' =>
                                              OK (EApp _ _ _ _ _ _ e1' e2')

| WETyLam tv e                     => let φ' := upφ tv φ in
                                        weakTypeOfWeakExpr e >>= fun te =>
                                          weakTypeToTypeOfKind φ' te ★>>= fun τ' =>
                                            weakExprToStrongExpr _ (weakCE Δ) φ'
                                              (fun x => (ψ x) >>= fun y => OK (weakCV y)) (weakLT∘ξ) ig _ (weakL lev) e
                                                >>= fun e' => castExpr we "WETyLam2" _ (ETyLam Γ Δ ξ tv (mkTAll' τ') lev e')

| WETyApp e t                      => weakTypeOfWeakExpr e >>= fun te =>
                                      match te with
                                        | WForAllTy wtv te' =>
                                          let φ' := upφ wtv φ in
                                            weakTypeToTypeOfKind φ' te' ★>>= fun te'' =>
                                              weakExprToStrongExpr Γ Δ φ ψ ξ ig (mkTAll te'') lev e >>= fun e' =>
                                                weakTypeToTypeOfKind φ t (wtv:Kind) >>= fun t' =>
                                                  castExpr we "WETyApp" _ (ETyApp Γ Δ wtv (mkTAll' te'') t' ξ lev e')
                                        | _                 => Error ("weakTypeToType: WETyApp body with type "+++toString te)
                                      end

| WECoApp e co                     => weakTypeOfWeakExpr e >>= fun te =>
                                      match te with
                                        | WCoFunTy t1 t2 t3 =>
                                          weakTypeToType φ t1 >>= fun t1' =>
                                            match t1' with
                                              haskTypeOfSomeKind κ t1'' =>
                                              weakTypeToTypeOfKind φ t2 κ >>= fun t2'' =>
                                                weakTypeToTypeOfKind φ t3 ★>>= fun t3'' =>
                                                  weakExprToStrongExpr Γ Δ φ ψ ξ ig (t1'' ∼∼ t2'' ⇒τ) lev e >>= fun e' =>
                                                    castExpr we "WECoApp" _ e' >>= fun e'' =>
                                                      OK (ECoApp Γ Δ κ t1'' t2''
                                                        (weakCoercionToHaskCoercion _ _ _ co) τ ξ lev e'')
                                            end
                                        | _                 => Error ("weakTypeToType: WECoApp body with type "+++toString te)
                                      end
```

```
| WECoLam cv e                        => let (_,_,t1,t2) := cv in
                                         weakTypeOfWeakExpr e >>= fun te =>
                                           weakTypeToTypeOfKind φ te ★>>= fun te' =>
                                             weakTypeToTypeOfKind φ t1 cv >>= fun t1' =>
                                               weakTypeToTypeOfKind φ t2 cv >>= fun t2' =>
                                                 weakExprToStrongExpr Γ (_ :: Δ) φ (weakψ ψ) ξ ig te' lev e >>= fun e' =>
                                                   castExpr we "WECoLam" _ (ECoLam Γ Δ cv te' t1' t2' ξ lev e')

| WECast  e co                      => let (t1,t2) := weakCoercionTypes co in
                                         weakTypeToTypeOfKind φ t1 ★>>= fun t1' =>
                                           weakTypeToTypeOfKind φ t2 ★>>= fun t2' =>
                                             weakExprToStrongExpr Γ Δ φ ψ ξ ig t1' lev e >>= fun e' =>
                                               castExpr we "WECast" _
                                                 (ECast Γ Δ ξ t1' t2' (weakCoercionToHaskCoercion _ _ _ co) lev e')

| WELetRec rb   e                   =>
  let ξ' := update_ξ ξ lev _ in
  let ig' := update_ig ig (map (fun x:(WeakExprVar*_) => (fst x):CoreVar) (leaves rb)) in
  let binds :=
    (fix binds (t:Tree ??(WeakExprVar * WeakExpr))
      : ???(ELetRecBindings Γ Δ ξ' lev (varsTypes t φ)) :=
    match t with
      | T_Leaf None           => let case_nil := tt in OK (ELR_nil _ _ _ _)
      | T_Leaf (Some (wev,e)) => let case_some := tt in (fun e' => _) (fun τ => weakExprToStrongExpr Γ Δ φ ψ ξ' ig' τ lev e)
      | T_Branch b1 b2        =>
        binds b1 >>= fun b1' =>
          binds b2 >>= fun b2' =>
            OK (ELR_branch Γ Δ ξ' lev _ _ b1' b2')
    end) rb
  in binds >>= fun binds' =>
       weakExprToStrongExpr Γ Δ φ ψ ξ' ig' τ lev e >>= fun e' =>
         OK (ELetRec Γ Δ ξ lev τ _ binds' e')

