```
(*****************************************************************************************************************************)
(* HaskProofFlattener:                                                                                                       *)
(*                                                                                                                           *)
(*    The Flattening Functor.                                                                                                *)
(*                                                                                                                           *)
(*****************************************************************************************************************************)

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 HaskProof.
Require Import NaturalDeduction.
Require Import NaturalDeductionCategory.

Require Import Algebras_ch4.
Require Import Categories_ch1_3.
Require Import Functors_ch1_4.
Require Import Isomorphisms_ch1_5.
Require Import ProductCategories_ch1_6_1.
Require Import OppositeCategories_ch1_6_2.
Require Import Enrichment_ch2_8.
Require Import Subcategories_ch7_1.
Require Import NaturalTransformations_ch7_4.
Require Import NaturalIsomorphisms_ch7_5.
Require Import BinoidalCategories.
Require Import PreMonoidalCategories.
Require Import MonoidalCategories_ch7_8.
Require Import Coherence_ch7_8.

Require Import HaskStrongTypes.
Require Import HaskStrong.
Require Import HaskProof.
Require Import HaskStrongToProof.
```

```
Require Import HaskProofToStrong.
Require Import ProgrammingLanguage.
Require Import HaskProofStratified.

Open Scope nd_scope.


(*
 *  The flattening transformation.  Currently only TWO-level languages are
 *  supported, and the level-1 sublanguage is rather limited.
*
 *  This file abuses terminology pretty badly.  For purposes of this file,
 *  "PCF" means "the level-1 sublanguage" and "FC" (aka System FC) means
 *  the whole language (level-0 language including bracketed level-1 terms)
 *)
Section HaskProofFlattener.


(*
  Definition code2garrow0 {Γ}(ec t1 t2:RawHaskType Γ ★) : RawHaskType Γ ★.
    admit.
    Defined.
  Definition code2garrow Γ (ec t:RawHaskType Γ ★) :=
      match t with
(*        | TApp ★★(TApp _ ★TArrow tx) t' => code2garrow0 ec tx       t'*)
        |                                _  => code2garrow0 ec unitType t
      end.
  Opaque code2garrow.
  Fixpoint typeMap {TV}{κ}(ty:@RawHaskType TV κ) : @RawHaskType TV κ :=
      match ty as TY in RawHaskType _ K return RawHaskType TV K with
        | TCode ec t        => code2garrow _ ec t
        | TApp _ _ t1 t2    => TApp (typeMap t1) (typeMap t2)
        | TAll _ f          => TAll _ (fun tv => typeMap (f tv))
        | TCoerc _ t1 t2 t3 => TCoerc (typeMap t1) (typeMap t2) (typeMap t3)
        | TVar   _ v        => TVar v
        | TArrow            => TArrow
        | TCon  tc          => TCon tc
        | TyFunApp  tf rhtl => (* FIXME *) TyFunApp tf rhtl
      end.
*)
```

```
(*
  Definition code2garrow Γ (ec t:RawHaskType Γ ★⇒★) :=
      match t with
(*        | TApp ★★(TApp _ ★TArrow tx) t' => code2garrow0 ec tx       t'*)
        |                                _  => code2garrow0 ec unitType t
      end.
  Opaque code2garrow.
  Fixpoint typeMap {TV}{κ}(ty:@RawHaskType TV κ) : @RawHaskType TV κ :=
      match ty as TY in RawHaskType _ K return RawHaskType TV K with
        | TCode ec t        => code2garrow _ ec t
        | TApp _ _ t1 t2    => TApp (typeMap t1) (typeMap t2)
        | TAll _ f          => TAll _ (fun tv => typeMap (f tv))
        | TCoerc _ t1 t2 t3 => TCoerc (typeMap t1) (typeMap t2) (typeMap t3)
        | TVar   _ v        => TVar v
        | TArrow            => TArrow
        | TCon  tc          => TCon tc
        | TyFunApp  tf rhtl => (* FIXME *) TyFunApp tf rhtl
      end.

  Definition typeMapL {Γ}(lht:LeveledHaskType Γ ★⇒★ : LeveledHaskType Γ ★ :=
    match lht with
(*      | t @@ nil       => (fun TV ite => typeMap (t TV ite)) @@ lev*)
      | t @@ lev => (fun TV ite => typeMap (t TV ite)) @@ lev
    end.
*)

  (* gathers a tree of guest-language types into a single host-language types via the tensor *)
  Definition tensorizeType {Γ} (lt:Tree ??(HaskType Γ ★⇒★) : HaskType Γ ★.
    admit.
