```
(*****************************************************************************************************************************)
(* HaskProof:                                                                                                                *)
(*                                                                                                                           *)
(*    Natural Deduction proofs of the well-typedness of a Haskell term.  Proofs use explicit structural rules (Gentzen-style)    *)
(*    and are in System FC extended with modal types indexed by Taha-Nielsen environment classifiers (λ^α)                    *)
(*                                                                                                                           *)
(*****************************************************************************************************************************)

Generalizable All Variables.
Require Import Preamble.
Require Import General.
Require Import NaturalDeduction.
Require Import Coq.Strings.String.
Require Import Coq.Lists.List.
Require Import HaskKinds.
Require Import HaskCoreTypes.
Require Import HaskLiteralsAndTyCons.
Require Import HaskStrongTypes.
Require Import HaskWeakVars.

(* A judgment consists of an environment shape (Γ and Δ) and a pair of trees of leveled types (the antecedent and succedent) valid
 * in any context of that shape.  Notice that the succedent contains a tree of types rather than a single type; think
 * of [ T1 |- T2 ] as asserting that a letrec with branches having types corresponding to the leaves of T2 is well-typed
 * in environment T1.  This subtle distinction starts to matter when we get into substructural (linear, affine, ordered, etc)
 * types *)
Inductive Judg  :=
  mkJudg :
  forall Γ:TypeEnv,
  forall Δ:CoercionEnv Γ,
  Tree ??(LeveledHaskType Γ ★) ->
  Tree ??(LeveledHaskType Γ ★) ->
  Judg.
  Notation "Γ > Δ > a '|-' s" := (mkJudg Γ Δ a s) (at level 52, Δ at level 50, a at level 52, s at level 50).

(* information needed to define a case branch in a HaskProof *)
Record ProofCaseBranch {tc:TyCon}{Γ}{Δ}{lev}{branchtype : HaskType Γ ★}{avars}{sac:@StrongAltCon tc} :=
{ pcb_freevars       :  Tree ??(LeveledHaskType Γ ★)
; pcb_judg           := sac_Γ sac Γ > sac_Δ sac Γ avars (map weakCK' Δ)
                > (mapOptionTree weakLT' pcb_freevars),,(unleaves (map (fun t => t@@weakL' lev)
                  (vec2list (sac_types sac Γ avars))))
                |- [weakLT' (branchtype @@ lev)]
```

```
}.
Implicit Arguments ProofCaseBranch [ ].

(* Figure 3, production $\vdash_E$, Uniform rules *)
Inductive Arrange {T} : Tree ??T -> Tree ??T -> Type :=
| RCanL   : forall a        ,                Arrange  (    [],,a   )      (       a   )
| RCanR   : forall a        ,                Arrange  (    a,,[]   )      (       a   )
| RuCanL  : forall a        ,                Arrange  (       a    )      (  [],,a    )
| RuCanR  : forall a        ,                Arrange  (       a    )      (  a,,[]    )
| RAssoc  : forall a b c    ,                Arrange  (a,,(b,,c)   )      ((a,,b),,c  )
| RCossa  : forall a b c    ,                Arrange  ((a,,b),,c   )      ( a,,(b,,c) )
| RExch   : forall a b      ,                Arrange  (   (b,,a)   )      (  (a,,b)   )
| RWeak   : forall a        ,                Arrange  (       []   )      (       a   )
| RCont   : forall a        ,                Arrange  (  (a,,a)    )      (       a   )
| RLeft   : forall {h}{c} x , Arrange h c -> Arrange  (    x,,h    )      (       x,,c)
| RRight  : forall {h}{c} x , Arrange h c -> Arrange  (    h,,x    )      (       c,,x)
| RComp   : forall {a}{b}{c}, Arrange a b -> Arrange b c -> Arrange a c
.

