Generalizable All Variables.
Require Import Preamble.
Require Import General.

(* Unlike most formalizations, this library offers two different ways
  to represent a natural deduction proof. To demonstrate this,
  consider the signature of the propositional calculus:

  *)

  Variable PropositionalVariable : Type.

  Inductive Formula : Prop :=
  | formula_var : PropositionalVariable -> Formula (* every propositional variable is a formula *)
  | formula_and : Formula -> Formula -> Formula (* the conjunction of any two formulae is a formula *)
  | formula_or : Formula -> Formula -> Formula (* the disjunction of any two formulae is a formula *)

  And couple this with the theory of conjunction and disjunction:
  * $\varphi \lor \psi$ is true if either $\varphi$ is true or $\psi$ is true, and $\varphi \land \psi$ is true
  * if both $\varphi$ and $\psi$ are true.

  1) Structurally implicit proofs

  This is what you would call the "usual" representation -- it’s
  * what most people learn when they first start programming in Coq:

  Inductive IsTrue : Formula -> Prop :=
  | IsTrue_or1 : forall f1 f2, IsTrue f1 -> IsTrue (formula_or f1 f2)
  | IsTrue_or2 : forall f1 f2, IsTrue f2 -> IsTrue (formula_or f1 f2)
  | IsTrue_and : forall f1 f2, IsTrue f1 f2 -> IsTrue (formula_and f1 f2)

  Here each judgment (such as "$\varphi$ is true") is represented by a Coq
  * type; furthermore:
1. A proof of a judgment is any inhabitant of that Coq type.

2. A proof of a judgment "J2" from hypothesis judgment "J1" is any Coq function from the Coq type for J1 to the Coq type for J2; Composition of (hypothetical) proofs is represented by composition of Coq functions.

3. A pair of judgments is represented by their product (Coq type [prod]) in Coq; a pair of proofs is represented by their pair (Coq function [pair]) in Coq.

4. Duplication of hypotheses is represented by the Coq function (fun x => (x,x)). Dereliction of hypotheses is represented by the coq function (fun (x,y) => x) or (fun (x,y) => y). Exchange of the order of hypotheses is represented by the Coq function (fun (x,y) => (y,x)).

The above can be done using lists instead of tuples.

The advantage of this approach is that it requires a minimum amount of syntax, and takes maximum advantage of Coq's automation facilities.

The disadvantage is that one cannot reason about proof-theoretic properties *generically* across different signatures and theories. Each signature has its own type of judgments, and each theory has its own type of proofs. In the present development we will want to prove -- in this generic manner -- that various classes of natural deduction calculi form various kinds of categories. So we will need this ability to reason about proofs independently of the type used to represent judgments and (more importantly) of the set of basic inference rules.

2) Structurally explicit proofs

Structurally explicit proofs are formalized in this file (NaturalDeduction.v) and are designed specifically in order to circumvent the problem in the previous paragraph.
These proofs are actually structurally explicit on (potentially) two different levels. The beginning of this file formalizes natural deduction proofs with explicit structural operations for manipulating lists of judgments – for example, the open hypotheses of an incomplete proof. The class TreeStructuralRules further down in the file instantiates ND such that Judgments is actually a pair of trees of propositions, and there will be a whole *other* set of rules for manipulating the structure of a tree of propositions *within* a single judgment.

The flattening functor ends up mapping the first kind of structural operation (moving around judgments) onto the second kind (moving around propositions/types). That’s why everything is so laboriously explicit – there’s important information in those structural operations.

(*
* REGARDING LISTS versus TREES:
*
* You’ll notice that this formalization uses (Tree (option A)) in a lot of places where you might find (list A) more natural. If this bothers you, see the end of the file for the technical reasons why. In short, it lets us avoid having to mess about with JMEq or EqDep, which are not as well-supported by the Coq core as the theory of CiC proper.
*)

Section Natural_Deduction.

(* any Coq Type may be used as the set of judgments *)
Context {Judgment : Type}.

(* any Coq Type -- indexed by the hypothesis and conclusion judgments -- may be used as the set of basic inference rules *)
Context {Rule : forall (hypotheses:Tree ??Judgment)(conclusion:Tree ??Judgment), Type}.

(*
* This type represents a valid Natural Deduction proof from a list (tree) of hypotheses; the notation H/···\(\lor\)C is meant to look like a proof tree with the middle missing if you tilt your head to
* the left (yeah, I know it's a stretch). Note also that this
* type is capable of representing proofs with multiple
* conclusions, whereas a Rule may have only one conclusion.
*)

Inductive ND :
  forall hypotheses:Tree ??Judgment,
  forall conclusions:Tree ??Judgment,
  Type :=

(* natural deduction: you may infer nothing from nothing *)
| nd_id0 : [ ] /\ · · · · · · [ ]

(* natural deduction: you may infer anything from itself -- "identity proof" *)
| nd_id1 : forall h, [ h ] /\ · · · · · · [ h ]

(* natural deduction: you may discard conclusions *)
| nd_weak1 : forall h, [ h ] /\ · · · · · · [ ]

(* natural deduction: you may duplicate conclusions *)
| nd_copy : forall h, h /\ · · · · · · (h,,h)