    Defined.

  Definition mkGA {Γ} : HaskType Γ ★-> HaskType Γ ★-> HaskType Γ ★.
    admit.
    Defined.

  Definition guestJudgmentAsGArrowType {Γ}{Δ}(lt:PCFJudg Γ Δ) : HaskType Γ ★:=
    match lt with
      (x,y) => (mkGA (tensorizeType x) (tensorizeType y))
    end.
```

```
Definition obact {Γ}{Δ} (X:Tree ??(PCFJudg Γ Δ)) : Tree ??(LeveledHaskType Γ ★) :=
  mapOptionTree guestJudgmentAsGArrowType X @@@ nil.

Hint Constructors Rule_Flat.
Context {ndr:@ND_Relation _ Rule}.

(*
 * Here it is, what you've all been waiting for!  When reading this,
 * it might help to have the definition for "Inductive ND" (see
 * NaturalDeduction.v) handy as a cross-reference.
 *)
Hint Constructors Rule_Flat.
Definition FlatteningFunctor_fmor {Γ}{Δ}{ec}
  : forall h c,
    (h~~{JudgmentsL (PCF Γ Δ ec)}~~>c) ->
    ((obact(Δ:=ec) h)~~{TypesL (SystemFCa Γ Δ)}~~>(obact(Δ:=ec) c)).

  set (@nil (HaskTyVar Γ ★)) as lev.

  unfold hom; unfold ob; unfold ehom; simpl; unfold pmon_I; unfold obact; intros.

  induction X; simpl.

  (* the proof from no hypotheses of no conclusions (nd_id0) becomes RVoid *)
  apply nd_rule; apply (org_fc _ _ [] [(_,_)] (RVoid _ _)). apply Flat_RVoid.

  (* the proof from hypothesis X of conclusion X (nd_id1) becomes RVar *)
  apply nd_rule; apply (org_fc _ _ [] [(_,_)] (RVar _ _ _ _)). apply Flat_RVar.

  (* the proof from hypothesis X of no conclusions (nd_weak) becomes RWeak;;RVoid *)
  eapply nd_comp;
    [ idtac
    | eapply nd_rule
    ; eapply (org_fc  _ _ [(_,_)] [(_,_)] (RArrange _ _ _ _ _ (RWeak _)))
    ; auto ].
    eapply nd_rule.
    eapply (org_fc _ _ [] [(_,_)] (RVoid _ _)); auto. apply Flat_RVoid.
    apply Flat_RArrange.

  (* the proof from hypothesis X of two identical conclusions X,,X (nd_copy) becomes RVar;;RJoin;;RCont *)
  eapply nd_comp; [ idtac | eapply nd_rule; eapply (org_fc _ _ [(_,_)] [(_,_)] (RArrange _ _ _ _ _ (RCont _))) ].
```

```
  eapply nd_comp; [ apply nd_llecnac | idtac ].
  set (snd_initial(SequentND:=pl_snd(ProgrammingLanguage:=SystemFCa Γ Δ))
    (mapOptionTree (guestJudgmentAsGArrowType(Δ:=ec)) h @@@ lev)) as q.
  eapply nd_comp.
  eapply nd_prod.
  apply q.
  apply q.
  apply nd_rule.
  eapply (org_fc _ _ ([(_,_)],,[(_,_)]) [(_,_)] (RJoin _ _ _ _ _ _ )).
  destruct h; simpl.
  destruct o.
  simpl.
  apply Flat_RJoin.
  apply Flat_RJoin.
  apply Flat_RJoin.
  apply Flat_RArrange.

(* nd_prod becomes nd_llecnac;;nd_prod;;RJoin *)
eapply nd_comp.
  apply (nd_llecnac ;; nd_prod IHX1 IHX2).
  apply nd_rule.
  eapply (org_fc _ _ ([(_,_)],,[(_,_)]) [(_,_)] (RJoin _ _ _ _ _ _ )).
  apply (Flat_RJoin Γ Δ (mapOptionTree guestJudgmentAsGArrowType h1 @@@ nil)
   (mapOptionTree guestJudgmentAsGArrowType h2 @@@ nil)
   (mapOptionTree guestJudgmentAsGArrowType c1 @@@ nil)
   (mapOptionTree guestJudgmentAsGArrowType c2 @@@ nil)).

(* nd_comp becomes pl_subst (aka nd_cut) *)
eapply nd_comp.
  apply (nd_llecnac ;; nd_prod IHX1 IHX2).
  clear IHX1 IHX2 X1 X2.