(* Figure 3, production $\vdash_E$, all rules *)
Inductive Rule : Tree ??Judg -> Tree ??Judg -> Type :=

| RArrange  : ∀ Γ Δ Σ₁ Σ₂ Σ,         Arrange Σ₁ Σ₂ -> Rule [Γ > Δ > Σ₁     |- Σ             ]   [Γ > Δ > Σ₂    |- Σ             ]

(* λ^α rules *)
| RBrak : ∀ Γ Δ t v Σ l,                          Rule [Γ > Δ > Σ     |- [t  @@ (v::l) ]]   [Γ > Δ > Σ    |- [<[v|-t]>   @@l]]
| REsc  : ∀ Γ Δ t v Σ l,                          Rule [Γ > Δ > Σ     |- [<[v|-t]> @@ l]]   [Γ > Δ > Σ    |- [t    @@ (v::l)]]

(* Part of GHC, but not explicitly in System FC *)
| RNote   : ∀ Γ Δ Σ τ l,          Note ->         Rule [Γ > Δ > Σ     |- [τ       @@ l]]   [Γ > Δ > Σ    |- [τ          @@l]]
| RLit    : ∀ Γ Δ v       l,                    Rule [                              ]   [Γ > Δ > []|- [literalType v  @@l]]

(* SystemFC rules *)
| RVar    : ∀ Γ Δ σ       l,                    Rule [                              ]   [Γ>Δ> [σ@@l]   |- [σ          @@l]]
| RGlobal : ∀ Γ Δ τ       l,   WeakExprVar ->   Rule [                              ]   [Γ>Δ>     []   |- [τ          @@l]]
| RLam    : forall Γ Δ Σ (tx:HaskType Γ ★) te l,  Rule [Γ>Δ> Σ,,[tx@@l]|- [te@@l]       ]   [Γ>Δ>    Σ     |- [tx--->te   @@l]]
| RCast   : forall Γ Δ Σ (σ₁ σ₂:HaskType Γ ★) l,
                  HaskCoercion Γ Δ (σ₁ ~~~ σ₂) ->  Rule [Γ>Δ> Σ        |- [σ₁@@l]       ]   [Γ>Δ>    Σ     |- [σ₂         @@l]]

| RJoin  : ∀ Γ Δ Σ₁ Σ₂ τ₁ τ₂ ,   Rule ([Γ > Δ > Σ₁ |- τ₁ ],,[Γ > Δ > Σ₂ |- τ₂ ])        [Γ>Δ>  Σ₁,,Σ₂  |- τ₁,,τ₂          ]
```

```
| RApp          : ∀ Γ Δ Σ₁ Σ₂ tx te l,  Rule ([Γ>Δ> Σ₁       |- [tx--->te @@l]],,[Γ>Δ> Σ₂ |- [tx@@l]])  [Γ>Δ> Σ₁,,Σ₂ |- [te   @@l]]

| RLet          : ∀ Γ Δ Σ₁ Σ₂ σ₁ σ₂ l,  Rule ([Γ>Δ> Σ₂ |- [σ₂@@l]],,[Γ>Δ> Σ₁,,[σ₂@@l] |- [σ₁@@l] ])     [Γ>Δ> Σ₁,,Σ₂ |- [σ₁   @@l]]

| RVoid    : ∀ Γ Δ ,              Rule [] [Γ > Δ > [] |- [] ]

| RAppT   : forall Γ Δ Σ κ σ (τ:HaskType Γ κ) l,      Rule [Γ>Δ> Σ   |- [HaskTAll κ σ @@l]]      [Γ>Δ>    Σ     |- [substT σ τ @@l]]
| RAbsT   : ∀ Γ Δ Σ κ σ   l,
  Rule [(κ::Γ)> (weakCE Δ)    >   mapOptionTree weakLT Σ |- [ HaskTApp (weakF σ) (FreshHaskTyVar _) @@ (weakL l)]]
       [Γ>Δ           >    Σ     |- [HaskTAll κ σ   @@ l]]

| RAppCo  : forall Γ Δ Σ κ (σ₁ σ₂:HaskType Γ κ) (γ:HaskCoercion Γ Δ (σ₁∼∼σ₂)) σ l,
    Rule [Γ>Δ> Σ |- [σ₁∼∼σ₂ ⇒σ@@l]] [Γ>Δ>    Σ     |- [σ       @@l]]
| RAbsCo  : forall Γ Δ Σ κ (σ₁ σ₂:HaskType Γ κ) σ l,
   Rule [Γ > ((σ₁∼∼σ₂)::Δ)           > Σ |- [σ @@ l]]
        [Γ > Δ >                       Σ |- [σ₁∼∼σ₂⇒σ @@l]]

| RLetRec       : forall Γ Δ Σ₁ τ₁ τ₂ lev, Rule [Γ > Δ > Σ₁,,(τ₂@@@lev) |- ([τ₁],,τ₂)@@@lev ] [Γ > Δ > Σ₁ |- [τ₁@@lev] ]
| RCase         : forall Γ Δ lev tc Σ avars tbranches
  (alts:Tree ??{ sac : @StrongAltCon tc & @ProofCaseBranch tc Γ Δ lev tbranches avars sac }),
                Rule
                    ((mapOptionTree (fun x => pcb_judg (projT2 x)) alts),,
                     [Γ > Δ >                                              Σ |- [ caseType tc avars @@ lev ] ])
                     [Γ > Δ > (mapOptionTreeAndFlatten (fun x => pcb_freevars (projT2 x)) alts),,Σ |- [ tbranches        @@ lev ] ]
.


