Reasoning about Categorial Type Logics

Pierre Castéran

About this course

- Objective: To show how a proof assistant like *Coq* can be useful in NL framework (syntax, semantics, ...)
- Level: Any level between "How to prove $P \Longrightarrow P$" and "how to prove soundness and completeness of Lambek calculus"
- Documentation:
  - Slides, *Coq* files, exercises on `esslli2004.loria.fr/casteran`,
  - The user contribution on Lambek calculus, on `coq.inria.fr`,
  - A book:
    "Interactive theorem proving and program development, Coq'Art: the Calculus of Inductive Constructions" ,
    Bertot and Castéran, Springer-Verlag, June 2004
  - A lot of documentation on `coq.inria.fr`

Overview of the course

1. Reasoning about CTL
2. A Gentle Introduction to Coq
3. Formalization of NL Lambek Calculus
4. Advanced Coq

The relative importance of 2, 3, and 4 will be determined according to your interest.

Reasoning about CTL

- A categorial grammar is determined by many parameters:
  - atomic types
  - modes
  - structural rules
  - lexicons
- It is hard to understand the consequences of the choice of these parameters on grammars we design
- A tool like Grail cannot handle "generic" properties of CTL, *i.e.* properties of classes of grammars.

Examples of derived rules

Co-application:

$$\frac{}{\forall A\, B,\ A \vdash (A \bullet B / B)}$$

Geach rule:

$$\frac{\mathbf{MA}(i,j)^1 \in R}{\forall\, A\, B\, C,\ B/_i C \vdash_{\mathbf{R}} (A/_j B)\backslash_j (A/_i C)}$$

$$\frac{}{^1\ \dfrac{\Gamma[(\Delta_1\,,\,(\Delta_2\,,\,\Delta_3)_i)_j] \vdash C}{\Gamma[((\Delta_1\,,\,\Delta_2)_j\,,\,\Delta_3)_i] \vdash C}\ MA(i,j)}$$

Some derived rules, like *unbounded dependencies* use auxiliary computations for reorganizing contexts:

$$\frac{\mathbf{L}_{\Diamond}(i,j) \in R \quad \phi_i(\Gamma, \langle\Delta\rangle_j) \vdash_{\mathbf{R}} C}{(\Gamma\,,\, \langle\Delta\rangle_j)_i \vdash_{\mathbf{R}} C}\ \mathrm{U}_{i,j}$$

$$\cfrac{\cfrac{\cfrac{\cfrac{\vdots}{(John\,,\,(believes\,,\,(Mary\,,\,(thinks\,,\,((the\,,\,girl)_a\,,\,(loves\,,\,\_ : \Diamond_c\Box_c np)_a)_a)_a)_a)_a)_a \vdash s}\ \mathrm{U}_{a,c}}{((John\,,\,(believes\,,\,(Mary\,,\,(thinks\,,\,((the\,,\,girl)_a\,,\,loves)_a)_a)_a)_a)_a\,,\,\_ : \Diamond_c\Box_c np)_a \vdash s}\ /_I}{(John\,,\,(believes\,,\,(Mary\,,\,(thinks\,,\,((the\,,\,girl)_a\,,\,loves)_a)_a)_a)_a)_a \vdash s/_a\Diamond_c\Box_c np}\ /_E}{(whom\,,\,(John\,,\,(believes\,,\,(Mary\,,\,(thinks\,,\,((the\,,\,girl)_a\,,\,loves)_a)_a)_a)_a)_a)_a \vdash n\backslash_a n}$$

whom	John , Mary	thinks, believes	the	girl
$(n\backslash_a n)/_a(s/_a \Diamond_c\Box_c\ np)$	np	$(np\backslash_a s)/_a s$	$np/_a n$	n

Proposition for a toolkit for studying CTL

- Multimodal framework
- Possibility of deriving/applying (arbitrarily complex) generic rules
- Decision procedures
- User interface

- Proofs and computations
- Higher-order programming and reasoning
- Interface with other tools (*Grail*)

A Gentle Introduction to Coq

- What's Coq?
- Terms and Types
- Propositions and Proofs
- Simple Induction

What's Coq?

- A computer tool for building/verifying theorem proofs.
- Domains: usual mathematics, proof theory, program verification ...
- Uses in logic and linguistics: partial knowledge (A. Nait Abdallah), categorial grammars, ...

Some characteristics

- A typed λ-calculus called the Calculus of Inductive Constructions:
  - Rich type system (polymorphic, dependent, higher-order, inductive types)
  - Computation facilities via reduction rules
  - Logical reasoning via Curry-Howard isomorphism
- Interactive developments via programmable tactics
- A standard library : theories, tactics, decision procedures, ...