  (*
  apply (@snd_cut _ _ _ _ _ _ (@pl_cnd _ _ _ _ (SystemFCa Γ Δ))).
  *)
  admit.

(* nd_cancell becomes RVar;;RuCanL *)
eapply nd_comp;
  [ idtac | eapply nd_rule; apply (org_fc _ _ [(_,_)] [(_,_)] (RArrange _ _ _ _ _ (RuCanL _))) ].
  apply (snd_initial(SequentND:=pl_cnd(ProgrammingLanguage:=(SystemFCa Γ Δ)))).
  apply Flat_RArrange.
```

```
(* nd_cancelr becomes RVar;;RuCanR *)
eapply nd_comp;
  [ idtac | eapply nd_rule; apply (org_fc _ _ [(_,_)] [(_,_)] (RArrange _ _ _ _ _ (RuCanR _))) ].
  apply (snd_initial(SequentND:=pl_cnd(ProgrammingLanguage:=(SystemFCa Γ Δ)))).
  apply Flat_RArrange.

(* nd_llecnac becomes RVar;;RCanL *)
eapply nd_comp;
  [ idtac | eapply nd_rule; apply (org_fc _ _ [(_,_)] [(_,_)] (RArrange _ _ _ _ _ (RCanL _))) ].
  apply (snd_initial(SequentND:=pl_cnd(ProgrammingLanguage:=(SystemFCa Γ Δ)))).
  apply Flat_RArrange.

(* nd_rlecnac becomes RVar;;RCanR *)
eapply nd_comp;
  [ idtac | eapply nd_rule; apply (org_fc _ _ [(_,_)] [(_,_)] (RArrange _ _ _ _ _ (RCanR _))) ].
  apply (snd_initial(SequentND:=pl_cnd(ProgrammingLanguage:=(SystemFCa Γ Δ)))).
  apply Flat_RArrange.

(* nd_assoc becomes RVar;;RAssoc *)
eapply nd_comp;
  [ idtac | eapply nd_rule; apply (org_fc _ _ [(_,_)] [(_,_)] (RArrange _ _ _ _ _ (RAssoc _ _ _))) ].
  apply (snd_initial(SequentND:=pl_cnd(ProgrammingLanguage:=(SystemFCa Γ Δ)))).
  apply Flat_RArrange.

(* nd_cossa becomes RVar;;RCossa *)
eapply nd_comp;
  [ idtac | eapply nd_rule; apply (org_fc _ _ [(_,_)] [(_,_)] (RArrange _ _ _ _ _ (RCossa _ _ _))) ].
  apply (snd_initial(SequentND:=pl_cnd(ProgrammingLanguage:=(SystemFCa Γ Δ)))).
  apply Flat_RArrange.

  destruct r as [r rp].
  refine (match rp as R in @Rule_PCF _ _ _ H C _ with
            | PCF_RArrange         h c r q          => let case_RURule        := tt in _
            | PCF_RLit             lit              => let case_RLit          := tt in _
            | PCF_RNote            Σ τ   n          => let case_RNote         := tt in _
            | PCF_RVar             σ                => let case_RVar          := tt in _
            | PCF_RLam             Σ tx te          => let case_RLam          := tt in _
            | PCF_RApp             Σ tx te   p      => let case_RApp          := tt in _
            | PCF_RLet             Σ σ₁  σ₂   p     => let case_RLet          := tt in _
            | PCF_RJoin    b c d e          => let case_RJoin := tt in _
```

```
      | PCF_RVoid                      => let case_RVoid   := tt in _
    (*| PCF_RCase           T κlen κ θ l x   => let case_RCase         := tt in _*)
    (*| PCF_RLetRec         Σ₁ τ₁ τ₂ lev     => let case_RLetRec       := tt in _*)
    end); simpl in *.
clear rp.
clear r h c.
rename r0 into r; rename h0 into h; rename c0 into c.

destruct case_RURule.
  refine (match q with
    | RLeft   a b c r => let case_RLeft  := tt in _
    | RRight  a b c r => let case_RRight := tt in _
    | RCanL     b     => let case_RCanL  := tt in _
    | RCanR     b     => let case_RCanR  := tt in _
    | RuCanL    b     => let case_RuCanL := tt in _
    | RuCanR    b     => let case_RuCanR := tt in _
    | RAssoc    b c d => let case_RAssoc := tt in _
    | RCossa    b c d => let case_RCossa := tt in _
    | RExch     b c   => let case_RExch  := tt in _
    | RWeak     b     => let case_RWeak  := tt in _
    | RCont     b     => let case_RCont  := tt in _
    | RComp a b c f g => let case_RComp  := tt in _
  end).