(* natural deduction: you may write two proof trees side by side on a piece of paper -- "proof product" *)
| nd_prod : forall {h1 h2 c1 c2}
  (pf1: h1 /\ · · · · · · /\ c1)
  (pf2: h2 /\ · · · · · · /\ c2),
  ( h1 ,, h2 /\ · · · · · · /\ c1 ,, c2)

(* natural deduction: given a proof of every hypothesis, you may discharge them -- "proof composition" *)
| nd_comp :
  forall {h x c}
  '(pf1: h /\ · · · /\ x)
  '(pf2: x /\ · · · /\ c),
  ( h /\ · · · /\ c)

(* Structural rules on lists of judgments - note that this is completely separate from the structural
* rules for *contexts* within a sequent. The rules below manipulate lists of *judgments* rather than
* lists of *propositions*. *)
| nd_cancell : forall {a}, [ ] ,, a /\ · · · /\ a
| nd_cancellr : forall {a}, a ,, [ ] /\ · · · /\ a
| nd_llecnac : forall {a}, a /\ · · · /\ [] ,, a
| nd_rlecnac : forall {a}, a /\ · · · /\ a ,, []
nd_assoc : forall {a b c}, (a,,b),,c /· · · /· a,,(b,,c)
nd_cossa : forall {a b c}, a,,(b,,c) /· · · /· (a,,b),,c

(* any Rule by itself counts as a proof *)
nd_rule : forall {h c} (r:Rule h c), h /· · · /· c

where "H /· · · /· C" := (ND H C).

Notation "H /· · · /· C" := (ND H C) : pf_scope.
Notation "a ;; b" := (nd_comp a b) : nd_scope.
Notation "a ** b" := (nd_prod a b) : nd_scope.
Open Scope nd_scope.
Open Scope pf_scope.

(* a predicate on proofs *)
Definition NDPredicate := forall h c, h /· · · /· c -> Prop.

(* the structural inference rules are those which do not change, add, remove, or re-order the judgments *)
Inductive Structural : forall {h c}, h /· · · /· c -> Prop :=
| nd_structural_id0 : Structural nd_id0
| nd_structural_id1 : forall h, Structural (nd_id1 h)
| nd_structural_cancell : forall {a}, Structural (@nd_cancell a)
| nd_structural_cancelr : forall {a}, Structural (@nd_cancelr a)
| nd_structural_llecnac : forall {a}, Structural (@nd_llecnac a)
| nd_structural_rlecnac : forall {a}, Structural (@nd_rlecnac a)
| nd_structural_assoc : forall {a b c}, Structural (@nd_assoc a b c)
| nd_structural_cossa : forall {a b c}, Structural (@nd_cossa a b c)
.

(* the closure of an NDPredicate under nd_comp and nd_prod *)
Inductive NDPredicateClosure (P:NDPredicate) : forall {h c}, h /· · · /· c -> Prop :=
| ndpc_p : forall h c f, P h c f -> NDPredicateClosure P f
| ndpc_prod : forall '(pf1:h1/· · · /· c1) (pf2:h2/· · · /· c2), NDPredicateClosure P pf1 -> NDPredicateClosure P pf2 -> NDPredicateClosure P (pf1**pf2)
| ndpc_comp : forall '(pf1:h1/· · · /· x) (pf2: x/· · · /· c2), NDPredicateClosure P pf1 -> NDPredicateClosure P pf2 -> NDPredicateClosure P (pf1;;pf2).

(* proofs built up from structural rules via comp and prod *)
Definition StructuralND {h}{c} f := @NDPredicateClosure (@Structural) h c f.

(* The Predicate (BuiltFrom f P h) asserts that "h" was built from a single occurrence of "f" and proofs which satisfy P *)
Inductive BuiltFrom {h'}{c'}(f:h'/c')(P:NDPredicate) : forall {h c}, h'/c -> Prop :=
| builtfrom_refl : BuiltFrom f P f |
| builtfrom_P : forall h c g, @P h c g -> BuiltFrom f P g |
| builtfrom_prod1 : forall h1 c1 f1 h2 c2 f2, P h1 c1 f1 -> @BuiltFrom _ _ f P h2 c2 f2 -> BuiltFrom f P (f1 ** f2) |
| builtfrom_prod2 : forall h1 c1 f1 h2 c2 f2, P h1 c1 f1 -> @BuiltFrom _ _ f P h2 c2 f2 -> BuiltFrom f P (f2 ** f1) |
| builtfrom_comp1 : forall h x c f1 f2, P h x f1 -> @BuiltFrom _ _ f P x c f2 -> BuiltFrom f P (f1 ;; f2) |
| builtfrom_comp2 : forall h x c f1 f2, P x c f1 -> @BuiltFrom _ _ f P h x f2 -> BuiltFrom f P (f2 ;; f1).

(* multi-judgment generalization of nd_id0 and nd_id1; making nd_id0/nd_id1 primitive and deriving all other *)
Fixpoint nd_id (sl:Tree ??Judgment) : sl / [] :=
match sl with
| T_Leaf None => nd_id0 |
| T_Leaf (Some x) => nd_id1 x |
| T_Branch a b => nd_prod (nd_id a) (nd_id b)
end.