Our presentation of *Coq* uses a small theory of `NL` grammars which can be downloaded.

How to use Coq?

- Interactive sessions, using a command language: the *Coq vernacular*,
- On a terminal (command `coqtop`), or under `[x]emacs`, by editing a vernacular file `foo.v` (`Proof General`, `coqIde`),
- With the graphical interface `Pcoq`,
- For large developments, use the `coqc` compiler (produces `.vo` files), makefile generation, ...

How to write a Coq File

The easiest way is to edit a file with suffix `.v` on *xemacs*:

```
xemacs foo.v
```

It runs *Proof General*. The blue region is read-only and contains the checked part of your file. The main commands are:

- `Ctrl-c Ctrl-n`: send next command (ending with `.<space>`), and extends the blue region.
- `Ctrl-c Ctrl-u`: undo last command and move the white/blue frontier upwards
- `Ctrl-c Ctrl-<ret>`: moves the white/blue frontier to the point
- `Ctrl-x Ctrl-s`: save
- `Ctrl-x Ctrl-c`: quit

If you are not familiar with *emacs*, you can use any editor, or copy/paste from a terminal to a `.v` file.

```
$ coqtop
Welcome to Coq 8.0 (Apr 2004)

Coq < Theorem my_first_thm : exists x:nat, x+x=x*x.
1 subgoal

  ============================
   exists x : nat, x + x = x * x

my_first_thm < Proof.

my_first_thm < exists 2.
```

```
1 subgoal

  ============================
   2 + 2 = 2 * 2

my_first_thm < simpl.
1 subgoal

  ============================
   4 = 4

my_first_thm < auto.
Proof completed.

my_first_thm < Qed.
```

```
 my_first_thm is defined

Coq < Print my_first_thm.
my_first_thm =
ex_intro (fun x : nat => x + x = x * x)
         2
         (refl_equal 4)
     : exists x : nat, x + x = x * x

ctrl-d
$
```

Terms and Types

The following session, using a module `NL.vo` we wrote, shows that every well formed term has a type, given by the command `Check`:

```
Require Import NL.

Check mary.
mary : word
Check _np.
_np : Atom
Check np.
np : Form
```

The type A of a term t may depend on some declarations gathered in a *context* Γ. This is expressed as a *typing judgment*:

$$\Gamma \vdash t : A$$

Other examples

```
Check 0.
0 : nat
Check -23.
-23 : Z
Check 2=3.
2=3 : Prop
Check False.
False : Prop
Check false.
false : bool
Check bool.
bool : Set
```

Typing rule for application

The following rule is used to control that functions are applied to arguments of the correct type:

$$\frac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B}\ app$$

```
Check atomic.
atomic : Form -> Prop

Check (atomic (np\\ s)).
atomic (np\\ s): Prop

Check (atomic -5).
Error: The term "atomic" has type "Form -> Prop"
 while it is expected to have type "Z"
```

Functions of several arguments

```
Check Zplus.
Zplus : Z -> Z -> Z

Check (Zplus (-2)).
(Zplus (-2):  Z -> Z

Check (Zplus (-2) 8).
-2 + 8 : Z

Check -2 + 8.
-2 + 8 : Z
```

Creating functions: The constructors of an inductive type

```
Print Form.

Inductive Form : Set :=
| At: Atom -> Form
| Slash: Form -> Form -> Form
| Backslash: Form -> Form -> Form
| Dot: Form -> Form -> Form

Check (Backslash np s).
np \\ s : Form

Check np o np \\ s.
np o np \\ s : Form
```

Creating functions: abstractions

$$\frac{\Gamma, (a : A) \vdash t : B}{\Gamma \vdash \mathit{fun}\, a : A \Rightarrow t \;:\; B}\; \mathit{lam}$$

```
Check (fun z:Z ⇒ 2*z).
fun z:Z ⇒ 2*z : Z → Z

Definition double z := 2*z.
double is defined

Definition lift A B  := B // (A \\ B).
lift is defined

Check lift.
lift : Form → Form → Form
```

Creating functions: primitive recursion

```
Fixpoint polarform (p:Atom)(F : Form){struct F}:Z :=
 match F with
         | (At a) => (if atom_eq_dec p a then 1 else 0)
         | A // B => polarform p A - polarform p B
         | B \\ A => polarform p A - polarform p B
         | A o B =>  polarform p A + polarform p B
 end.
```

```
polarform : Atom -> Form -> Z
```

Computing with reductions

The Calculus of Inductive Constructions is provided with various types of reduction, whose combination gives a strongly normalizing, confluent calculus.

```
Eval compute in 2*3.
= 6 : Z
```

```
Eval compute in  polarform _np (lift s np).
= 0 : Z
```

Propositions and Proofs

A *proposition* is any term A of the sort `Prop`. A *proof* of A is any term of type A.
In this logical framework, a type judgment $\Gamma \vdash t : A$ can be read as follows:

- Γ Hypotheses
- t Proof term for A
- A Theorem statement

The Curry-Howard Isomorphism

Maps every proposition to the type of its proofs.

