```
(*****************************************************************************************************************************)
(* HaskProofStratified:                                                                                                      *)
(*                                                                                                                           *)
(*    An alternate representation for HaskProof which ensures that deductions on a given level are grouped into contiguous    *)
(*    blocks.  This representation lacks the attractive compositionality properties of HaskProof, but makes it easier to      *)
(*    perform the flattening process.                                                                                        *)
(*                                                                                                                           *)
(*****************************************************************************************************************************)

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 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.

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 HaskProofStratified.

  Section PCF.

  Context (ndr_systemfc:@ND_Relation _ Rule).

  Context Γ (Δ:CoercionEnv Γ).
  Definition PCFJudg (ec:HaskTyVar Γ ★) :=
    @prod (Tree ??(HaskType Γ ★)) (Tree ??(HaskType Γ ★)).
  Definition pcfjudg (ec:HaskTyVar Γ ★) :=
    @pair (Tree ??(HaskType Γ ★)) (Tree ??(HaskType Γ ★)).

  (* given an PCFJudg at depth (ec::depth) we can turn it into an PCFJudg
   * from depth (depth) by wrapping brackets around everything in the
   * succedent and repopulating *)
  Definition brakify {ec} (j:PCFJudg ec) : Judg :=
    match j with
      (Σ,τ) => Γ > Δ > (Σ@@@(ec::nil)) |- (mapOptionTree (fun t => HaskBrak ec t) τ @@@ nil)
      end.

  Definition pcf_vars {Γ}(ec:HaskTyVar Γ ★)(t:Tree ??(LeveledHaskType Γ ★)) : Tree ??(HaskType Γ ★)
    := mapOptionTreeAndFlatten (fun lt =>
      match lt with t @@ l => match l with
                                | ec'::nil => if eqd_dec ec ec' then [t] else []
                                | _ => []
                              end
      end) t.
```

```
Inductive MatchingJudgments {ec} : Tree ??(PCFJudg ec) -> Tree ??Judg -> Type :=
  | match_nil    : MatchingJudgments [] []
  | match_branch : forall a b c d, MatchingJudgments a b -> MatchingJudgments c d -> MatchingJudgments (a,,c) (b,,d)
  | match_leaf   :
    forall Σ τ lev,
      MatchingJudgments
        [((pcf_vars ec Σ)         ,                                τ        )]
        [Γ > Δ >              Σ  |- (mapOptionTree (HaskBrak ec) τ @@@ lev)].

Definition fc_vars {Γ}(ec:HaskTyVar Γ ★)(t:Tree ??(LeveledHaskType Γ ★)) : Tree ??(HaskType Γ ★)
  := mapOptionTreeAndFlatten (fun lt =>
    match lt with t @@ l => match l with
                              | ec'::nil => if eqd_dec ec ec' then [] else [t]
                              | _ => []
                            end
    end) t.

Definition pcfjudg2judg ec (cj:PCFJudg ec) :=
  match cj with (Σ,τ) => Γ > Δ > (Σ @@@ (ec::nil)) |- (τ @@@ (ec::nil)) end.

(* Rules allowed in PCF; i.e. rules we know how to turn into GArrows     *)
(* Rule_PCF consists of the rules allowed in flat PCF: everything except *)
(* AppT, AbsT, AppC, AbsC, Cast, Global, and some Case statements        *)