Fixpoint nd_weak (sl:Tree ??Judgment) : sl / [] :=
match sl as SL return SL / [] with
| T_Leaf None => nd_id0 |
| T_Leaf (Some x) => nd_weak1 x |
| T_Branch a b => nd_prod (nd_weak a) (nd_weak b) ;; nd_cancelr
end.

Hint Constructors Structural.
Hint Constructors BuiltFrom.
Hint Constructors NDPredicateClosure.

Hint Extern 1 => apply nd_structural_id0.
Hint Extern 1 => apply nd_structural_id1.
Hint Extern 1 => apply nd_structural_cancell.
Hint Extern 1 => apply nd_structural_cancelr.
Hint Extern 1 => apply nd_structural_llecnc.
Hint Extern 1 => apply nd_structural_rlecnc.
Hint Extern 1 => apply nd_structural_assoc.
Hint Extern 1 => apply nd_structural_cossa.
Hint Extern 1 => apply ndpc_p.
Hint Extern 1 => apply ndpc_prod.
Hint Extern 1 => apply ndpc_comp.

Lemma nd_id_structural :forall sl, StructuralND (nd_id sl).
intros.
induction sl; simpl; auto.
destruct a; auto.
Defined.

(* An equivalence relation on proofs which is sensitive only to the logical content of the proof -- insensitive to
* structural variations *)
Class ND_Relation :=
{ ndr_eqv : forall {h c }, h /······ c -> h /······ c -> Prop where "pf1 == pf2" := (@ndr_eqv _ _ pf1 pf2)
; ndr_eqv_equivalence : forall h c, Equivalence (@ndr_eqv h c)
(* the relation must respect composition, be associative wrt composition, and be left and right neutral wrt the identity proof *)
; ndr_comp_respects : forall {a b c}(f f':a/······b)(g g':b/······c), f == f' -> g == g' -> f;;g == f';;g'
; ndr_comp_associativity : forall '(f:a/······b)'(g:b/······c)'(h:c/······d), (f;;g);;h == f;;(g;;h)
(* the relation must respect products, be associative wrt products, and be left and right neutral wrt the identity proof *)
; ndr_prod_respects : forall {a b c d}(f f':a/······b)(g g':c/······d), f==f' -> g==g' -> f**g == f'**g'
; ndr_prod_associativity : forall '(f:a/······a)'(g:b/······b)'(h:c/······c)', (f**g)**h == nd_assoc ;; f**(g**h) ;; nd_cossa
(* products and composition must distribute over each other *)
; ndr_prod_preserves_comp : forall '(f:a/······b)'(f':a'/······b')'(g:b/······c)'(g':b'/······c'), (f;;g)**(f';;g') == (f**f');(g**g')
(* Given a proof f, any two proofs built from it using only structural rules are indistinguishable. Keep in mind that
* nd_weak and nd_copy aren't considered structural, so the hypotheses and conclusions of such proofs will be an identical
* list, differing only in the "parenthesization" and addition or removal of empty leaves. *)
; ndr_builtfrom_structural : forall '(f:a/······b){a' b'}(g1 g2:a'/······b'),
   BuiltFrom f (@StructuralND) g1 ->
   BuiltFrom f (@StructuralND) g2 ->
   g1 == g2
(* proofs of nothing are not distinguished from each other *)
; ndr_void_proofs_irrelevant : forall '(f:a/······)()(g:a/······), f == g
(* products and duplication must distribute over each other *)
; ndr_prod_preserves_copy : forall '(f:a/······b),
    nd_copy a;; f**f == f ;; nd_copy b
(* duplicating a hypothesis and discarding it is irrelevant *)
; ndr_copy_then_weak_left : forall a,
    nd_copy a;; (nd_weak _ ** nd_id _) ;; nd_cancell == nd_id _
; ndr_copy_then_weak_right : forall a,
    nd_copy a;; (nd_id _ ** nd_weak _) ;; nd_canceller == nd_id _
}.

(*
* Natural Deduction proofs which are Structurally Implicit on the level of judgments. These proofs have poor compositionality properties (vertically, they look more like lists than trees) but are easier to do induction over.
* )

Inductive SIND : Tree ??Judgment -> Tree ??Judgment -> Type :=
  | scnd_weak : forall c , SIND c []
  | scnd_comp : forall ht ct c , SIND ht ct -> Rule ct [c] -> SIND ht [c]
  | scnd_branch : forall ht c1 c2, SIND ht c1 -> SIND ht c2 -> SIND ht (c1,,c2)
.
Hint Constructors SIND.
(* Any ND whose primitive Rules have at most one conclusion (note that nd_prod is allowed!) can be turned into an SIND. *)

Definition mkSIND (all_rules_one_conclusion : forall h c1 c2, Rule h (c1,,c2) -> False) :
  forall h x c, ND x c -> SIND h x -> SIND h c.
  intros h x c nd.
  induction nd; intro k.
    apply k.
    apply k.
    apply scnd_weak.
    eapply scnd_branch; apply k.
    inversion k; subst.
    apply (scnd_branch _ _ _ (IHnd1 X) (IHnd2 X0)).
    apply IHnd2.
    apply IHnd1.
    apply k.
    inversion k; subst; auto.
    inversion k; subst; auto.
    apply scnd_branch; auto.
    apply scnd_branch; auto.
    inversion k; subst; inversion X; subst; auto.
    inversion k; subst; inversion X0; subst; auto.
    destruct c.
    destruct o.
    eapply scnd_comp. apply k. apply r.
    eapply scnd_weak.
    set (all_rules_one_conclusion _ _ _ r) as bogus.
    inversion bogus.
    Defined.