- Proving a proposition A is building a term of type A
- $\longrightarrow$ is both functional arrow and (intuitionist) implication
- Applying a theorem (by modus ponens) is just like applying a function
- Programming and logical intuitions cooperate

Minimal Propositional Logic

```
Section Minimal_Propositional_Logic.
 Variables P Q R S : Prop.

 Check P -> P.
 P -> P : Prop
 Check P -> Q -> P.
 P -> Q -> P : Prop

 Check fun p:P ⇒ p.
 fun p:P ⇒ p : P -> P
 Check fun (p:P)(q:Q) ⇒ p.
 fun (p:P)(q:Q) ⇒ p  : P -> Q -> P
```

Saving theorems

```
Theorem delta : (P->P->Q)->P->Q.
Proof fun H p => (H p p).
delta  is defined

Theorem imp_trans : (P->Q) -> (Q->R) -> P->R.
Proof fun H H0 p ⇒ H0 (H p).
imp_trans is defined
```

Goals and tactics

- A *goal* is a pair of a context Γ and a type A (written $\Gamma \vdash A$)
- A *solution* of this goal is any term t such that the judgment $\Gamma \vdash t : A$ holds.
- A *tactic* is a function which transforms some goal into a sequence of *subgoals* and combines their solutions to solve the original goal.

Basic tactics: intro, apply and assumption

```
Theorem imp_trans : (P->Q)->(Q->R)->P->R.
Proof.

============================
(P -> Q) -> (Q -> R) -> P -> R

 intro H.

1 subgoal:
H : P -> Q
============================
(Q -> R) -> P -> R
```

```
H : P -> Q
=========================
(Q -> R) -> P -> R

 intros H' p.

1 subgoal:
  H : P -> Q
  H' : Q -> R
  p : P
  ===========
  R
```

```
H : P -> Q
H' : Q -> R
p : P
============
R
```

```
 apply H'.
```

```
1 subgoal:
H : P -> Q
H' : Q -> R
p : P
============
Q
```

```
H: P -> Q
H' : Q -> R
p : P
============
Q
```

```
 apply H.
```

```
1 subgoal:
H : P -> Q
H' : Q -> R
p : P
============
P
```

```
H : P -> Q
H' : Q -> R
p : P
==========
P

 assumption.

Proof completed
Qed.
imp_trans is defined
Print imp_trans.
= fun (H:P->Q) (H':Q->R)(p:P) => H' (H p)
: (P->Q)->(Q->R)->P->R
```

Some variants

```
Theorem imp_trans : (P->Q)->(Q->R)->P->R.
Proof.
  intros H H' p.
  apply H'; apply H; assumption.
Qed

Theorem imp_trans : (P->Q)->(Q->R)->P->R.
Proof.
  auto.
Qed
```

Thursday

You are now ready to write your own proofs . . . .

First, let us look again at some details.

Coq syntax

```
Section Minimal_Propositional_Logic.
 Variables P Q R S : Prop.

 Theorem imp_trans : (P->Q)->(Q->R)->P->R.
 Proof.
  intros H H0 p.
  apply H0.
  apply H.
  assumption.
 Qed.

End Minimal_Propositional_Logic.
```

How to use Coq?

- Interactive sessions, using a command language: the *Coq vernacular*,
- On a terminal (command `coqtop`), or under `[x]emacs`, by editing a vernacular file `foo.v` (`Proof General`, `coqIde`),
- With the graphical interface `Pcoq`,
- For large developments, use the `coqc` compiler (produces `.vo` files), makefile generation, ...