(* A rule is considered "flat" if it is neither RBrak nor REsc *)
(* TODO: change this to (if RBrak/REsc -> False) *)
Inductive Rule_Flat : forall {h}{c}, Rule h c -> Prop :=
| Flat_RArrange        : ∀ Γ Δ  h c r         a ,  Rule_Flat (RArrange Γ Δ h c r a)
| Flat_RNote           : ∀ Γ Δ Σ τ l n           ,  Rule_Flat (RNote Γ Δ Σ τ l n)
| Flat_RLit            : ∀ Γ Δ Σ τ               ,  Rule_Flat (RLit Γ Δ Σ τ  )
| Flat_RVar            : ∀ Γ Δ  σ              l,  Rule_Flat (RVar Γ Δ  σ        l)
| Flat_RLam            : ∀ Γ Δ  Σ tx te    q     ,  Rule_Flat (RLam Γ Δ  Σ tx te     q )
| Flat_RCast           : ∀ Γ Δ  Σ σ τ γ    q     ,  Rule_Flat (RCast Γ Δ  Σ σ τ γ    q )
| Flat_RAbsT           : ∀ Γ   Σ κ σ a     q     ,  Rule_Flat (RAbsT Γ   Σ κ σ a     q )
| Flat_RAppT           : ∀ Γ Δ  Σ κ σ τ    q     ,  Rule_Flat (RAppT Γ Δ  Σ κ σ τ    q )
| Flat_RAppCo          : ∀ Γ Δ  Σ σ₁ σ₂ σ γ q l,  Rule_Flat (RAppCo Γ Δ  Σ  σ₁ σ₂ σ γ  q l)
| Flat_RAbsCo          : ∀ Γ   Σ κ σ  σ₁ σ₂ q1 q2   , Rule_Flat (RAbsCo Γ   Σ κ σ  σ₁ σ₂  q1 q2   )
| Flat_RApp            : ∀ Γ Δ  Σ tx te   p     l,  Rule_Flat (RApp Γ Δ  Σ tx te   p l)
```

```
| Flat_RLet            : ∀ Γ Δ  Σ σ₁  σ₂   p     l,  Rule_Flat (RLet Γ Δ  Σ σ₁  σ₂   p l)
| Flat_RJoin    : ∀ q a b c d e ,  Rule_Flat (RJoin q a b c d e)
| Flat_RVoid      : ∀ q a                  ,  Rule_Flat (RVoid q a)
| Flat_RCase           : ∀ Σ Γ  T κlen κ θ l x  , Rule_Flat (RCase Σ Γ T κlen κ θ l x)
| Flat_RLetRec         : ∀ Γ Δ Σ₁ τ₁ τ₂ lev,      Rule_Flat (RLetRec Γ Δ Σ₁ τ₁ τ₂ lev).

Lemma no_rules_with_empty_conclusion : forall c h, @Rule c h -> h=[] -> False.
  intros.
  destruct X; try destruct c; try destruct o; simpl in *; try inversion H.
  Qed.

Lemma no_rules_with_multiple_conclusions : forall c h,
  Rule c h -> { h1:Tree ??Judg & { h2:Tree ??Judg & h=(h1,,h2) }} -> False.
  intros.
  destruct X; try destruct c; try destruct o; simpl in *; try inversion H;
    try apply no_urules_with_empty_conclusion in u; try apply u.
    destruct X0; destruct s; inversion e.
    auto.
    destruct X0; destruct s; inversion e.
    destruct X0; destruct s; inversion e.
    destruct X0; destruct s; inversion e.
    destruct X0; destruct s; inversion e.
    destruct X0; destruct s; inversion e.
    destruct X0; destruct s; inversion e.
    destruct X0; destruct s; inversion e.
    destruct X0; destruct s; inversion e.
    destruct X0; destruct s; inversion e.
    destruct X0; destruct s; inversion e.
    destruct X0; destruct s; inversion e.
    destruct X0; destruct s; inversion e.
    destruct X0; destruct s; inversion e.
    destruct X0; destruct s; inversion e.
    destruct X0; destruct s; inversion e.
    destruct X0; destruct s; inversion e.
    destruct X0; destruct s; inversion e.
    destruct X0; destruct s; inversion e.
    Qed.

Lemma systemfc_all_rules_one_conclusion : forall h c1 c2 (r:Rule h (c1,,c2)), False.
  intros.
  eapply no_rules_with_multiple_conclusions.
```

```
apply r.
exists c1.
exists c2.
auto.
Qed.
```