(* a "ClosedSIND" is a proof with no open hypotheses and no multi-conclusion rules *)
Inductive ClosedSIND : Tree ??Judgment -> Type :=
| cnd_weak : ClosedSIND []
| cnd_rule : forall h c , ClosedSIND h -> Rule h c -> ClosedSIND c
| cnd_branch : forall c1 c2, ClosedSIND c1 -> ClosedSIND c2 -> ClosedSIND (c1,,c2).

(* we can turn an SIND without hypotheses into a ClosedSIND *)

Definition closedFromSIND h c (pn2:SIND h c)(cnd:ClosedSIND h) : ClosedSIND c.
refine ((fix closedFromPnodes h c (pn2:SIND h c)(cnd:ClosedSIND h) {struct pn2} :=
 (match pn2 in SIND H C return H=h -> C=c -> _ with
   | scnd_weak c => let case_weak := tt in _
   | scnd_comp ht ct c pn' rule => let case_comp := tt in let qq := closedFromPnodes _ _ pn' in _
   | scnd_branch ht c1 c2 pn' pn'' => let case_branch := tt in
     let q1 := closedFromPnodes _ _ pn' in
     let q2 := closedFromPnodes _ _ pn'' in _

   end (refl_equal _) (refl_equal _))) h c pn2 cnd).

destruct case_weak.
  intros; subst.
  apply cnd_weak.

destruct case_comp.
  intros.
  clear pn2.
  apply (cnd_rule ct).
  apply qq.
  subst.
  apply cnd0.
  apply rule.

destruct case_branch.
  intros.
  apply cnd_branch.
  apply q1. subst. apply cnd0.
  apply q2. subst. apply cnd0.
  Defined.

(* undo the above *)

Fixpoint closedNDtoNormalND {c}(cnd:ClosedSIND c) : ND [] c :=
match cnd in ClosedSIND C return ND [] C with
cnd_weak \Rightarrow \text{nd_id0}
cnd_rule h c cndh rhc \Rightarrow \text{closedNDtoNormalND} cndh ;; \text{nd_rule rhc}
cnd_branch c1 c2 cnd1 cnd2 \Rightarrow \text{nd_llecnc} ;; \text{nd_prod} (\text{closedNDtoNormalND} cnd1) (\text{closedNDtoNormalND} cnd2)
end.

(* Natural Deduction systems whose judgments happen to be pairs of the same type *)

Section SequentND.

Context \{S:Type\}. (* type of sequent components *)
Context (sequent:S\rightarrow S\rightarrow \text{Judgment}). (* pairing operation which forms a sequent from its halves *)
Notation "a \mid= b" := (sequent a b).

(* a SequentND is a natural deduction whose judgments are sequents, has initial sequents, and has a cut rule *)
Class SequentND :=
\{ snd_initial : forall a, 
\[\[\] /\cdots/ [ a \mid= a ] \]
; snd_cut : forall a b c, [ a \mid= b ] , , [ b \mid= c ] /\cdots/ [ a \mid= c ] \}\.

Context (sequentND:SequentND).
Context (ndr:ND_Relation).

(*
* A predicate singling out structural rules, initial sequents,
* and cut rules.
*
* Proofs using only structural rules cannot add or remove
* judgments - their hypothesis and conclusion judgment-trees will
* differ only in "parenthesization" and the presence/absence of
* empty leaves. This means that a proof involving only
* structural rules, cut, and initial sequents can ADD new
* non-empty judgment-leaves only via snd_initial, and can only
* REMOVE non-empty judgment-leaves only via snd_cut. Since the
* initial sequent is a left and right identity for cut, and cut
* is associative, any two proofs (with the same hypotheses and
* conclusions) using only structural rules, cut, and initial
* sequents are logically indistinguishable - their differences
* are logically insignificant.
*
* Note that it is important that nd_weak and nd_copy aren’t
* considered to be "structural".
*)
Inductive SequentND_Inert : forall h c, h/\cdots/c \rightarrow \text{Prop} :=
(* An ND_Relation for a sequent deduction should not distinguish between two proofs having the same hypotheses and conclusions * if those proofs use only initial sequents, cut, and structural rules (see comment above) *)

Class SequentND_Relation :=
{ sndr_ndr := ndr
; sndr_inert : forall a b (f g:a//b),
NDPredicateClosure SequentND_Inert f ->
NDPredicateClosure SequentND_Inert g ->
ndr_eqv f g }

End SequentND.