How to write a Coq File

The easiest way is to edit a file with suffix `.v` on *xemacs*:

```
xemacs foo.v
```

It runs *Proof General*. The blue region is read-only and contains the checked part of your file. The main commands are:

- `Ctrl-c Ctrl-n`: send next command (ending with `.<space>`), and extends the blue region.
- `Ctrl-c Ctrl-u`: undo last command and move the white/blue frontier upwards
- `Ctrl-c Ctrl-<ret>`: moves the white/blue frontier to the point
- `Ctrl-x Ctrl-s`: save
- `Ctrl-x Ctrl-c`: quit

If you are not familiar with *emacs*, you can use any editor, or copy/paste from a terminal to a `.v` file.

```
$ coqtop
Welcome to Coq 8.0 (Apr 2004)

Coq < Theorem my_first_thm : exists x:nat, x+x=x*x.
1 subgoal

  ============================
   exists x : nat, x + x = x * x

my_first_thm < Proof.

my_first_thm < exists 2.
```

```
1 subgoal

  ============================
   2 + 2 = 2 * 2

my_first_thm < simpl.
1 subgoal

  ============================
   4 = 4

my_first_thm < auto.
Proof completed.

my_first_thm < Qed.
```

```
 my_first_thm is defined

Coq < Print my_first_thm.
my_first_thm =
ex_intro (fun x : nat => x + x = x * x)
         2
         (refl_equal 4)
     : exists x : nat, x + x = x * x

ctrl-d
$
```

Exercise

Please look at `esslli2004.loria.fr/casteran` (first exercise). Replace the command `Admitted` with a real proof (a proof term or a sequence of tactic calls).

```
Section Minimal_Propositional_Logic.
 Variables P Q R S : Prop.

 Theorem imp_perm : (P->Q->R)->(Q->P->R).

 Theorem imp_dist : (P->Q->R)->(P->Q)->P->R.

 Theorem P3_Q : (((P->Q)->Q)->Q) -> P->Q.

 Theorem weak_peirce : ((((P->Q)->P)->P)->Q)->Q.

End Minimal_Propositional_Logic.
```

Complete the development in file `exercise1.v`, replacing the command `Admitted` by a real proof.
For each theorem:

- Give an explicit proof term
- Use tactics
- Use `auto`

Intuitionist Propositional Logic

For each connective, as well as `True` and `False`, we have

- typing rules for building propositions,
- introduction rules and tactics (except for `False`),
- elimination rules and tactics.

Conjunction

Typing rule: `and : Prop → Prop → Prop`

Introduction tactic : `split` transforms the subgoal $\Gamma \vdash A/\backslash B$ into two subgoals $\Gamma \vdash A$ and $\Gamma \vdash B$.

Elimination tactic: If $\Gamma \vdash t : A/\backslash B$, then `elim t` transforms the goal $\Gamma \vdash C$ into $\Gamma \vdash A \rightarrow B \rightarrow C$.

```
Theorem and_comm : P /\ Q -> Q /\ P.
Proof.
 intro H.
1 subgoal:
H :  P /\ Q
==========
Q /\ P

 elim H.

1 subgoal:
====================
 P -> Q -> Q /\ P

 intros; split; assumption.
Qed.
```

Disjunction

Typing rule: `or : Prop → Prop → Prop`

Introduction tactics : `left` (resp.`right`) transforms a subgoal $\Gamma \vdash A \backslash / B$ into the subgoal $\Gamma \vdash A$ (resp. $\Gamma \vdash B$).

Elimination tactic: If $\Gamma \vdash t : A \backslash / B$, then `elim t` transforms the goal $\Gamma \vdash C$ into the subgoals $\Gamma \vdash A \rightarrow C$ and $\Gamma \vdash B \rightarrow C$

```
Theorem or_comm : P \/ Q -> Q \/ P.
Proof.
 intro H; elim H.
2 subgoals
====================
 P  -> Q \/ P

subgoal 2 is:
====================
 Q  -> Q \/ P

 intro; right.
 intro; left.
Qed.
```

Falsehood and negation

Typing rules: `False : Prop`

`not := fun P : Prop ⇒False`

Introduction tactics :

- No introduction tactic for `False`!
- `red` transforms a subgoal $\Gamma \vdash {\sim}A$ into the subgoal $\Gamma \vdash A{\rightarrow}$`False`.

Elimination tactics:

- If $\Gamma \vdash t$: `False`, then `elim t` solves immediately any goal.
- If $\Gamma \vdash t : {\sim}A$, then `elim t` replaces the goal $\Gamma \vdash B$ by the subgoal $\Gamma \vdash A$.

```
Theorem contrap : (P -> Q) -> ~Q -> ~P.
Proof.
 intros H H0.
 red; intro p.

H :  P -> Q
H0 : ~Q
p : P
============
False
 elim H0; auto.
Qed.
```

Exercises (2)

Complete the development in file `exercise2.v`, replacing the command `Admitted` by a real proof.
For each theorem:

