```
(*****************************************************************************************************************************)
(* HaskWeakTypes: types HaskWeak                                                                                             *)
(*****************************************************************************************************************************)

Generalizable All Variables.
Require Import Preamble.
Require Import General.
Require Import Coq.Strings.String.
Require Import Coq.Lists.List.
Require Import HaskKinds.
Require Import HaskLiteralsAndTyCons.
Require Import HaskCoreVars.

(* a WeakTypeVar merely wraps a CoreVar and includes its Kind *)
Inductive WeakTypeVar := weakTypeVar : CoreVar -> Kind -> WeakTypeVar.

(*
 * WeakType is much like CoreType, but:
 *   1. avoids mutually-inductive definitions
 *   2. gives special cases for the tycons which have their own typing rules so we can pattern-match on them
 *   3. separates type functions from type constructors, and uses a normal "AppTy" for applying the latter
 *)
Inductive WeakType :=
| WTyVarTy  : WeakTypeVar                                  -> WeakType
| WAppTy    : WeakType           ->            WeakType -> WeakType
| WTyFunApp : TyFun              ->      list WeakType -> WeakType
| WTyCon    : TyCon                                       -> WeakType
| WFunTyCon :                                                WeakType    (* never use (WTyCon ArrowCon);    always use this! *)
| WCodeTy   : WeakTypeVar        ->            WeakType -> WeakType    (* never use the raw tycon *)
| WCoFunTy  : WeakType           -> WeakType -> WeakType -> WeakType
| WForAllTy : WeakTypeVar        ->            WeakType -> WeakType
| WClassP   : Class_             ->      list WeakType -> WeakType
| WIParam   : CoreIPName CoreName ->           WeakType -> WeakType.

(* EqDecidable instances for WeakType *)
Instance WeakTypeVarEqDecidable : EqDecidable WeakTypeVar.
  apply Build_EqDecidable.
  intros.
  destruct v1 as [cv1 k1].
  destruct v2 as [cv2 k2].
  destruct (eqd_dec cv1 cv2); subst.
```

```
    destruct (eqd_dec k1 k2); subst.
    left; auto.
    right; intro; apply n; inversion H; subst; auto.
    right; intro; apply n; inversion H; subst; auto.
    Defined.

(* a WeakCoerVar just wraps a CoreVar and tags it with the pair of types amongst which it coerces *)
Inductive WeakCoerVar := weakCoerVar : CoreVar -> Kind -> WeakType -> WeakType -> WeakCoerVar.

Inductive WeakCoercion : Type :=
| WCoVar          : WeakCoerVar                                  -> WeakCoercion (* g      *)
| WCoType         : WeakType                                     -> WeakCoercion (* τ      *)
| WCoApp          : WeakCoercion -> WeakCoercion                 -> WeakCoercion (* γ γ    *)
| WCoAppT         : WeakCoercion -> WeakType                     -> WeakCoercion (* γ@v    *)
| WCoAll          : Kind  -> (WeakTypeVar -> WeakCoercion)       -> WeakCoercion (* ∀a:κ.γ *)
| WCoSym          : WeakCoercion                                 -> WeakCoercion (* sym    *)
| WCoComp         : WeakCoercion -> WeakCoercion                 -> WeakCoercion (* ◯      *)
| WCoLeft         : WeakCoercion                                 -> WeakCoercion (* left   *)
| WCoRight        : WeakCoercion                                 -> WeakCoercion (* right  *)
| WCoUnsafe       : WeakType -> WeakType                         -> WeakCoercion (* unsafe *)
(*| WCoCFApp        : ∀ n, CoFunConst n -> vec WeakCoercion n       -> WeakCoercion (* C   γ8319 *)*)
(*| WCoTFApp        : ∀ n, TyFunConst n -> vec WeakCoercion n       -> WeakCoercion (* S_n γ8319 *)*)
.

Fixpoint weakCoercionTypes (wc:WeakCoercion) : WeakType * WeakType :=
match wc with
| WCoVar     (weakCoerVar _ _ t1 t2) => (t1,t2)
| WCoType    t                       => (WFunTyCon,WFunTyCon)   (* FIXME!!! *)
| WCoApp     c1 c2                   => (WFunTyCon,WFunTyCon)   (* FIXME!!! *)
| WCoAppT    c t                     => (WFunTyCon,WFunTyCon)   (* FIXME!!! *)
| WCoAll     k f                     => (WFunTyCon,WFunTyCon)   (* FIXME!!! *)
| WCoSym     c                       => let (t2,t1) := weakCoercionTypes c in (t1,t2)
| WCoComp    c1 c2                   => (WFunTyCon,WFunTyCon)   (* FIXME!!! *)
| WCoLeft    c                       => (WFunTyCon,WFunTyCon)   (* FIXME!!! *)
| WCoRight   c                       => (WFunTyCon,WFunTyCon)   (* FIXME!!! *)
| WCoUnsafe  t1 t2                   => (t1,t2)
end.


(* this is a trick to allow circular definitions, post-extraction *)
Variable weakTypeToString : WeakType -> string.
```

```
    Extract Inlined Constant weakTypeToString => "(coreTypeToString . weakTypeToCoreType)".
Instance WeakTypeToString : ToString WeakType := { toString := weakTypeToString }.

Variable tyConToCoreTyCon : TyCon  -> CoreTyCon.          Extract Inlined Constant tyConToCoreTyCon  => "(\x -> x)".
Variable tyFunToCoreTyCon : TyFun  -> CoreTyCon.          Extract Inlined Constant tyFunToCoreTyCon  => "(\x -> x)".
Coercion tyConToCoreTyCon : TyCon >-> CoreTyCon.
Coercion tyFunToCoreTyCon : TyFun >-> CoreTyCon.
```