Assistants de Preuve

Bruno Barras et Christine Paulin-Mohring

Université Paris-Sud 11, INRIA

01/12/08

Calcul des Constructions Inductives 2

Plan

Types inductifs récursifs : l'exemple des entiers

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

$$
\begin{array}{l}
\lambda P : \texttt{nat} \to s, \\
\lambda H_O : P(O), \\
\lambda H_S : \forall m : \texttt{nat}, P(S\ m), \\
\lambda n : \texttt{nat}, \\
\quad \texttt{match}\ n\ \texttt{as}\ y\ \texttt{return}\ P(y)\ \texttt{with} \\
\qquad O \Rightarrow H_O \mid S\ m \Rightarrow H_S\ m \\
\quad \texttt{end}
\end{array}
$$

est une preuve de

$$\forall P : \texttt{nat} \to s, P(O) \to (\forall m : \texttt{nat}, P(S\ m)) \to \forall n : \texttt{nat}, P(n)$$

Comment dériver le schéma de récursion standard ?

L'opérateur de point-fixe (première étape)

Ajout d'une expression de point fixe anonyme typée

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

. . . 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 (réduction)

Expression de point fixe anonyme à résultat dépendant

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

- Typage

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

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

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

L'opérateur de point-fixe : application

de la construction par cas au récurseur sur les entiers

la construction par cas

$$\begin{array}{l}\lambda P : \texttt{nat} \to s,\\ \lambda H_O : P(O),\\ \lambda H_S : \forall m : \texttt{nat}, P(S\ m),\\ \lambda n : \texttt{nat},\\ \quad \texttt{match}\ n\ \texttt{as}\ y\ \texttt{return}\ P(y)\ \texttt{with}\\ \qquad O \Rightarrow H_O \mid S\ m \Rightarrow H_S\ m\\ \quad \texttt{end}\end{array}$$

est de type

$$\begin{array}{l}\forall P : \texttt{nat} \to s,\\ P(O) \to\\ (\forall m : \texttt{nat}, P(S\ m)) \to\\ \forall n : \texttt{nat}, P(n)\end{array}$$

le récurseur

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

est de type

$$\begin{array}{l}\forall P : \texttt{nat} \to s,\\ P(O) \to\\ (\forall m : \texttt{nat}, P(m) \to P(S\ m)) \to\\ \forall n : \texttt{nat}, P(n)\end{array}$$

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

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

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

$$\frac{I \text{ inductif } \quad \Gamma \vdash I : s \quad \Gamma, x : A \vdash C : s \quad \Gamma, x : I, f : (\forall x : I, C) \vdash t : C \quad t|_f^{\emptyset} <_I x}{\Gamma \vdash (\texttt{fix}\ f(x : I) : C := t) : \forall x : I, C}$$

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

$$\frac{z \in \rho \cup \{x\} \quad (u_i|_f^{\rho} <_I x)_{i=1\ldots n} \quad A|_f^{\rho} <_I x \quad (t_i|_f^{\rho \cup \{x \in \vec{x_i} | x : \forall y : \vec{U}. I\, \vec{u}\}} <_I x)_i}{\texttt{match}\ z\ u_1 \ldots u_n\ \texttt{return}\ A\ \texttt{with} \ldots\ c_i\ \vec{x_i} \Rightarrow t_i\ \ldots \texttt{end}|_f^{\rho} <_I x}$$

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

$$\frac{y \in \rho}{f\ y|_f^{\rho} <_I x} \qquad \frac{f \notin t}{t|_f^{\rho} <_I x}$$

+ règles contextuelles . . .

Remarques sur le critère

- Couvre simplement le schéma de définition primitif récursif/ preuve par récurrence
  Appel récursif sur tous les sous-termes récursifs immédiats :

$$
\begin{array}{l}
\lambda P : \texttt{list}\ A \to s, \\
\lambda f_1 : P\ \texttt{nil}, \\
\lambda f_2 : \forall (a : A)(l : \texttt{list}\ A), P\ l \to P\ (\texttt{cons}\ a\ l), \\
\texttt{fix}\ Rec(x : \texttt{list}\ A) : P\ x := \\
\quad \texttt{match}\ x\ \texttt{return}\ P\ x\ \texttt{with} \\
\qquad \texttt{nil} \Rightarrow f_1 \mid (\texttt{cons}\ a\ l) \Rightarrow f_2\ a\ l\ (Rec\ l) \\
\quad \texttt{end}
\end{array}
$$

- de type

$$
\begin{array}{l}
\forall P : \texttt{list}\ A \to s, \\
P\ \texttt{nil}, \to \\
(\forall (a : A)(l : \texttt{list}\ A), P\ l \to P\ (\texttt{cons}\ a\ l)) \to \\
\forall (x : \texttt{list}\ A), P\ x
\end{array}
$$

Remarques sur le critère

Possibilité d'appel récursif sur des sous-termes profonds

```
Fixpoint mod2 (n:nat) : nat :=
   match n with O ⇒ O | S O ⇒ S O
               | S (S x) ⇒ mod2 x
   end
```

Possibilité aussi d'appel récursif sur des termes construits par cas si chaque branche est un sous-terme strict.

```
Fixpoint F (n:nat) : C :=
  match iszero n with
    (left (H:n=O)) ⇒ ...
  | (right (H:n<>0)) ⇒
      F (match n return n<>0→C with
           S p ⇒ (fun h ⇒ p)
         | _   ⇒ (fun h ⇒
                    match h (refl_equal 0) return C
                    with end
                 H)
  end
```