(* Deductions on sequents whose antecedent is a tree of propositions (i.e. a context) *)

Section ContextND.
Context {P:Type}{sequent:Tree ??P -> Tree ??P -> Judgment}.
Context {snd:SequentND(sequent:=sequent)}.
Notation "a | b" := (sequent a b).

(* Note that these rules mirror nd_{cancell,cancelr,rlecnac,llecnac,assoc,cossa} but are completely separate from them *)

Class ContextND :=
{ cnd_ant_assoc : forall x a b c, ND [(a,,b,,c) | x] [(a,,(b,,c)) | x ]
; cnd_ant_cossa : forall x a b c, ND [(a,,(b,,c)) | x] [((a,,b,,c)) | x ]
; cnd_ant_cancell : forall x a , ND [ [],a | x] [ a | x ]
; cnd_ant_cancelr : forall x a , ND [a,[], | x] [ a | x ]
; cnd_ant_llecnac : forall x a , ND [ a | x] [ [],a | x ]
; cnd_ant_rlecnac : forall x a , ND [ a | x] [ a,[], | x ]
; cnd_expand_left : forall a b c , ND [ a = b ] [ c,,a | c,,b]
; cnd_expand_right : forall a b c , ND [ a = b ] [ a,,c | b,,c]
; cnd_snd := snd
}.

Context (ContextND).

(*
 * A predicate singling out initial sequents, cuts, context expansion,
 * and structural rules.
 *)
* Any two proofs (with the same hypotheses and conclusions) whose
* non-structural rules do nothing other than expand contexts,
* re-arrange contexts, or introduce additional initial-sequent
* conclusions are indistinguishable. One important consequence
* is that asking for a small initial sequent and then expanding
* it using cnd_expand_{right,left} is no different from simply
* asking for the larger initial sequent in the first place.
*
*)

Inductive ContextND_Inert : forall h c, h/· · · · · ·/c -> Prop :=
| cnd_inert_initial : forall a, ContextND_Inert _ _ (snd_initial a)
| cnd_inert_cut : forall a b c, ContextND_Inert _ _ (snd_cut a b c)
| cnd_inert_structural : forall a b f, Structural f -> ContextND_Inert a b f
| cnd_inert_cnd_ant_assoc : forall x a b c, ContextND_Inert _ _ (cnd_ant_assoc x a b c)
| cnd_inert_cnd_ant_cossa : forall x a b c, ContextND_Inert _ _ (cnd_ant_cossa x a b c)
| cnd_inert_cnd_ant_cancell : forall x a , ContextND_Inert _ _ (cnd_ant_cancell x a)
| cnd_inert_cnd_ant_cancelr : forall x a , ContextND_Inert _ _ (cnd_ant_cancelr x a)
| cnd_inert_cnd_ant_llecnac : forall x a , ContextND_Inert _ _ (cnd_ant_llecnac x a)
| cnd_inert_cnd_ant_rlecnac : forall x a , ContextND_Inert _ _ (cnd_ant_rlecnac x a)
| cnd_inert_se Expand_left : forall t g s , ContextND_Inert _ _ (@cnd_expand_left _ t g s)
| cnd_inert_se Expand_right : forall t g s , ContextND_Inert _ _ (@cnd_expand_right _ t g s).

Class ContextND_Relation {ndr}{sndr:SequentND_Relation _ ndr} :=
{ cndr_inert : forall {a}{b}(f g:a/· · · · · ·/b),
  NDPredicateClosure ContextND_Inert f ->
  NDPredicateClosure ContextND_Inert g ->
  ndr_eqv f g
 ; cndr_sndr := sndr
}.

(* a proof is Analytic if it does not use cut *)
(*
Definition Analytic_Rule : NDPredicate :=
  fun h c f => forall c, not (snd_cut _ _ c = f).
Definition AnalyticND := NDPredicateClosure Analytic_Rule.

(* a proof system has cut elimination if, for every proof, there is an analytic proof with the same conclusion *)
Class CutElimination :=
{ ce_eliminate : forall h c, h/· · · /c -> h/· · · /c
 ; ce_analytic : forall h c f, AnalyticND (ce_eliminate h c f)
}.
(* cut elimination is strong if the analytic proof is furthermore equivalent to the original proof *)
Class StrongCutElimination :=
{ sce_ce <: CutElimination
; ce_strong : forall h c f, f === ce_eliminate h c f
}.
*)

End ContextND.

Close Scope nd_scope.
Open Scope pf_scope.

End Natural_Deduction.

Coercion snd_cut : SequentND >-> Funclass.
Coercion cnd_snd : ContextND >-> SequentND.
Coercion sndr_ndr : SequentND_Relation >-> ND_Relation.
Coercion cndr_sndr : ContextND_Relation >-> SequentND_Relation.