Inductive Rule_PCF (ec:HaskTyVar Γ ★)
  : forall (h c:Tree ??(PCFJudg ec)), Rule (mapOptionTree (pcfjudg2judg ec) h) (mapOptionTree (pcfjudg2judg ec) c) -> Type :=
| PCF_RArrange    : ∀ x y t     a,  Rule_PCF ec [(_, _)] [(_, _)] (RArrange Γ Δ (x@@@(ec::nil)) (y@@@(ec::nil)) (t@@@(ec::nil)) a)
| PCF_RLit        : ∀ lit        ,  Rule_PCF ec [           ] [ ([],[_]) ] (RLit   Γ Δ  lit (ec::nil))
| PCF_RNote       : ∀ Σ τ   n    ,  Rule_PCF ec [(_,[_])] [(_,[_])] (RNote  Γ Δ  (Σ@@@(ec::nil)) τ         (ec::nil) n)
| PCF_RVar        : ∀ σ          ,  Rule_PCF ec [           ] [([_],[_])] (RVar   Γ Δ    σ         (ec::nil)  )
| PCF_RLam        : ∀ Σ tx te    ,  Rule_PCF ec [((_,,[_]),[_])] [(_,[_])] (RLam   Γ Δ  (Σ@@@(ec::nil)) tx te  (ec::nil)  )

| PCF_RApp             : ∀ Σ Σ' tx te ,
  Rule_PCF ec ([(_,[_])],,[(_,[_])]) [((_,,_),[_])]
  (RApp Γ Δ (Σ@@@(ec::nil))(Σ'@@@(ec::nil)) tx te (ec::nil))

| PCF_RLet             : ∀ Σ Σ' σ₂   p,
  Rule_PCF ec ([(_,[_])],,[((_,,[_]),[_])]) [((_,,_),[_])]
  (RLet Γ Δ (Σ@@@(ec::nil)) (Σ'@@@(ec::nil)) σ₂ p (ec::nil))

| PCF_RVoid      :                Rule_PCF ec [           ] [([],[])] (RVoid   Γ Δ  )
```

```
(*| PCF_RLetRec          : ∀ Σ₁ τ₁ τ₂    ,  Rule_PCF (ec::nil) _ _ (RLetRec Γ Δ Σ₁ τ₁ τ₂ (ec::nil) )*)
  | PCF_RJoin    : ∀ Σ₁ Σ₂ τ₁ τ₂ ,  Rule_PCF ec ([(_,_)],,[(_,_)]) [((_,,_),(_,,_))]
    (RJoin Γ Δ (Σ₁@@@(ec::nil)) (Σ₂@@@(ec::nil)) (τ₁@@@(ec::nil)) (τ₂@@@(ec::nil))).
  (* need int/boolean case *)
  Implicit Arguments Rule_PCF [ ].

  Definition PCFRule lev h c := { r:_ & @Rule_PCF lev h c r }.
  End PCF.

  Definition FCJudg Γ (Δ:CoercionEnv Γ) :=
    @prod (Tree ??(LeveledHaskType Γ ★)) (Tree ??(LeveledHaskType Γ ★)).
  Definition fcjudg2judg {Γ}{Δ}(fc:FCJudg Γ Δ) :=
    match fc with
      (x,y) => Γ > Δ > x |- y
        end.
  Coercion fcjudg2judg : FCJudg >-> Judg.

  Definition pcfjudg2fcjudg {Γ}{Δ} ec (fc:PCFJudg Γ ec) : FCJudg Γ Δ :=
    match fc with
      (x,y) => (x @@@ (ec::nil),y @@@ (ec::nil))
        end.

  (* An organized deduction has been reorganized into contiguous blocks whose
   * hypotheses (if any) and conclusion have the same Γ and Δ and a fixed nesting depth.  The boolean
   * indicates if non-PCF rules have been used *)
  Inductive OrgR Γ Δ : Tree ??(FCJudg Γ Δ) -> Tree ??(FCJudg Γ Δ) -> Type :=

  | org_fc        : forall (h c:Tree ??(FCJudg Γ Δ))
    (r:Rule (mapOptionTree fcjudg2judg h) (mapOptionTree fcjudg2judg c)),
    Rule_Flat r ->
    OrgR _ _ h c

  | org_pcf      : forall ec h c,
    ND (PCFRule Γ Δ ec)  h c  ->
    OrgR        Γ Δ     (mapOptionTree (pcfjudg2fcjudg ec) h)  (mapOptionTree (pcfjudg2fcjudg ec) c).

  Definition mkEsc Γ Δ ec (h:Tree ??(PCFJudg Γ ec))
    : ND Rule
    (mapOptionTree (brakify Γ Δ) h)
    (mapOptionTree (pcfjudg2judg Γ Δ ec) h).
    apply nd_replicate; intros.
```

```
  destruct o; simpl in *.
  induction t0.
  destruct a; simpl.
  apply nd_rule.
  apply REsc.
  apply nd_id.
  apply (Prelude_error "mkEsc got multi-leaf succedent").
  Defined.

Definition mkBrak Γ Δ ec (h:Tree ??(PCFJudg Γ ec))
  : ND Rule
  (mapOptionTree (pcfjudg2judg Γ Δ ec) h)
  (mapOptionTree (brakify Γ Δ) h).
  apply nd_replicate; intros.
  destruct o; simpl in *.
  induction t0.
  destruct a; simpl.
  apply nd_rule.
  apply RBrak.
  apply nd_id.
  apply (Prelude_error "mkBrak got multi-leaf succedent").
  Defined.