- Use tactics
- Use `auto` and/or `tauto`

```
Theorem P_or_False : P \/ False -> P.

Theorem and_idempotent_1 : P -> P /\ P.

Theorem and_idempotent_2 : P/\P -> P.

Theorem and_or_dist : (P\/Q)/\R -> (P/\R)\/(Q/\R).

Theorem absurd : P -> ~P -> Q.

Theorem demorgan1 : P /\ Q -> ~(~P \/ ~Q).

Theorem demorgan2 : ~(P \/ Q) -> ~P /\~Q.

Theorem demorgan3 : P \/ Q -> ~(~P /\ ~Q).
```

Predicate and Higher-order Logic

- Universal quantification
- Existential quantification
- Equality

Universal quantification (dependent product)

$$\frac{\Gamma, (x : A) \vdash B \ : \texttt{Prop}}{\Gamma \vdash \forall\, \texttt{x:A, B} \ : \texttt{Prop}}$$

$$\frac{\Gamma, (x : A) \vdash t : B}{\Gamma \vdash \texttt{fun}\, x : A \Rightarrow t \ : \forall\, \texttt{x:A, B}}$$

$$\frac{\Gamma \vdash f \ : \forall\, \texttt{x:A, B} \quad \Gamma \vdash t \ : A}{\Gamma \vdash f\ t \ : B\{x := t\}}$$

The symbol $\forall$ *is typed* `forall`

Tactics for the dependent products

Introduction tactic : `intro` y transforms a subgoal $\Gamma \vdash \forall\, \mathtt{x:A,\ B}$ into the subgoal $\Gamma, (y : A) \vdash B\{x := y\}$.

Elimination tactic If $\Gamma \vdash t : \forall\, \mathtt{x:A,\ B}$, then `apply t` solves the subgoal $\Gamma \vdash B\{x := t\}$.

Notice that there are some variants (see examples).

```
Theorem all_perm :
 ∀ (A:Set)(P : A -> A -> Prop),
   (∀ x y, P x y) ->
   ∀ x y, P y x.
Proof.
 intros A P H x y.

  A : Set
  P : A -> A -> Prop
  H : ∀ x y : A, P x y
  x : A
  y : A
  ============
  P y x

 apply H.
```

```
Print all_perm.
```

```
= fun (A:Set)(P: A -> A-> Prop)
      (H: ∀ x y:A, P x y)
      (x y: A) ⇒
      H y x
:  ∀(A:Set)(P : A -> A -> Prop),
   (∀x y, P x y) ->
   ∀x y, P y x
```

```
Lemma toy: (∀x y:Z, x < y)-> (∀x y:Z, y < x).
 intros; apply all_perm.
 assumption.
Qed.
```

Existential quantification

$$\frac{\Gamma \vdash P : A \to \texttt{Prop}}{\Gamma \vdash ex\ P : \texttt{Prop}}$$

Notation $\exists\, x : A,\ P\, x$ is just an abreviation for
`ex (fun x:A⇒P x)`.

Introduction tactic : `exists` t transforms a subgoal $\Gamma \vdash \texttt{ex}\ P$ into the subgoal $\Gamma \vdash P\ t$.

Elimination tactic: If $\Gamma \vdash t : \texttt{ex}\ P$, then `elim` t transforms the subgoal $\Gamma \vdash A$ into $\Gamma \vdash \forall x : A, P\, x \to A$.

```
Theorem not_ex_all_not :
 ∀ (A:Set)(P : A -> Prop),
   ~(exists x, P x) ->
   ∀ x , ~ P x.
Proof.
 intros A P H x.
 red; intro H0.
 elim H.

  x : A
  H0 : P x
  ==========
  exists x:A, P x

 exists x; assumption.
Qed.
```

```
Theorem all_not_ex_not :  ∀ (A:Set)(P : A -> Prop),
  (∀ a, P a) -> ∼(exists a, ∼(P a)).
 red; intros A P H H0.

H : forall a : A, P a
H0 : exists a : A, ∼ P a
============================
False

 elim H0; intros b Hb.

H : forall a : A, P a
b : A
Hb : ∼ P b
============================
False
```

```
H : forall a : A, P a
b : A
Hb : ~ P b
============================
False
 elim Hb; apply H.
Qed.
```

Equality

$$\frac{\Gamma \vdash t_1 : A \quad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 = t_2 : \texttt{Prop}}$$

Introduction tactic : `reflexivity` solves the subgoal $\Gamma \vdash t_1 = t_2$ if t_1 and t_2 are convertible.

Elimination tactics : If $\Gamma \vdash t : t_1 = t_2$, then `rewrite` t transforms the subgoal $\Gamma \vdash P\ t_1$ into $\Gamma \vdash P\ t_2$.