| WECase vscrut escrut tbranches tc avars alts =>
    weakTypeOfWeakExpr escrut >>= fun tscrut =>
      weakTypeToTypeOfKind φ tscrut ★>>= fun tscrut' =>
        if doesWeakVarOccurAlts vscrut alts
        then Error "encountered a Case which actually used its binder - these should have been desugared away!!"
        else mkAvars avars (tyConKind tc) φ >>= fun avars' =>
            weakTypeToTypeOfKind φ tbranches ★ >>= fun tbranches' =>
```

```
            (fix mkTree (t:Tree ??(WeakAltCon*list WeakTypeVar*list WeakCoerVar*list WeakExprVar*WeakExpr)) : ???(Tree
                ??{ sac : _ & {scb : StrongCaseBranchWithVVs CoreVar CoreVarEqDecidable tc avars' sac &
                  Expr (sac_Γ sac Γ) (sac_Δ sac Γ avars' (weakCK'' Δ))(scbwv_ξ scb ξ lev)(weakLT' (tbranches' @@  lev))}}) :=
              match t with
                | T_Leaf None           => OK []
                | T_Leaf (Some (ac,extyvars,coervars,exprvars,ebranch)) =>
                  mkStrongAltConPlusJunk' tc ac >>= fun sac =>
                    list2vecOrFail (map (fun ev:WeakExprVar => ev:CoreVar) exprvars) _ (fun _ _ => "WECase")
                    >>= fun exprvars' =>
                      (let case_pf := tt in _) >>= fun pf =>
                      let scb := @Build_StrongCaseBranchWithVVs CoreVar CoreVarEqDecidable tc Γ avars' sac exprvars' pf in
                        weakExprToStrongExpr (sac_Γ sac Γ) (sac_Δ sac Γ avars' (weakCK'' Δ)) (sacpj_φ sac _ φ)
                        (sacpj_ψ sac _ _ avars' ψ)
                        (scbwv_ξ scb ξ lev)
                        (update_ig ig (map (@fst _ _) (vec2list (scbwv_varstypes scb))))
                        (weakT' tbranches') (weakL' lev) ebranch >>= fun ebranch' =>
                          let case_case := tt in OK [ _ ]
                | T_Branch b1 b2        =>
                  mkTree b1 >>= fun b1' =>
                    mkTree b2 >>= fun b2' =>
                      OK (b1',,b2')
              end) alts >>= fun tree =>

              weakExprToStrongExpr Γ Δ φ ψ ξ ig (caseType tc avars') lev escrut >>= fun escrut' =>
                castExpr we "ECase" (τ@@lev) (ECase Γ Δ ξ lev tc tbranches' avars' escrut' tree)
end)); try clear binds; try apply ConcatenableString.

destruct case_some.
apply (addErrorMessage "case_some").
  simpl.
  destruct (weakTypeToTypeOfKind φ wev ★); try apply (Error error_message).
  matchThings h (unlev (ξ' wev)) "LetRec".
  destruct wev.
  rewrite matchTypeVars_pf.
  clear matchTypeVars_pf.
  set (e' (unlev (ξ' (weakExprVar c w)))) as e''.
  destruct e''; try apply (Error error_message).
  apply OK.
  apply ELR_leaf.
  unfold ξ'.
  simpl.
```

```
  induction (leaves (varsTypes rb φ)).
    simpl; auto.
    destruct (ξ c).
    simpl.
  apply e1.

destruct case_pf.
  set (distinct_decidable (vec2list exprvars')) as dec.
  destruct dec; [ idtac | apply (Error "malformed HaskWeak: case branch with variables repeated") ].
  apply OK; auto.

destruct case_case.
  exists sac.
  exists scb.
  apply ebranch'.

Defined.
```