  (*
Definition Partition {Γ} ec (Σ:Tree ??(LeveledHaskType Γ ★)) :=
  { vars:(_ * _) |
    fc_vars  ec Σ = fst vars /\
    pcf_vars ec Σ = snd vars }.
    *)

Definition pcfToND Γ Δ : forall ec h c,
  ND (PCFRule Γ Δ ec) h c -> ND Rule (mapOptionTree (pcfjudg2judg Γ Δ ec) h) (mapOptionTree (pcfjudg2judg Γ Δ ec) c).
  intros.
  eapply (fun q => nd_map' _ q X).
  intros.
  destruct X0.
  apply nd_rule.
  apply x.
  Defined.

Instance OrgPCF Γ Δ lev : @ND_Relation _ (PCFRule Γ Δ lev) :=
```

```
  { ndr_eqv := fun a b f g => (pcfToND  _ _ _ _ _ f) === (pcfToND _ _ _ _ _ g) }.
  Admitted.

(*
 * An intermediate representation necessitated by Coq's termination
 * conditions.  This is basically a tree where each node is a
 * subproof which is either entirely level-1 or entirely level-0
 *)
Inductive Alternating : Tree ??Judg -> Type :=

  | alt_nil    : Alternating []

  | alt_branch : forall a b,
    Alternating a -> Alternating b -> Alternating (a,,b)

  | alt_fc     : forall h c,
    Alternating h ->
    ND Rule h c ->
    Alternating c

  | alt_pcf    : forall Γ Δ ec h c h' c',
    MatchingJudgments Γ Δ  h  h' ->
    MatchingJudgments Γ Δ  c  c' ->
    Alternating h' ->
    ND (PCFRule Γ Δ ec) h c ->
    Alternating c'.

Require Import Coq.Logic.Eqdep.

Lemma magic a b c d ec e :
  ClosedSIND(Rule:=Rule) [a > b > c |- [d @@  (ec :: e)]] ->
  ClosedSIND(Rule:=Rule) [a > b > pcf_vars ec c @@@ (ec :: nil) |- [d @@  (ec :: nil)]].
  admit.
  Defined.

Definition orgify : forall Γ Δ Σ τ (pf:ClosedSIND(Rule:=Rule) [Γ > Δ > Σ |- τ]), Alternating [Γ > Δ > Σ |- τ].

  refine (
    fix  orgify_fc' Γ Δ Σ τ (pf:ClosedSIND [Γ > Δ > Σ |- τ]) {struct pf} : Alternating [Γ > Δ > Σ |- τ] :=
      let case_main := tt in _
    with orgify_fc c (pf:ClosedSIND c) {struct pf} : Alternating c :=
```

```
  (match c as C return C=c -> Alternating C with
    | T_Leaf None                    => fun _ => alt_nil
    | T_Leaf (Some (Γ > Δ > Σ |- τ)) => let case_leaf := tt in fun eqpf => _
    | T_Branch b1 b2                 => let case_branch := tt in fun eqpf => _
  end (refl_equal _))
  with orgify_pcf   Γ Δ ec pcfj j (m:MatchingJudgments Γ Δ pcfj j)
    (pf:ClosedSIND (mapOptionTree (pcfjudg2judg Γ Δ ec) pcfj)) {struct pf} : Alternating j :=
    let case_pcf := tt in _
  for orgify_fc').

  destruct case_main.
  inversion pf; subst.
  set (alt_fc _ _ (orgify_fc _ X) (nd_rule X0)) as backup.
  refine (match X0 as R in Rule H C return
            match C with
              | T_Leaf (Some (Γ > Δ > Σ |- τ)) =>
                h=H -> Alternating [Γ > Δ > Σ |- τ] -> Alternating [Γ > Δ > Σ |- τ]
              | _                              => True
            end
             with
            | RBrak   Σ a b c n m          => let case_RBrak := tt         in fun pf' backup => _
            | REsc    Σ a b c n m          => let case_REsc := tt          in fun pf' backup => _
            | _ => fun pf' x => x
          end (refl_equal _) backup).
  clear backup0 backup.

  destruct case_RBrak.
    rename c into ec.
    set (@match_leaf Σ0 a ec n [b] m) as q.
    set (orgify_pcf Σ0 a ec _ _ q) as q'.
    apply q'.
    simpl.
    rewrite pf' in X.
    apply magic in X.
    apply X.

  destruct case_REsc.
    apply (Prelude_error "encountered Esc in wrong side of mkalt").

destruct case_leaf.
  apply orgify_fc'.
```

```
    rewrite eqpf.
    apply pf.