Variant: `rewrite <-` t (right to left)

```
Theorem eq_trans:
 ∀ (A:Set)(a b c:A),
   a = b -> b = c -> a = c.
Proof.
 intros A a b c H H0.

  H : a = b
  H0 : b = c
  ==========
   a = c

 rewrite H; assumption.
Qed.
```

Notation: $t_1 <> t_2$ is an abreviation for $\sim t_1 = t_2$

```
Lemma no_diff : forall A:Set, A -> ~(forall x y: A, x<>y).
Proof.
 intros A a H.

  A : Set
  a : A
  H : forall x y : A, x <> y
  ============================
   False

 elim (H a a); trivial.
Qed.
```

Exercises (3)

Complete the development in file `exercise3.v`, replacing the command `Admitted` by a real proof.

Some questions are quite difficult but interesting.

Friday

You can now write simple proofs . . .

. . . but not all proofs are so simple.

Simple Induction

Let A be an inductive type.

- Each constructor c of A is an introduction rule:
  the associated tactic is `apply` c.
  Variants: `constructor` i, `split`, `left`, `right`, `exists` t,
  etc.
- The elimination tactic is `elim` t,
  variants: `induction` v, `case` t, etc.

Example

The two constructors for `nat` are `O:nat` and `S:nat→nat`. If $\Gamma \vdash$ `n:nat`, the tactic `elim n` transforms the goal $\Gamma \vdash P\ n$ into the two subgoals

- $\Gamma \vdash P\ O$
- $\Gamma \vdash \forall\, p : nat, P\ p \rightarrow P\ (S\ p)$

```
Theorem plus_assoc : forall n p q:nat, (n+p)+q = n+(p+q).
 intro n; elim n; simpl; auto.
Qed.
```

Notice the higher-order pattern-matching:

$$P\ n = \lambda\, p\, q.(n + p) + q = n + (p + q)$$

Formalization of the Lambek Calculus (NL)

Let us start with the axiomatic (à la Hilbert) formalization of the logic of residuation, as the binary relation $\underset{NL}{\longrightarrow}$ on `Form` (derivability).

In *Coq*, we define it as an inductive dependent type `NL_arrow`:

- `NL_arrow` has type `Form`$\rightarrow$`Form`$\rightarrow$`Set`
- Each rule of derivability is represented by a constructor.

The definition of NL_arrow

```
Inductive NL_arrow  : Form  -> Form  -> Set :=
   | one  :    forall A, NL_arrow A A
   | comp :    forall A B C, NL_arrow A B ->
                             NL_arrow B C ->
                             NL_arrow A C
   | beta :    forall A B C, NL_arrow (A o B) C ->
                             NL_arrow A (C // B)
   | beta' :   forall A B C , NL_arrow A (C // B) ->
                              NL_arrow (A o B) C
   | gamma :   forall A B C, NL_arrow (A o B) C ->
                             NL_arrow B (A \\ C)
   | gamma' :  forall A B C,  NL_arrow B (A \\ C) ->
                              NL_arrow (A o B) C.
```

```
Check one.
one : ∀ A : Form, A -NL->  A

Definition deriv0 : np \\ s -NL-> np \\ s.
 apply one.
Defined.
Print deriv0.
deriv0 = one (np \\ s) :
 np \\ s -NL-> np \\ s
```

```
Lemma scheme1 : forall A B, A -NL-> (A o B)//B.
 intros A B; apply beta; apply one.
Defined.

Lemma ex2 : (np\\s)//np  -NL->  ((np\\s)//np o np)//np.
 apply scheme1.
Defined.
```

Notice that one, beta and gamma are in the ctl base for auto.

```
Lemma scheme1 : forall A B, A -NL-> (A o B)//B.
 auto with ctl.
Defined.
```

More on apply

```
Definition  Dot_mono_left :
   forall A B C, A -NL-> C   ->
                 A o B -NL-> C o B.
 intros A B C H; apply beta'.

H : A -NL->  C
 ============================
A -NL->  (C o  B) //  B
```

```
H : A -NL->  C
 ============================
A -NL->  (C o  B) //  B

apply comp.

Error: generated subgoal "A -NL->  ?638"
 has metavariables in it

comp :    forall A B C,  A -NL-> B ->  B -NL-> C ->
                  A -NL-> C
```

Explicit argument for apply

```
  H : A -NL->  C
 ========================
A -NL->  (C o  B) //  B
```

```
apply comp with C; auto with ctl.
```

```
Defined.
```

