Assistants de preuve

MPRI - 2007-2008

2- Les spécificités du Calcul des Constructions Inductives

Types inductifs récursifs : l'exemple des entiers

On a analyse de cas et construction par cas : le terme

$$
\begin{array}{l}
\lambda P : nat \to s. \\
\lambda H_O : P(O). \\
\lambda H_S : \forall m : nat.\, P(S\ m). \\
\lambda n : nat. \\
\ \texttt{match}\ n\ \texttt{as}\ y\ \texttt{return}\ P(y)\ \texttt{with} \\
\ \ |\ O\ \texttt{=>}\ H_O \\
\ \ |\ S\ m\ \texttt{=>}\ H_S\ m \\
\ \texttt{end}
\end{array}
$$

est une preuve de

$$\forall P : nat \to s.\, P(O) \to (\forall m : nat.\, P(S\ m)) \to \forall n : nat.\, P(n)$$

Comment dériver le schéma de récursion standard ?

L'opérateur de point-fixe (première étape)

Ajout d'une expression de point fixe anonyme typée

$$(\texttt{fix}\ f\ (x : A)\ :\ B\ :=\ t(f, x))$$

L'opérateur de point-fixe (première étape)

Ajout d'une expression de point fixe anonyme typée

$$(\texttt{fix}\ f\ (x : A)\ :\ B\ :=\ t(f, x))$$

. . . et tant qu'à faire, d'une expression de point fixe anonyme à résultat dépendant

$$(\texttt{fix}\ f\ (x : A)\ :\ B(x)\ :=\ t(f, x))$$

L'opérateur de point-fixe (première étape)

Ajout d'une expression de point fixe anonyme typée

$$(\texttt{fix}\ f\ (x : A)\ :\ B\ :=\ t(f, x))$$

. . . et tant qu'à faire, d'une expression de point fixe anonyme à résultat dépendant

$$(\texttt{fix}\ f\ (x : A)\ :\ B(x)\ :=\ t(f, x))$$

Comparaison avec le `let rec` à la ML (point fixe nommé)

$$(\texttt{fix}\ f\ (x : A)\ :\ B(x)\ :=\ t(f, x))$$

$$=$$

$$\texttt{let rec}\ f\ (x : A) = t(f, x)\ \texttt{in}\ f$$

L'opérateur de point-fixe (première étape)

Ajout d'une expression de point fixe anonyme typée

$$(\texttt{fix}\ f\ (x : A)\ :\ B\ :=\ t(f, x))$$

. . . et tant qu'à faire, d'une expression de point fixe anonyme à résultat dépendant

$$(\texttt{fix}\ f\ (x : A)\ :\ B(x)\ :=\ t(f, x))$$

Comparaison avec le `let rec` à la ML (point fixe nommé)

$$(\texttt{fix}\ f\ (x : A)\ :\ B(x)\ :=\ t(f, x))$$

$$=$$

$$\texttt{let rec}\ f\ (x : A) = t(f, x)\ \texttt{in}\ f$$

Règle de réduction (première approximation) : le dépliage du point-fixe

$$(\texttt{fix}\ f\ (x : A)\ :\ B(x)\ :=\ t(f, x))\ u\ \rightarrow\ t((\texttt{fix}\ f\ (x : A)\ :\ B(x)\ :=\ t(f, x)), u)$$

L'opérateur de point-fixe : application

de la construction par cas au récurseur sur les entiers

la construction par cas

$\lambda P : nat \to s.$
$\lambda H_O : P(O).$
$\lambda H_S : \forall m : nat.\, P(S\ m).$
$\lambda n : nat.$
`match` n `as` y `return` $P(y)$ `with`
$\mid O$ `=>` H_O
$\mid S\ m$ `=>` $H_S\ m$
`end`

est de type

$\forall P : nat \to s.$
$P(O) \to$
$(\forall m : nat.\, P(S\ m)) \to$
$\forall n : nat.\, P(n)$

le récurseur

$\lambda P : nat \to s.$
$\lambda H_O : P(O).$
$\lambda H_S : \forall m : nat.\, P(m) \to P(S\ m).$
`fix` $f\ (n : nat)\ :\ P(n)\ :=$
`match` n `as` y `return` $P(y)$ `with`
$\mid O$ `=>` H_O
$\mid S\ m$ `=>` $H_S\ m\ (f\ m)$
`end`

est de type

$\forall P : nat \to s.$
$P(O) \to$
$(\forall m : nat.\, P(m) \to P(S\ m)) \to$
$\forall n : nat.\, P(n)$