Implicit Arguments ND [ Judgment ].
Hint Constructors Structural.
Hint Extern 1 => apply nd_id_structural.
Hint Extern 1 => apply ndr_builtfrom_structural.
Hint Extern 1 => apply nd_structural_id0.
Hint Extern 1 => apply nd_structural_id1.
Hint Extern 1 => apply nd_structural_cancell.
Hint Extern 1 => apply nd_structural_cancelr.
Hint Extern 1 => apply nd_structural_llecnac.
Hint Extern 1 => apply nd_structural_rlecnac.
Hint Extern 1 => apply nd_structural_assoc.
Hint Extern 1 => apply nd_structural_cossa.
Hint Extern 1 => apply ndpc_p.
Hint Extern 1 => apply ndpc_prod.
Hint Extern 1 => apply ndpc_comp.
Hint Extern 1 => apply builtfrom_refl.
Hint Extern 1 => apply builtfrom_prod1.
Hint Extern 1 => apply builtfrom_prod2.
Hint Extern 1 => apply builtfrom_comp1.
Hint Extern 1 => apply builtfrom_comp2.
Hint Extern 1 => apply builtfrom_P.
Hint Extern 1 => apply snd_inert_initial.
Hint Extern 1 => apply snd_inert_cut.
Hint Extern 1 => apply snd_inert_structural.

Hint Extern 1 => apply cnd_inert_initial.
Hint Extern 1 => apply cnd_inert_cut.
Hint Extern 1 => apply cnd_inert_structural.
Hint Extern 1 => apply cnd_inert_cnd_ant_assoc.
Hint Extern 1 => apply cnd_inert_cnd_ant_cossa.
Hint Extern 1 => apply cnd_inert_cnd_ant_cancell.
Hint Extern 1 => apply cnd_inert_cnd_ant_cancelr.
Hint Extern 1 => apply cnd_inert_cnd_ant_llecnac.
Hint Extern 1 => apply cnd_inert_cnd_ant_rlecnac.
Hint Extern 1 => apply cnd_inert_se_expand_left.
Hint Extern 1 => apply cnd_inert_se_expand_right.

(* This first notation gets its own scope because it can be confusing when we're working with multiple different kinds of proofs. When only one kind of proof is in use, it's quite helpful though. *)
Notation "H /· · · · · · / C" := (@ND _ _ H C) : pf_scope.
Notation "a ;; b" := (nd_comp a b) : nd_scope.
Notation "a ** b" := (nd_prod a b) : nd_scope.
Notation "[# a #]" := (nd_rule a) : nd_scope.
Notation "a === b" := (@ndr_eqv _ _ _ _ _ a b) : nd_scope.

(* enable setoid rewriting *)
Open Scope nd_scope.
Open Scope pf_scope.

Add Parametric Relation {jt rt ndr h c} : (h/· · · /c) (@ndr_eqv jt rt ndr h c)  
  reflexivity proved by (@Equivalence_Reflexive _ _ (ndr_eqv_equivalence h c))  
  symmetry proved by (@Equivalence_Symmetric _ _ (ndr_eqv_equivalence h c))  
  transitivity proved by (@Equivalence_Transitive _ _ (ndr_eqv_equivalence h c))  
  as parametric_relation_ndr_eqv.
Add Parametric Morphism {jt rt ndr h x c} : (@nd_comp jt rt h x c)  
  with signature ((ndr_eqv(ND_Relation:=ndr)) => (ndr_eqv(ND_Relation:=ndr)) => (ndr_eqv(ND_Relation:=ndr)))  
  as parametric_morphism_nd_comp.
  intros; apply ndr_comp_respects; auto.
  Defined.
Add Parametric Morphism {jt rt ndr a b c d} : (@nd_prod jt rt a b c d)  
  with signature ((ndr_eqv(ND_Relation:=ndr)) => (ndr_eqv(ND_Relation:=ndr)) => (ndr_eqv(ND_Relation:=ndr)))

14
as parametric_morphism_nd_prod.
intros; apply ndr_prod_respects; auto.
Defined.

Section ND_Relation_Facts.
Context '{ND_Relation}.

(* useful *)
Lemma ndr_comp_right_identity : forall h c (f:h/· · · · · ·/c), ndr_eqv (f ;; nd_id c) f.
  intros; apply (ndr_builtfrom_structural f); auto.
Defined.

(* useful *)
Lemma ndr_comp_left_identity : forall h c (f:h/· · · · · ·/c), ndr_eqv (nd_id h ;; f) f.
  intros; apply (ndr_builtfrom_structural f); auto.
Defined.

End ND_Relation_Facts.

(* a generalization of the procedure used to build (nd_id n) from nd_id0 and nd_id1 *)
Definition nd_replicate :
  forall
    {Judgment}{Rule}{Ob}
    (h:Ob->Judgment)
    (c:Ob->Judgment)
    (j:Tree ??Ob),
    (forall (o:Ob), @ND Judgment Rule [h o] [c o]) ->
    @ND Judgment Rule (mapOptionTree h j) (mapOptionTree c j).
intros.
induction j; simpl.
destruct a; simpl.
apply X.
apply nd_id0.
apply nd_prod; auto.
Defined.