Prolog-like resolution

```
H : A -NL->  C
 ========================
A -NL->  (C o  B) //  B

 eapply comp.

  H : A -NL->  C
  ==============
   A -NL->  ?74

subgoal 2 is:
 ?74 -NL->  (C o  B) //  B
```

first subgoal

```
H : A -NL->  C
==============
A -NL->  ?74
```

second subgoal

```
 ?74 -NL->  (C o  B) //  B

 eexact H. (* or eauto *)


 C -NL->  (C o  B) //  B
eauto with ctl.
Defined.
```

A One-liner proof

```
Lemma  Dot_mono_left :
   forall A B C  : Form,
    A -NL-> C ->  A o B -NL-> C o B.
 intros; apply beta'; eauto with ctl.
Defined.
```

Exercises (4)

Derive the following rules (see `exercise4.v`).
Print the terms associed with your definitions.
Use as many derived rules and automatisms as possible.

To compile the theory NL, just download `distrib.tar.gz` and execute `tar zxvf distrib.tar.gz`, then goto the `exercises` subdirectory.
.../...

```
Definition deriv1 : s -NL-> (s o np) // np.

Definition deriv2 : (s//np) o np -NL-> s.

Definition deriv3 : np o (np \\ (s // np)) o np -NL-> s.

Definition deriv4 : (np//n)o(n//n o n) -NL-> np.

Definition  Dot_mono_right :
   forall A B C  : Form,
    B -NL-> C ->
    A o B -NL-> A o C.
```

```
Definition Dot_mono : forall A B C D,
     A -NL-> B ->
     C -NL-> D ->
     A o C -NL-> B o D.

Definition isotonicity :
   forall A B C : Form , A -NL-> B ->
                         A // C -NL-> B // C.

Definition antitonicity:
   forall A B C : Form ,  C -NL-> B ->
                          A // B -NL-> A // C.
```

```
Definition antitonicity':
   forall A B C : Form ,
   A -NL-> B ->
   B \\ C -NL-> A \\ C.

Definition isotonicity' :
   forall A B C : Form,
   B -NL-> C ->
   A \\ B -NL-> A \\ C.

Definition lifting:
  forall A B, A -NL-> B//(A\\B).

Definition lifting':
  forall A B, A -NL-> (B//A)\\B.
```

Advanced Coq

- Induction on dependent types
- Building new tactics
- Higher-Order reasoning

Induction on derivations

Let us consider a goal of the form:

```
A : Form
B : Form
d : A -NL-> B
==============
P A B
```

The tactic elim d generates the following goals:

```
∀A, P A A
```

```
∀ A B C : Form,
   (A -NL-> B) -> P A B ->
   (B -NL-> C) -> P B C ->
   P A C
```

etc.

Example : an invariant on derivations

```
Theorem Hilbert_polar : forall A B a, A -NL-> B ->
                                 polarform a A = polarform a B.
 intros A B a H; elim H; simpl; auto with zarith.
Qed.
```

In fact *Coq* computed the following elimination predicate:

```
P = fun A B ⇒
   ∀a,  polarform a A = polarform a B
```

Example: How to falsify a derivation

```
Lemma ex5  : np o np//s//np o np  -NL-> s
             ->  False.
 intro H;  elim H.
```

```
6 subgoals

  H: np o  np //  s //  np o  np -NL->  s
  ============================
  forall A: Form, False
...
```

The elimination predicate is `fun A B ⇒ False` !

A better attempt

```
Lemma ex5  : np o np//s//np o np  -NL-> s  ->  False.
 intro H; generalize (Hilbert_polar  _np H).

H : np o  np//s//np o  np  -NL->  s
============================
polarform _np (np o  np//s//np o  np) =  polarform _np s
 -> False
 simpl.

============================
4 = 0 -> False
 auto with zarith.
Qed.
```

Other applications of induction

- Definition of sequent calculus and natural deduction as inductive types
- Proof of equivalence between the three systems.

```
Definition NL_arrowToseq :
  forall (A B : Form),
  A -NL-> B -> (form A) ==> B.
 intros A B H; elim H.
 ....
Defined.
```

See `../Light/*.v`

Building new tactics

Coq is not an automatic theorem prover, but

- We can build `Hint` databases (for `auto` and `eauto`)
- There exist some automatic tactics for useful fragments: `tauto`, `intuition`, `omega`, `ring`, etc.
- We can program new tactics with `Ltac`

A tactic for falsifying a derivation

```
Ltac noderiv H atom :=
 match goal with
   H : NL_arrow ?A ?B |- False =>
         generalize(Hilbert_polar  atom H);
         simpl ;  auto with zarith
 end.

Lemma ex5' : np o np//s//np o np  -NL-> s
              ->  False.
 intro H;  noderiv H _s.
Defined.
```