L'opérateur de point-fixe : la question de la terminaison

Implémentation du Calcul des Constructions Inductives :

- repose sur la décidabilité du typage et de la conversion
- interdire le dépliage ad infinitum des points fixes

Cohérence du Calcul des Constructions Inductives :

- interdire les preuves infinies telles que $(\texttt{fix}\ f\ (n : nat)\ :\ False\ :=\ f\ n) : False$

$\hookrightarrow$ choix d'exiger un critère de bonne fondaison des points-fixes

L'opérateur de point-fixe : bonne fondaison

L'exigence du Calcul des Constructions Inductives :

- l'argument du point-fixe est de type inductif
- les appels récursifs sont sur des arguments *structurellement* plus petits

L'exemple du récurseur sur les entiers

$$
\begin{array}{l}
\lambda P : nat \to s. \\
\lambda H_O : P(O). \\
\lambda H_S : \forall m : nat.\, P(m) \to P(S\ m). \\
\texttt{fix}\ f\ (n : nat)\ :\ P(n)\ := \\
\quad \texttt{match}\ n\ \texttt{as}\ y\ \texttt{return}\ P(y)\ \texttt{with} \\
\quad\ |\ O\ \texttt{=>}\ H_O \\
\quad\ |\ S\ m\ \texttt{=>}\ H_S\ m\ (f\ m) \\
\quad \texttt{end}
\end{array}
$$

est correct vis à vis du CCI : appel récursif sur m qui est structurellement plus petit que n dans l'inductif nat.

L'opérateur de point-fixe : règle de typage (cas simplement typé)

$$\frac{I \text{ instance de famille inductive} \qquad \Gamma \vdash I : s \qquad \Gamma \vdash C \qquad \Gamma, x : I, f : I \to C \vdash t(x,f) : C \qquad t|_f^{\emptyset} <_I x}{\Gamma \vdash (\texttt{fix}\ f\ (x : I)\ :\ C\ :=\ t(x,f)) : I \to C}$$

où les clauses principales de $t|_f^\rho <_I x$ sont

$$\frac{z \in \rho \cup \{x\} \qquad u_1|_f^\rho <_I x\ \ldots\ u_n|_f^\rho <_I x \qquad A|_f^\rho <_I x \qquad \ldots\ t_i|_f^{\rho \cup \{x \in \vec{x_i} | x : \forall y : \vec{U}.I\,\vec{u}\}} <_I x\ \ldots}{\texttt{match}\ z\ u_1\ \ldots\ u_n\ \texttt{return}\ A\ \texttt{with}\ \ldots\ c_i\ \vec{x_i}\ \texttt{=>}\ t_i\ \ldots\ \texttt{end}|_f^\rho <_I x}$$

$$\frac{t \neq (z\ \vec{u}) \text{ pour } z \in \rho \cup \{x\} \qquad t|_f^\rho <_I x \qquad A|_f^\rho <_I x \qquad \ldots\ t_i|_f^\rho <_I x\ \ldots}{\texttt{match}\ t\ \texttt{return}\ A\ \texttt{with}\ \ldots\ c_i\ \vec{x_i}\ \texttt{=>}\ t_i\ \ldots\ \texttt{end}|_f^\rho <_I x}$$

$$\frac{y \in \rho}{f\ y|_f^\rho <_I x} \qquad \frac{t|_f^\rho <_I x \qquad u|_f^\rho <_I x \qquad t \neq f}{t\ u|_f^\rho <_I x} \qquad \frac{A|_f^\rho <_I x \qquad t|_f^\rho <_I x}{\lambda y : A.\, t|_f^\rho <_I x} \qquad \frac{A|_f^\rho <_I x \qquad t|_f^\rho <_I x}{\forall y : A.\, t|_f^\rho <_I x}$$

$$\frac{}{s|_f^\rho <_I x} \qquad \frac{}{y|_f^\rho <_I x} \qquad \frac{}{c_i|_f^\rho <_I x} \qquad \frac{}{I|_f^\rho <_I x}$$

\+ règles contextuelles pour le point fixe lui-même...