(* "map" over natural deduction proofs, where the result proof has the same judgments (but different rules) *)
Definition nd_map :
  forall
    {Judgment}{Rule0}{Rule1}
    (r:forall h c, Rule0 h c -> @ND Judgment Rule1 h c)
\{h}\{c\}
(pf:\&ND\ Judgment\ Rule0\ h\ c)
,
\&ND\ Judgment\ Rule1\ h\ c.
intros\ Judgment\ Rule0\ Rule1\ r.

refine\ ((fix\ nd_map\ h\ c\ pf\ \{struct\ pf\}:=\n\((\text{match}\ pf\ \text{in}\ \&ND\ _\ _\ H\ C\ \text{return}\ \&ND\ Judgment\ Rule1\ H\ C\ with}\n\begin{align*}\text{nd_id0} & \Rightarrow \text{let case_nd_id0 := tt in } \\
\text{nd_id1} & \Rightarrow \text{let case_nd_id1 := tt in } \\
\text{nd_weak1} & \Rightarrow \text{let case_nd_weak := tt in } \\
\text{nd_copy} & \Rightarrow \text{let case_nd_copy := tt in } \\
\text{nd_prod} & \_\ _\ _\ \_\ _\ _\ \text{lpf\ rpf} \Rightarrow \text{let case_nd_prod := tt in } \\
\text{nd_comp} & \_\ _\ _\ \text{top\ bot} \Rightarrow \text{let case_nd_comp := tt in } \\
\text{nd_rule} & \_\ _\ _\ \text{rule} \Rightarrow \text{let case_nd_rule := tt in } \\
\text{nd_cancell} & \_\ _\ _\ \Rightarrow \text{let case_nd_cancell := tt in } \\
\text{nd_canceirl} & \_\ _\ _\ \Rightarrow \text{let case_nd_canceirl := tt in } \\
\text{nd_llecnac} & \_\ _\ _\ \Rightarrow \text{let case_nd_llecnac := tt in } \\
\text{nd_rlecnac} & \_\ _\ _\ \Rightarrow \text{let case_nd_rlecnac := tt in } \\
\text{nd_assoc} & \_\ _\ _\ \Rightarrow \text{let case_nd_assoc := tt in } \\
\text{nd_cossa} & \_\ _\ _\ \Rightarrow \text{let case_nd_cossa := tt in } \\
\end{align*}\))\);\ simpl\ in\ *.

destruct\ case_nd_id0.\ apply\ nd_id0.
destruct\ case_nd_id1.\ apply\ nd_id1.
destruct\ case_nd_weak.\ apply\ nd_weak.
destruct\ case_nd_copy.\ apply\ nd_copy.
destruct\ case_nd_prod.\ apply\ (nd_prod\ (nd_map\ _\ _\ _\ \text{lpf})\ (nd_map\ _\ _\ _\ \text{rpf})).
destruct\ case_nd_comp.\ apply\ (nd_comp\ (nd_map\ _\ _\ _\ \text{top})\ (nd_map\ _\ _\ _\ \text{bot})).
destruct\ case_nd_cancell.\ apply\ nd_cancell.
destruct\ case_nd_canceirl.\ apply\ nd_canceirl.
destruct\ case_nd_llecnac.\ apply\ nd_llecnac.
destruct\ case_nd_rlecnac.\ apply\ nd_rlecnac.
destruct\ case_nd_assoc.\ apply\ nd_assoc.
destruct\ case_nd_cossa.\ apply\ nd_cossa.
apply\ r.\ apply\ rule.
Defined.
Definition nd_map'

: forall
{Judgment0}{Rule0}{Judgment1}{Rule1}
(f:Judgment0->Judgment1)
(r:forall h c, Rule0 h c -> ND Judgment1 Rule1 (mapOptionTree f h) (mapOptionTree f c))
{h}{c}
(pf@ND Judgment0 Rule0 h c)
, ND Judgment1 Rule1 (mapOptionTree f h) (mapOptionTree f c).
intros Judgment0 Rule0 Judgment1 Rule1 f r.

refine ((fix nd_map' h c pf {struct pf} :=
((match pf
  in @ND _ _ H C
  return
   @ND Judgment1 Rule1 (mapOptionTree f H) (mapOptionTree f C)
with
| nd_id0          => let case_nd_id0 := tt in _
| nd_id1 h       => let case_nd_id1 := tt in _
| nd_weak1 h     => let case_nd_weak := tt in _
| nd_copy h      => let case_nd_copy := tt in _
| nd_prod _ _ _ lpf rpf => let case_nd_prod := tt in _
| nd_comp _ _ _ top bot => let case_nd_comp := tt in _
| nd_rule _ _ rule => let case_nd_rule := tt in _
| nd_cancell _   => let case_nd_cancell := tt in _
| nd_cancellr _  => let case_nd_cancellr := tt in _
| nd_llecncac _  => let case_nd_llecncac := tt in _
| nd_rlecncac _  => let case_nd_rlecncac := tt in _
| nd_assoc _ _ _ => let case_nd_assoc := tt in _
| nd_cossa _ _ _ => let case_nd_cossa := tt in _
end))) ); simpl in *.
destruct case_nd_id0. apply nd_id0.
destruct case_nd_id1. apply nd_id1.
destruct case_nd_weak. apply nd_weak.
destruct case_nd_copy. apply nd_copy.
destruct case_nd_prod. apply (nd_prod (nd_map' _ _ lpf) (nd_map' _ _ rpf)).
destruct case_nd_comp. apply (nd_comp (nd_map' _ _ top) (nd_map' _ _ bot)).
destruct case_nd_cancell. apply nd_cancell.
destruct case Nd cancelr. apply nd_cancelr.
destruct case Nd llecnac. apply nd llecnac.
destruct case Nd rlecnac. apply nd rlecnac.
destruct case Nd assoc. apply nd assoc.
destruct case Nd cossa. apply nd cossa.
apply r. apply rule.
Defined.