Higher-Order reasoning

```
Section semantic_defs.

(* Kripke models for CTL *)

Variables (W : Set) (* worlds *)
          (R : W -> W -> W -> Prop) (* accessibility *)
          (v_at : Atom -> W -> Prop). (* valuation  *)
```

```
Fixpoint val (F : Form) : W -> Prop :=
  match F with
  | At a => v_at a
  | A o B =>
      fun x => exists y : W,
     (exists z : W, R x y z /\ val A y /\ val B z)
  | C // B =>
       fun y => forall x z : W, R x y z -> val B z ->
       val C x
  | A \\ C =>
       fun z => forall x y : W, R x y z -> val A y ->
    val C x
  end.
```

```
Definition satisfies (w : W) (A : Form) : Prop
                                        := val A w.

End semantic_defs.

Definition sem_implies : Form  -> Form -> Prop :=
   fun A B : Form  =>
    forall (W : Set) (R : W -> W -> W -> Prop)
           (v_at : Atom -> W -> Prop),
    forall w : W, satisfies R v_at w A ->
                  satisfies R v_at w B.
```

```
Lemma GAMMA :
 forall A B C,
 sem_implies (A o B) C -> sem_implies B (A \\ C).
Proof.
 unfold sem_implies, satisfies in |- *;
  simpl in |- *; auto.
 intros  A B C H W R v_at  w H1 x y H2 H3.
 apply H.
 exists y; exists w; auto.
Qed.
```

```
Definition NL_sound   :=
  forall A B : Form, NL_arrow  A B -> sem_implies A B.

Theorem NL_sound_thm : NL_sound.
Proof.
 unfold NL_sound.
 simple induction 1.
 apply ONE.
 intros; eapply COMP; eauto.
 intros; apply GAMMA'BETA; apply GAMMA; auto.
 intros; apply BETA'; assumption.
 intros; apply GAMMA; auto.
 intros; apply GAMMA'; assumption.
Qed.
```

Extensions of the development

- Polymorphism
- Variants of Lambek calculus
- Multimodal categorial grammars

Polymorphism

The type `At` is now arbitrary, and locally declared:

```
Section lambek_defs.
 Variable A:Set.
 Inductive Form  : Set :=
   | At : A -> Form
   | Slash : Form -> Form  -> Form
   | Dot : Form  -> Form  -> Form
   | Backslash : Form  -> Form  -> Form .
...
```

All types and constants are generalized at the end of the section:

```
End lambek_defs.

Check Form.
Form : Set -> Set

Check Dot.
Dot: forall A:Set, Form A -> Form A -> Form A

Check NL_arrow.
NL_arrow: forall A:Set, Form A -> Form A -> Set
```

Variants of Lambek Calculus

```
Definition structrules := Form -> Form -> Set.

Inductive NL : structrules :=.

Inductive L : structrules :=
  L1 : forall A B C, L (A o (B o C)) ((A o B) o C)
| L2 : forall A B C, L ((A o B) o C) (A o (B o C)).

Inductive P : structrules :=
  P1 : forall A B, P (A o B) (B o A).

Definition LP := union L P.
```

The type of the derivation relation is now:

```
forall (At:Set)(R: structrules At),
  Form At -> Form At -> Set
```

Completeness, soundness, cut-elimination, equivalence between the three calculus, are proved for **L**, **NLP**, and **LP** (look at Coq contrib on Lambek calculus).

Multimodal categorial grammars

We simply add new parameters:

- Uunary operators: $\Box$ and $\Diamond$
- Composition modes, for labeling the binary operators
- Unary modes, for labeling $\Box$ and $\Diamond$
- Interaction principles, generalizing the structural rules
- Meta-theorems and tactics parameterized by the structural rules (look at the reader)

Present state of the development

- syntax/semantics interface (beginning)
- To do:
  - Libraries of modes,
  - Graphical interface,
  - Interface with Grail
  - Montague semantics with non simply typed λ-calculi (2nd-order types, dynamic aspects, etc.)

A last (long) exercise

In the (monomorphic) system of `../Light/`, consider extensions by structural rules, like associativity and/or commutativity of the o operator.
For instance, you should derive the Geach rule : if o is associative, then `A//B` can be derived into `(A//C)//(B//C)`.

Conclusion

Proof assistants like *Coq* seem to be a good tool for exploring complex theories. They are a good compromise between a hopeless automaticity and hand-made proofs and computations.