  destruct case_branch.
    rewrite <- eqpf in pf.
    inversion pf; subst.
    apply no_rules_with_multiple_conclusions in X0.
    inversion X0.
    exists b1. exists b2.
    auto.
    apply (alt_branch _ _ (orgify_fc _ X) (orgify_fc _ X0)).

  destruct case_pcf.
  Admitted.

Definition pcfify Γ Δ ec : forall Σ τ,
  ClosedSIND(Rule:=Rule) [ Γ > Δ > Σ@@@(ec::nil) |- τ @@@ (ec::nil)]
    -> ND (PCFRule Γ Δ ec) [] [(Σ,τ)].

  refine ((
    fix pcfify Σ τ (pn:@ClosedSIND _ Rule [ Γ > Δ > Σ@@@(ec::nil) |- τ @@@ (ec::nil)]) {struct pn}
    : ND (PCFRule Γ Δ ec) [] [(Σ,τ)] :=
   (match pn in @ClosedSIND _ _ J return J=[Γ > Δ > Σ@@@(ec::nil) |- τ @@@ (ec::nil)] -> _ with
    | cnd_weak             => let case_nil    := tt in _
    | cnd_rule h c cnd' r  => let case_rule   := tt in _
    | cnd_branch _ _ c1 c2 => let case_branch := tt in _
    end (refl_equal _)))).
    intros.
    inversion H.
    intros.
    destruct c; try destruct o; inversion H.
    destruct j.
    Admitted.

(* any proof in organized form can be "dis-organized" *)
(*
Definition unOrgR : forall Γ Δ h c, OrgR Γ Δ h c -> ND Rule h c.
  intros.
  induction X.
    apply nd_rule.
    apply r.
```

```
  eapply nd_comp.
    (*
    apply (mkEsc h).
    eapply nd_comp; [ idtac |  apply (mkBrak c) ].
    apply pcfToND.
    apply n.
    *)
    Admitted.
Definition unOrgND Γ Δ h c :  ND (OrgR Γ Δ) h c -> ND Rule h c := nd_map (unOrgR Γ Δ).
*)

Hint Constructors Rule_Flat.

Definition PCF_Arrange {Γ}{Δ}{lev} : forall x y z, Arrange x y -> ND (PCFRule Γ Δ lev) [(x,z)] [(y,z)].
  admit.
  Defined.

Definition PCF_cut Γ Δ lev : forall a b c, ND (PCFRule Γ Δ lev) ([(a,b)],,[(b,c)]) [(a,c)].
  intros.
  destruct b.
  destruct o.
  destruct c.
  destruct o.

  (* when the cut is a single leaf and the RHS is a single leaf: *)
  eapply nd_comp.
    eapply nd_prod.
    apply nd_id.
    apply (PCF_Arrange [h] ([],,[h]) [h0]).
    apply RuCanL.
    eapply nd_comp; [ idtac | apply (PCF_Arrange ([],,a) a [h0]); apply RCanL ].
    apply nd_rule.
    (*
    set (@RLet Γ Δ [] (a@@@(ec::nil)) h0 h (ec::nil)) as q.
    exists q.
    apply (PCF_RLet _ [] a h0 h).
  apply (Prelude_error "cut rule invoked with [a|=[b]] [[b]|=[]]").
  apply (Prelude_error "cut rule invoked with [a|=[b]] [[b]|=[x,,y]]").
  apply (Prelude_error "cut rule invoked with [a|=[]]  [[]|=c]").
  apply (Prelude_error "cut rule invoked with [a|=[b,,c]] [[b,,c]|=z]").
  *)
```

```
  Admitted.

Instance PCF_sequents Γ Δ lev ec : @SequentND _ (PCFRule Γ Δ lev) _ (pcfjudg Γ ec) :=
  { snd_cut := PCF_cut Γ Δ lev }.
  apply Build_SequentND.
  intros.
  induction a.
  destruct a; simpl.
  apply nd_rule.
    exists (RVar _ _ _ _).
    apply PCF_RVar.
  apply nd_rule.
    exists (RVoid _ _ ).
    apply PCF_RVoid.
  eapply nd_comp.
    eapply nd_comp; [ apply nd_llecnac | idtac ].
    apply (nd_prod IHa1 IHa2).
    apply nd_rule.
      exists (RJoin _ _ _ _ _ _).
      apply PCF_RJoin.
    admit.
      Defined.