(* witnesses the fact that every Rule in a particular proof satisfies the given predicate *)
Inductive nd_property {Judgment} {Rule}(P:forall h c, @Rule h c -> Prop) : forall {h}{c}, @ND Judgment Rule h c -> Prop :=
| nd_property_structural : forall h c pf, Structural pf -> @nd_property _ _ P h c pf
| nd_property_prod : forall h0 c0 pf0 h1 c1 pf1,
  @nd_property _ _ P h0 c0 pf0 -> @nd_property _ _ P h1 c1 pf1 -> @nd_property _ _ P _ _ (nd_prod pf0 pf1)
| nd_property_comp : forall h0 c0 pf0 c1 pf1,
  @nd_property _ _ P h0 c0 pf0 -> @nd_property _ _ P c0 c1 pf1 -> @nd_property _ _ P _ _ (nd_comp pf0 pf1)
| nd_property_rule : forall h c r, P h c r -> @nd_property _ _ P h c (nd_rule r).

Hint Constructors nd_property.

(* witnesses the fact that every Rule in a particular proof satisfies the given predicate (for ClosedSIND) *)
Inductive cnd_property {Judgment} {Rule}(P:forall h c, @Rule h c -> Prop) : forall {c}, @ClosedSIND Judgment Rule c -> Prop :=
| cnd_property_weak : @cnd_property _ _ P _ cnd_weak
| cnd_property_rule : forall h c r cnd',
  P h c r ->
  @cnd_property _ _ P h cnd' ->
  @cnd_property _ _ P c (cnd_rule _ _ cnd' r)
| cnd_property_branch :
  forall c1 c2 cnd1 cnd2,
  @cnd_property _ _ P c1 cnd1 ->
  @cnd_property _ _ P c2 cnd2 ->
  @cnd_property _ _ P _ (cnd_branch _ _ cnd1 cnd2).

(* witnesses the fact that every Rule in a particular proof satisfies the given predicate (for SIND) *)
Inductive scnd_property {Judgment} {Rule}(P:forall h c, @Rule h c -> Prop) : forall {h c}, @SIND Judgment Rule h c -> Prop :=
| scnd_property_weak : forall c, @scnd_property _ _ P _ _ (scnd_weak c)
| scnd_property_comp : forall h x c r cnd',
  P x [c] r ->
  @scnd_property _ _ P h x cnd' ->
  @scnd_property _ _ P h _ (scnd_comp _ _ cnd' r)
| scnd_property_branch :
  forall x c1 c2 cnd1 cnd2,
  @scnd_property _ _ P x c1 cnd1 ->
@scnd_property _ _ P x c2 cnd2 ->
@scnd_property _ _ P x _ (scnd_branch _ _ _ cnd1 cnd2).

(* renders a proof as LaTeX code *)

Section ToLatex.

Context {Judgment : Type}.
Context {Rule : forall (hypotheses:Tree ??Judgment)(conclusion:Tree ??Judgment), Type}.
Context {JudgmentToLatexMath : ToLatexMath Judgment}.
Context {RuleToLatexMath : forall h c, ToLatexMath (Rule h c)}.

Open Scope string_scope.

Definition judgments2latex (j:Tree ??Judgment) := treeToLatexMath (mapOptionTree toLatexMath j).

Definition eolL : LatexMath := rawLatexMath eol.

(* invariant: each proof shall emit its hypotheses visibly, except nd_id0 *)

Section SIND_toLatex.

(* indicates which rules should be hidden (omitted) from the rendered proof; useful for structural operations *)
Context (hideRule : forall h c (r:Rule h c), bool).

Fixpoint SIND_toLatexMath {h}{c}(pns:SIND(Rule:=Rule) h c) : LatexMath :=
match pns with
| scnd_branch ht c1 c2 pns1 pns2 => SIND_toLatexMath pns1 +++ rawLatexMath " \hspace{1cm} " +++ SIND_toLatexMath pns2
| scnd_weak c => rawLatexMath ""
| scnd_comp ht ct c pns rule => if hideRule _ _ rule
  then SIND_toLatexMath pns
  else rawLatexMath \tfrac{"+++ toLatexMath rule +++ rawLatexMath "}{"+++ eolL +++
                    SIND_toLatexMath pns +++ rawLatexMath "}" +++ eolL
                    toLatexMath c +++ rawLatexMath "}" +++ eolL
end.
End SIND_toLatex.

(* this is a work-in-progress; please use SIND_toLatexMath for now *)

Fixpoint nd_toLatexMath {h}{c}(nd:@ND _ Rule h c)(indent:string) :=
match nd with
| nd_id0 => rawLatexMath indent +++
  rawLatexMath "% nd_id0 " +++ eolL
| nd_id1 h' => rawLatexMath indent +++
end.
Close Scope pf_scope.
Close Scope nd_scope.