Note : seuls les arguments récursifs de *même* type sont considérés récursifs (sinon paradoxes avec l'imprédicativité)

L'opérateur de point-fixe : règle de typage (cas dépendant)

$$\frac{I \text{ instance de famille inductive} \qquad \Gamma, x : I \vdash C(x) \qquad \Gamma, x : I, f : (\forall x : I.\, C(x)) \vdash t(x, f) : C(x) \qquad t|_f^{\emptyset} <_I x}{\Gamma \vdash (\texttt{fix}\ f\ (x : I)\ :\ C(x)\ :=\ t(x, f)) : \forall x : I.\, C(x)}$$

où les clauses principales de $t|_f^\rho <_I x$ sont

$$\frac{z \in \rho \cup \{x\} \qquad u_1|_f^\rho <_I x\ \ldots\ u_n|_f^\rho <_I x \qquad A|_f^\rho <_I x \qquad \ldots\ t_i|_f^{\rho \cup \{x \in \vec{x_i} | x : \forall y : \vec{U}.\, I\, \vec{u}\}} <_I x\ \ldots}{\texttt{match}\ z\ u_1\ \ldots\ u_n\ \texttt{return}\ A\ \texttt{with}\ \ldots\ c_i\ \vec{x_i}\ \texttt{=>}\ t_i\ \ldots\ \texttt{end}|_f^\rho <_I x}$$

$$\frac{t \neq (z\ \vec{u}) \text{ pour } z \in \rho \cup \{x\} \qquad t|_f^\rho <_I x \qquad A|_f^\rho <_I x \qquad \ldots\ t_i|_f^\rho <_I x\ \ldots}{\texttt{match}\ t\ \texttt{return}\ A\ \texttt{with}\ \ldots\ c_i\ \vec{x_i}\ \texttt{=>}\ t_i\ \ldots\ \texttt{end}|_f^\rho <_I x}$$

$$\frac{y \in \rho}{f\ y|_f^\rho <_I x} \qquad \frac{t|_f^\rho <_I x \qquad u|_f^\rho <_I x \qquad t \neq f}{t\ u|_f^\rho <_I x} \qquad \frac{A|_f^\rho <_I x \qquad t|_f^\rho <_I x}{\lambda y : A.\, t|_f^\rho <_I x} \qquad \frac{A|_f^\rho <_I x \qquad t|_f^\rho <_I x}{\forall y : A.\, t|_f^\rho <_I x}$$

$$\frac{}{s|_f^\rho <_I x} \qquad \frac{}{y|_f^\rho <_I x} \qquad \frac{}{c_i|_f^\rho <_I x} \qquad \frac{}{I|_f^\rho <_I x}$$

\+ règles contextuelles pour le point fixe lui-même...

Note : seuls les arguments récursifs de *même* type sont considérés récursifs (sinon paradoxes avec l'imprédicativité)

Un peu de terminologie

Le Calcul des Constructions Inductives prédicative a les sortes `Prop`, `Set` $=$ $\texttt{Type}_0$, $\texttt{Type}_1$, $\texttt{Type}_2$, ...

`Prop` et `Set` sont dites petites (parce qu'elles ne typent aucune autre sorte)

Les sortes $\texttt{Type}_i$ (pour $i \geq 1$) sont dites grandes (parce qu'elles typent `Prop` et `Set`)

Restrictions d'élimination selon la sorte

Règle d'élimination du type $bool$ (vers toutes les sortes)

$$\frac{\Gamma \vdash t : bool \qquad \Gamma, x : bool \vdash A(x) : s \qquad \Gamma \vdash t_1 : A(true) \qquad \Gamma \vdash t_2 : A(false)}{\Gamma \vdash (\texttt{match}\ t\ \texttt{as}\ x\ \texttt{return}\ A(x)\ \texttt{with}\ true\ \texttt{=>}\ t_1 \mid false\ \texttt{=>}\ t_2\ \texttt{end}) : A(t)}$$

Règle d'élimination du type $or\ A\ B$ (seulement vers `Prop`)

$$\frac{\Gamma \vdash t : or\ A\ B \qquad \Gamma, x : or\ A\ B \vdash C(x) : \texttt{Prop} \qquad \Gamma, p : A \vdash t_1 : C(or\_introl\ p) \qquad \Gamma, q : B \vdash t_2 : C(or\_intror\ q)}{\Gamma \vdash (\texttt{match}\ t\ \texttt{as}\ x\ \texttt{return}\ C(x)\ \texttt{with}\ or\_introl\ p\ \texttt{=>}\ t_1 \mid or\_intror\ q\ \texttt{=>}\ t_2\ \texttt{end}) : C(t)}$$