Definition PCF_left Γ Δ lev a b c : ND (PCFRule Γ Δ lev) [(b,c)] [((a,,b),(a,,c))].
  eapply nd_comp; [ apply nd_llecnac | eapply nd_comp; [ idtac | idtac ] ].
  eapply nd_prod; [ apply snd_initial | apply nd_id ].
  apply nd_rule.
  set (@PCF_RJoin Γ Δ lev a b a c) as q'.
  refine (existT _ _ _).
  apply q'.
  Admitted.

Definition PCF_right Γ Δ lev a b c : ND (PCFRule Γ Δ lev) [(b,c)] [((b,,a),(c,,a))].
  eapply nd_comp; [ apply nd_rlecnac | eapply nd_comp; [ idtac | idtac ] ].
  eapply nd_prod; [ apply nd_id | apply snd_initial ].
  apply nd_rule.
  set (@PCF_RJoin Γ Δ lev b a c a) as q'.
  refine (existT _ _ _).
  apply q'.
  Admitted.
```

```
Instance PCF_sequent_join Γ Δ lev : @ContextND _ (PCFRule Γ Δ lev) _ (pcfjudg Γ lev) _ :=
{ cnd_expand_left  := fun a b c => PCF_left  Γ Δ lev c a b
; cnd_expand_right := fun a b c => PCF_right Γ Δ lev c a b }.

  intros; apply nd_rule. unfold PCFRule. simpl.
    exists (RArrange _ _ _ _ _ (RCossa _ _ _)).
    apply (PCF_RArrange _ _ lev ((a,,b),,c) (a,,(b,,c)) x).

  intros; apply nd_rule. unfold PCFRule. simpl.
    exists (RArrange _ _ _ _ _ (RAssoc _ _ _)).
    apply (PCF_RArrange _ _ lev (a,,(b,,c)) ((a,,b),,c) x).

  intros; apply nd_rule. unfold PCFRule. simpl.
    exists (RArrange _ _ _ _ _ (RCanL _)).
    apply (PCF_RArrange _ _ lev ([],,a) _ _).

  intros; apply nd_rule. unfold PCFRule. simpl.
    exists (RArrange _ _ _ _ _ (RCanR _)).
    apply (PCF_RArrange _ _ lev (a,,[]) _ _).

  intros; apply nd_rule. unfold PCFRule. simpl.
    exists (RArrange _ _ _ _ _ (RuCanL _)).
    apply (PCF_RArrange _ _ lev _ ([],,a) _).

  intros; apply nd_rule. unfold PCFRule. simpl.
    exists (RArrange _ _ _ _ _ (RuCanR _)).
    apply (PCF_RArrange _ _ lev _ (a,,[]) _).
    Defined.

Instance OrgPCF_SequentND_Relation Γ Δ lev : SequentND_Relation (PCF_sequent_join Γ Δ lev) (OrgPCF Γ Δ lev).
  admit.
  Defined.

Definition OrgPCF_ContextND_Relation Γ Δ lev
  : @ContextND_Relation _ _ _ _ _ (PCF_sequent_join Γ Δ lev) (OrgPCF Γ Δ lev) (OrgPCF_SequentND_Relation Γ Δ lev).
  admit.
  Defined.

(* 5.1.3 *)
Instance PCF Γ Δ lev : ProgrammingLanguage :=
{ pl_cnd     := PCF_sequent_join Γ Δ lev
```

```
; pl_eqv     := OrgPCF_ContextND_Relation Γ Δ lev
}.

Definition SystemFCa_cut Γ Δ : forall a b c, ND (OrgR Γ Δ) ([(a,b)],,[(b,c)]) [(a,c)].
  intros.
  destruct b.
  destruct o.
  destruct c.
  destruct o.

  (* when the cut is a single leaf and the RHS is a single leaf: *)
  (*
  eapply nd_comp.
    eapply nd_prod.
    apply nd_id.
    eapply nd_rule.
    set (@org_fc) as ofc.
    set (RArrange Γ Δ _ _ _ (RuCanL [l0])) as rule.
    apply org_fc with (r:=RArrange _ _ _ _ _ (RuCanL [_])).
    auto.
    eapply nd_comp; [ idtac | eapply nd_rule; apply org_fc with (r:=RArrange _ _ _ _ _ (RCanL _)) ].
    apply nd_rule.
    destruct l.
    destruct l0.
    assert (h0=h2). admit.
    subst.
    apply org_fc with (r:=@RLet Γ Δ [] a h1 h h2).
    auto.
    auto.