destruct case_RCanL.
  (* ga_cancell *)
  admit.

destruct case_RCanR.
  (* ga_cancelr *)
  admit.

destruct case_RuCanL.
  (* ga_uncancell *)
  admit.

destruct case_RuCanR.
  (* ga_uncancelr *)
  admit.

destruct case_RAssoc.
```

```
  (* ga_assoc *)
  admit.

destruct case_RCossa.
  (* ga_unassoc *)
  admit.

destruct case_RExch.
  (* ga_swap *)
  admit.

destruct case_RWeak.
  (* ga_drop *)
  admit.

destruct case_RCont.
  (* ga_copy *)
  admit.

destruct case_RLeft.
  (* ga_second *)
  admit.

destruct case_RRight.
  (* ga_first *)
  admit.

destruct case_RComp.
  (* ga_comp *)
  admit.

destruct case_RLit.
  (* ga_literal *)
  admit.

(* hey cool, I figured out how to pass CoreNote's through... *)
destruct case_RNote.
  eapply nd_comp.
  eapply nd_rule.
  eapply (org_fc _ _ [] [(_,_)] (RVar _ _ _ _)) . auto.
  apply Flat_RVar.
```

```
      apply nd_rule.
      apply (org_fc _ _ [(_,_)] [(_,_)] (RNote _ _ _ _ _ n)). auto.
      apply Flat_RNote.

    destruct case_RVar.
      (* ga_id *)
      admit.

    destruct case_RLam.
      (* ga_curry, but try to avoid this someday in the future if the argument type isn't a function *)
      admit.

    destruct case_RApp.
      (* ga_apply *)
      admit.

    destruct case_RLet.
      (* ga_comp! perhaps this means the ga_curry avoidance can be done by turning lambdas into lets? *)
      admit.

    destruct case_RVoid.
      (* ga_id u *)
      admit.

    destruct case_RJoin.
      (* ga_first+ga_second; technically this assumes a specific evaluation order, which is bad *)
      admit.

    Defined.

Instance FlatteningFunctor {Γ}{Δ}{ec} : Functor (JudgmentsL (PCF Γ Δ ec)) (TypesL (SystemFCa Γ Δ)) (obact) :=
  { fmor := FlatteningFunctor_fmor }.
  admit.
  admit.
  admit.
  Defined.

(*
Definition ReificationFunctor Γ Δ : Functor (JudgmentsL _ _ (PCF n Γ Δ)) SystemFCa' (mapOptionTree brakifyJudg).
  refine {| fmor := ReificationFunctor_fmor Γ Δ |}; unfold hom; unfold ob; simpl ; intros.
  unfold ReificationFunctor_fmor; simpl.
```

```
    admit.
    unfold ReificationFunctor_fmor; simpl.
    admit.
    unfold ReificationFunctor_fmor; simpl.
    admit.
    Defined.

  Definition PCF_SMME (n:nat)(Γ:TypeEnv)(Δ:CoercionEnv Γ) : ProgrammingLanguageSMME.
    refine {| plsmme_pl := PCF n Γ Δ |}.
    admit.
    Defined.

  Definition SystemFCa_SMME (n:nat)(Γ:TypeEnv)(Δ:CoercionEnv Γ) : ProgrammingLanguageSMME.
    refine {| plsmme_pl := SystemFCa n Γ Δ |}.
    admit.
    Defined.

  Definition ReificationFunctorMonoidal n : MonoidalFunctor (JudgmentsN n) (JudgmentsN (S n)) (ReificationFunctor n).
    admit.
    Defined.

  (* 5.1.4 *)
  Definition PCF_SystemFCa_two_level n Γ Δ : TwoLevelLanguage (PCF_SMME n Γ Δ) (SystemFCa_SMME (S n) Γ Δ).
    admit.
    (*  ... and the retraction exists *)
    Defined.
  *)
  (* Any particular proof in HaskProof is only finitely large, so it uses only finitely many levels of nesting, so
   * it falls within (SystemFCa n) for some n.  This function calculates that "n" and performs the translation *)
  (*
  Definition HaskProof_to_SystemFCa :
    forall h c (pf:ND Rule h c),
      { n:nat & h ~~{JudgmentsL (SystemFCa_SMME n)}~~> c }.
      *)
  (* for every n we have a functor from the category of (n+1)-bounded proofs to the category of n-bounded proofs *)

End HaskProofFlattener.
```