L'élimination des types inductifs dans `Type` (hiérarchie prédicative) est sans restriction (*élimination faible – vers* `Prop` *et* `Set` *– et forte – vers* `Type`)

L'élimination des types inductifs dans `Prop` est restreinte :

- en général, on ne peut construire un type de `Type` à partir d'une proposition conformément à l'interprétation implicite de `Prop` comme « proof-irrelevant » (*élimination propositionnelle seulement*)
- exception : si le type dans `Prop` a zéro constructeur (type vide) ou un unique constructeur dont les arguments sont dans `Prop` (car une telle proposition est intrinséquement « proof-irrelevant ») (*élimination faible et forte*)
- exception partielle : si le type dans `Prop` a un unique constructeur dont les arguments sont des propositions `Prop` *ou* des petites arités (des schémas de type construisant dans `Prop`), alors l'élimination vers `Set` est autorisée (*élimination faible – vers les petits types – seulement*)

En pratique dans Coq

Pour chaque définition inductive d'un type I, Coq définit automatiquement des schémas d'élimination associés

- élimination forte (vers **Type**) : I_rect
- élimination vers le petit type calculatoire (vers **Set**) : I_rec
- élimination dans les propositions (vers **Prop**) : I_ind

De plus, par défaut, les éliminations sont dépendantes si I est calculatoire (dans **Set** ou **Type**) et non dépendantes si dans **Prop**.

```
Inductive True : Prop := I : True.
True_rect : forall P : Type, P -> True -> P
True_rec : forall P : Set, P -> True -> P
True_ind : forall P : Prop, P -> True -> P

Inductive unit : Type := tt : unit.
unit_rect : forall P : unit -> Type, P tt -> forall u : unit, P u
unit_rec : forall P : unit -> Set, P tt -> forall u : unit, P u
unit_ind : forall P : unit -> Prop, P tt -> forall u : unit, P u
```

Pour engendrer des schémas non engendrés automatiquement, il faut utiliser la commande **Scheme**. Exemple :

```
Scheme True_indd := Induction for True Sort Prop.
```

Inductifs avec dépendances internes

```
Inductive ex (A:Type) (P:A -> Prop) : Prop :=
  ex_intro : forall x:A, P x -> ex (A:=A) P.
```

Peut-on projeter les première et seconde composantes ?

```
Inductive sigT (A:Type) (P:A -> Type) : Type :=
    existT : forall x:A, P x -> sigT P.
```

Peut-on projeter les première et seconde composantes ?

Inductifs d'ordre supérieur : l'exemple des ordinaux récursifs de Kleene

```
Inductive ord : Type :=
| O : ord
| S : ord -> ord
| lim : (nat -> ord) -> ord
```

Schéma d'induction (syntaxe Coq)

```
fun (P : ord -> Type) (f : P O) (f0 : forall o : ord, P o -> P (S o))
  (f1 : forall o : nat -> ord, (forall n : nat, P (o n)) -> P (lim o)) =>
fix F (o : ord) : P o :=
  match o as o0 return (P o0) with
  | O => f
  | S o0 => f0 o0 (F o0)
  | lim o0 => f1 o0 (fun n : nat => F (o0 n))
  end
     : forall P : ord -> Type,
       P O ->
       (forall o : ord, P o -> P (S o)) ->
       (forall o : nat -> ord, (forall n : nat, P (o n)) -> P (lim o)) ->
       forall o : ord, P o
```

Inductifs dépendants : l'exemple de l'égalité

```
Inductive eq (A:Type) (x:A) : A -> Prop :=
  refl_equal : eq A x x.
```

- une famille de types inductifs

$$\frac{}{\Gamma \vdash eq : \forall A : \texttt{Type}.\, A \to A \to \texttt{Prop}}$$

  - les deux premiers paramètres sont des paramètres de famille
  - le troisième paramètre est un paramètre de « prédicat »[1]

- règle d'élimination sans dépendance avec le terme filtré : réécriture !

$$\frac{\Gamma \vdash t : eq\ A\ a\ b \qquad \Gamma, c : A \vdash A(c) : s \qquad \Gamma \vdash u : A(a)}{\Gamma \vdash (\texttt{match}\ t\ \texttt{in}\ eq\ \_\ \_\ c\ \texttt{return}\ A(c)\ \texttt{with}\ refl\_equal\ \texttt{=>}\ u\ \texttt{end}) : A(b)}$$

Remarque : élimination vers toutes les sortes parce que type singleton

[1] terminologie non stabilisée !