    *)
  admit.
  apply (Prelude_error "systemfc cut rule invoked with [a|=[b]] [[b]|=[]]").
  apply (Prelude_error "systemfc cut rule invoked with [a|=[b]] [[b]|=[x,,y]]").
  apply (Prelude_error "systemfc rule invoked with [a|=[]]  [[]|=c]").
  apply (Prelude_error "systemfc rule invoked with [a|=[b,,c]] [[b,,c]|=z]").
  Defined.

Instance SystemFCa_sequents Γ Δ : @SequentND _ (OrgR Γ Δ) _ _ :=
{ snd_cut := SystemFCa_cut Γ Δ }.
  apply Build_SequentND.
  intros.
```

```
  induction a.
  destruct a; simpl.
  (*
  apply nd_rule.
    destruct l.
    apply org_fc with (r:=RVar _ _ _ _).
    auto.
  apply nd_rule.
    apply org_fc with (r:=RVoid _ _ ).
    auto.
  eapply nd_comp.
    eapply nd_comp; [ apply nd_llecnac | idtac ].
    apply (nd_prod IHa1 IHa2).
    apply nd_rule.
      apply org_fc with (r:=RJoin _ _ _ _ _ _).
      auto.
    admit.
    *)
    admit.
    admit.
    admit.
    admit.
    Defined.

Definition SystemFCa_left Γ Δ a b c : ND (OrgR Γ Δ) [(b,c)] [((a,,b),(a,,c))].
  admit.
  (*
  eapply nd_comp; [ apply nd_llecnac | eapply nd_comp; [ idtac | idtac ] ].
  eapply nd_prod; [ apply snd_initial | apply nd_id ].
  apply nd_rule.
  apply org_fc with (r:=RJoin Γ Δ a b a c).
  auto.
  *)
  Defined.

Definition SystemFCa_right Γ Δ a b c : ND (OrgR Γ Δ) [(b,c)] [((b,,a),(c,,a))].
  admit.
  (*
  eapply nd_comp; [ apply nd_rlecnac | eapply nd_comp; [ idtac | idtac ] ].
  eapply nd_prod; [ apply nd_id | apply snd_initial ].
  apply nd_rule.
```

```
  apply org_fc with (r:=RJoin Γ Δ b a c a).
  auto.
  *)
  Defined.

Instance SystemFCa_sequent_join Γ Δ : @ContextND _ _ _ _ (SystemFCa_sequents Γ Δ) :=
{ cnd_expand_left  := fun a b c => SystemFCa_left Γ Δ c a b
; cnd_expand_right := fun a b c => SystemFCa_right Γ Δ c a b }.
  (*
  intros; apply nd_rule. simpl.
    apply (org_fc _ _ _ _ ((RArrange _ _ _ _ _ (RCossa _ _ _)))).
    auto.

  intros; apply nd_rule. simpl.
    apply (org_fc _ _ _ _ (RArrange _ _ _ _ _ (RAssoc _ _ _))); auto.

  intros; apply nd_rule. simpl.
    apply (org_fc _ _ _ _ (RArrange _ _ _ _ _ (RCanL _))); auto.

  intros; apply nd_rule. simpl.
    apply (org_fc _ _ _ _ (RArrange _ _ _ _ _ (RCanR _))); auto.

  intros; apply nd_rule. simpl.
    apply (org_fc _ _ _ _ (RArrange _ _ _ _ _ (RuCanL _))); auto.

  intros; apply nd_rule. simpl.
    apply (org_fc _ _ _ _ (RArrange _ _ _ _ _ (RuCanR _))); auto.
    *)
    admit.
    admit.
    admit.
    admit.
    admit.
    admit.
    admit.
    Defined.

Instance OrgFC Γ Δ : @ND_Relation _ (OrgR Γ Δ).
  Admitted.

Instance OrgFC_SequentND_Relation Γ Δ : SequentND_Relation (SystemFCa_sequent_join Γ Δ) (OrgFC Γ Δ).
  admit.
```

```
    Defined.

  Definition OrgFC_ContextND_Relation Γ Δ
    : @ContextND_Relation _ _ _ _ _ (SystemFCa_sequent_join Γ Δ) (OrgFC Γ Δ) (OrgFC_SequentND_Relation Γ Δ).
    admit.
    Defined.

  (* 5.1.2 *)
  Instance SystemFCa Γ Δ : @ProgrammingLanguage (LeveledHaskType Γ ★) _ :=
  { pl_eqv                := OrgFC_ContextND_Relation Γ Δ
  ; pl_snd                := SystemFCa_sequents Γ Δ
  }.

End HaskProofStratified.
```