Inductifs dépendants : l'exemple de l'égalité

```
Inductive eq (A:Type) (x:A) : A -> Prop :=
  refl_equal : eq A x x.
```

- une famille de types inductifs

$$\frac{}{\Gamma \vdash eq : \forall A : \texttt{Type}.\, A \to A \to \texttt{Prop}}$$

  - les deux premiers paramètres sont des paramètres de famille
  - le troisième paramètre est un paramètre de « prédicat »[2]

- règle d'élimination avec dépendance en le terme filtré

$$\frac{\Gamma \vdash t : eq\ A\ a\ b \qquad \Gamma, c : A, x : eq\ A\ a\ c \vdash A(c,x) : s \qquad \Gamma \vdash u : A(a, refl\_equal\ A\ a)}{\Gamma \vdash (\texttt{match}\ t\ \texttt{as}\ x\ \texttt{in}\ eq\ \_\ \_\ c\ \texttt{return}\ A(c,x)\ \texttt{with}\ refl\_equal\ \texttt{=>}\ u\ \texttt{end}) : A(b,t)}$$

Remarque : élimination vers toutes les sortes parce que type singleton

[2] terminologie non stabilisée !

Inductifs : condition de positivité

Exemple d'inductif contredisant la normalisation

```
Inductive lambda : Type :=
| Lam : (lambda -> lambda) -> lambda
```

De fait, on peut alors définir

$$app \triangleq \lambda x : lambda.\, \lambda y : lambda.\, \texttt{match}\ x\ \texttt{return}\ lambda\ \texttt{with}\ Lam\ f\ \texttt{=>}\ f\ y\ \texttt{end}$$

$$\Delta \triangleq Lam(\lambda x : lambda.\, app\ x\ x)$$
$$\Omega \triangleq app\ \Delta\ \Delta$$

et l'évaluation de Ω boucle.
Un type inductif se définit comme le plus petit type engendré par un ensemble de constructeurs. On peut le voir comme un $\mu_X.\ \oplus_{1\leq i\leq n} \Gamma_i(X)$ (ou μ est un opérateur de point fixe sur les types) et l'existence de ce plus petit type nécessite que opérateur $\lambda X.\ \oplus_{1\leq i\leq n} \Gamma_i(X)$ soit monotone (et donc que X apparaisse uniquement en position positive).
Dans la pratique, on exige une positivité stricte (X n'apparaît pas à gauche d'une implication, même si en occurrence positive). La positivité stricte permet d'éviter de dériver de pouvoir coder le paradoxe de Russell (dans `Type`) et c'est de toutes façons suffisant pour les exemples intéressants.

Inductifs mutuels : l'exemple des forêts d'arbres

```
Inductive tree (A:Type) : Type :=
  | node : A -> (forest A) -> (tree A)
with forest (A:Type) : Type :=
  | empty : (forest A)
  | add : (tree A) -> (forest A) -> (forest A).
```

Simulable par

```
Inductive tree_forest (A:Type) : bool -> Type :=
  | node : A -> (tree_forest A false) -> (tree_forest A true)
  | empty : (tree_forest A false)
  | add : (tree_forest A true) -> (tree_forest A false)
          -> (tree_forest A false).

Definition tree (A:Type) := tree_forest A true.
Definition forest (A:Type) := tree_forest A false.
```

Inductifs mutuels : l'exemple des forêts d'arbres

```
Inductive tree (A:Type) : Type :=
  | node : A -> (forest A) -> (tree A)
with forest (A:Type) : Type :=
  | empty : (forest A)
  | add : (tree A) -> (forest A) -> (forest A).
```

Simulable aussi par

```
Inductive tree_aux (A:Type) (forest:Type): Type :=
  | node : A -> forest -> (tree A forest).

Inductive forest (A:Type) : Type :=
  | empty : (forest A)
  | add : (tree_aux A (forest A)) -> (forest A) -> (forest A).

Definition tree (A:Type) := tree_aux A (forest A).
```

Dans le cas de types inductifs mutuels dans des sortes distinctes, seulement le deuxième codage est possible et il nécessite une condition de positivité stricte imbriquée.

Points-fixes mutuels : exemple de la taille d'une forêt

```
Definition tree_size := fun (A:Type) =>
  fix tree_size (t:tree A) : nat :=
    match t with
    | node A f => S (forest_size f)
    end
  with forest_size (f:forest A) : nat :=
    match f with
    | empty => 0
    | add t f' => tree_size t + forest_size f'
    end
  for tree_size.
```

Point-fixe avec paramètres

Un point-fixe du Calcul des Constructions Inductifs peut avoir plusieurs arguments

```
Inductive vect : nat -> Type :=
| vnil : vect 0
| vcons : forall n, nat -> vect n -> vect (S n).

Definition sum :=
  fix sum (n:nat) (ln:vect n) {struct ln} : nat :=
    match ln return nat with
    | vnil => 0
    | vcons n' p ln' => p + sum n' ln'
    end.
```

On utilise alors la notation `{struct x}` pour indiquer l'argument structurellement décroissant.

Inductifs dépendants : l'exemple de l'accessibilité

```
Inductive Acc (A:Type) (R:A->A->Prop) : A->Prop :=
  Acc_intro : forall x:A, (forall y:A, R y x -> Acc R y) -> Acc R x.
```

$Acc\ A\ R\ x$ exprime que toute chaine descendante à partir de x est bien fondée selon R

$\forall x.\, Acc\ A\ R\ x$ exprime que R est bien fondé dans A.

Décroissance non structurelle

Acc est l'outil pour plonger toute relation d'ordre bien fondée en un ordre structurel. Toute fonction $f(x)$ prouvablement terminante via à un ordre bien fondé $\leq$ peut se définir par

```
fix merge_sort (l:list nat) (H:Acc le (length l)) {struct H} : list nat :=
  match H with Acc n Hn =>
  ... merge_sort l1 (Hn (length l1) (* proof of length l1 < length l *) ...
  ... merge_sort l2 (Hn (length l2) (* proof of length l1 < length l *) ...
  end.
```

Coq has a macro for doing that : `Function`.

```
Definition R (l1 l2:list nat) := length l1 < length l2.

Function merge_sort (l:list nat) {wf R l} : list nat :=
  match H with Acc n Hn =>
  ... merge_sort l1 (Hn (length l1) (* proof of length l1 < length l *) ...
  ... merge_sort l2 (Hn (length l2) (* proof of length l1 < length l *) ...
  end.
```

Décroissance non structurelle

Acc est l'outil pour plonger toute relation d'ordre bien fondée en un ordre structurel. Toute fonction $f(x)$ prouvablement terminante via à un ordre bien fondé $\leq$ peut se définir par

```
fix merge_sort (l:list nat) (H:Acc le (length l)) {struct H} : list nat :=
  match H with Acc n Hn =>
  ... merge_sort l1 (Hn (length l1) (* proof of length l1 < length l *) ...
  ... merge_sort l2 (Hn (length l2) (* proof of length l1 < length l *) ...
  end.
```

Coq has a macro for doing that : `Function`.

```
Function merge_sort (l:list nat) {measure length l} : list nat :=
  match H with Acc n Hn =>
  ... merge_sort l1 (Hn (length l1) (* proof of length l1 < length l *) ...
  ... merge_sort l2 (Hn (length l2) (* proof of length l1 < length l *) ...
  end.
```

Paramètres récursivement non uniformes

Coq 8.1 autorise des paramètres récursivement non uniformes. Ainsi on peut réécrire Acc ainsi

```
Inductive Acc (A:Type) (R:A->A->Prop) (x:A) : Prop :=
  Acc_intro : (forall y:A, R y x -> Acc R y) -> Acc R x.
```

Inductifs dépendants : exemple

```
Inductive prove : list formula -> formula -> Prop :=
| ProofImplyE : forall A B Gamma, prove Gamma (A --> B) -> prove Gamma A -> prove Gamma B
| ProofImplyI : forall A B Gamma, prove (A::Gamma) B -> prove Gamma (A --> B)
| ProofAx : forall A Gamma C, In A Gamma -> Gamma |- A

where "Gamma |- A" := (prove Gamma A).
```

Inversion

Principe d'inversion

```
prove Gamma C ->
(exists A, exists B, C=A-->B /\ prove (A::Gamma) B) \/
(exists A, prove Gamma (A --> B) /\ prove Gamma A) \/
(In C Gamma)
```

Types coinductifs

```
CoInductive Stream :  Set := Cons : A -> Stream -> Stream.
```

```
CoFixpoint zeros  : Stream nat := Cons 0 zeros.
CoFixpoint from (n:nat) : Stream nat := Cons n (from (S n)).
```

Condition de garde : appels récursifs protégés par un constructeur.