Remarques sur le critère

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

```
Inductive Singl (A:Prop) : Prop := c : A → Singl A.
Definition T := ∀(A:Prop), A → A.
Definition t : T := fun A x ⇒ x.
Fixpoint f (x : Singl T) : bool :=
     match x with (c a) ⇒ f (a (Singl T) (c T t)) end.
```

$$f\,(c\,T\,t) \longrightarrow f\,(t\,(Singl\,T)\,(c\,T\,t)) \longrightarrow f\,(c\,T\,t)$$

Un peu de terminologie

- Le Calcul des Constructions Inductives prédicative a les sortes **Prop**, **Set** $=$ $\mathbf{Type}_0$, $\mathbf{Type}_1$, $\mathbf{Type}_2$, ...
- **Prop** et **Set** sont dites petites (parce qu'elles ne typent aucune autre sorte)
- Les sortes $\mathbf{Type}_i$ (pour $i \geq 1$) sont dites grandes (parce qu'elles typent **Prop** et **Set**)

Inductifs : condition de positivité

Condition de positivité sticte. L'argument récursif d'un constructeur a pour type $\forall (z_1 : C_1) \ldots (z_k : C_k) I\, t_1 \ldots t_n$.
Exemple d'inductif non monotone contredisant la normalisation :

```
Inductive lambda : Type :=
| Lam : (lambda → lambda) → lambda
```

De fait, on peut alors définir

```
Definition app (x y:lambda)
   := match x with (Lam f) ⇒ f y end.
Definition Delta := Lam (fun x ⇒ app x x).
Definition Omega := app Delta Delta.
```

et l'évaluation de Ω boucle.

Inductifs : condition de positivité

- 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 pouvoir coder le paradoxe de Russell (dans **Type**) et c'est de toutes façons suffisant pour les exemples intéressants.

Inductifs : condition de positivité stricte

La monotonie est suffisante au niveau imprédicatif :

$$\mu F := \forall (X : \textbf{Prop}), (F\,X \to X) \to X$$

Mais problématique au niveau **Type**.

```
Inductive X : Type := inj : ((X → Prop) → Prop) → X.
```

$$\begin{array}{lcl} P_0 & \triangleq & \lambda x : X, \exists P', x = in(\lambda (P : X \to \textbf{Prop}), P = P') \wedge \neg P'(x) \\ x_0 & \triangleq & inj(\lambda (P : X \to \textbf{Prop}), P = P_0) \end{array}$$

$$\begin{array}{lcl} P_0(x_0) & \leftrightarrow & \exists P', x_0 = inj(\lambda P.P = P') \wedge \neg P'(x_0) \\ & \leftrightarrow & \exists P', inj(\lambda P, P = P_0) = inj(\lambda P.P = P') \wedge \neg P'(x_0) \\ & \leftrightarrow & \exists P', P' = P_0 \wedge \neg P'(x_0) \\ & \leftrightarrow & \exists P', P' = P_0 \wedge \neg P_0(x_0) \\ & \leftrightarrow & \neg P_0(x_0) \end{array}$$

Conditions sur les sortes des inductifs

- arité et sorte de l'inductif $I : \forall (x_1 : A_1) \ldots (x_n : A_n) s$
- un constructeur est de la forme
  $c : \forall (y_1 : B_1) \ldots (y_p : B_p) I\, u_1 \ldots u_n$
- condition de typage

$$I : (x_1 : A_1) \ldots (x_n : A_n) s \vdash \forall (y_1 : B_1) \ldots (y_p : B_p) I\, u_1 \ldots u_n : s$$

- La sorte d'un inductif prédicatif (dans la hiérarchie **Type**) est le max des sortes des types des arguments de ses constructeurs.
- La sorte d'un inductif imprédicatif (de type **Prop**) n'est pas contrainte.

```
Inductive PB : Prop := in : Prop → Pb.
```

Potentiellement problématique car PB : **Prop** mais PB intuitivement isomorphe à **Prop**.

Restrictions d'élimination selon la sorte

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

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

Règle d'élimination du type *or A B* (seulement vers **Prop**)

$$\frac{\begin{array}{cc}\Gamma \vdash t : \mathit{or}\ A\ B & \Gamma, p : A \vdash t_1 : C(\texttt{or\_introl}\ p) \\ \Gamma, x : \mathit{or}\ A\ B \vdash C(x) : \mathit{Prop} & \Gamma, q : B \vdash t_2 : C(\texttt{or\_intror}\ q)\end{array}}{\Gamma \vdash \left(\begin{array}{l}\texttt{match}\ t\ \texttt{as}\ x\ \texttt{return}\ C(x)\ \texttt{with} \\ \quad \texttt{or\_introl}\ p \Rightarrow t_1 \mid \texttt{or\_intror}\ q \Rightarrow t_2 \\ \texttt{end}\end{array}\right) : C(t)}$$

Règles sur les sortes d'élimination

- 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*)

```
fun (p:or A B) ⇒ match p with
      (or_introl a) ⇒ true | (or_intror b) ⇒ false
                    end.
```

  - exception : si le type dans **Prop** a zéro constructeur (absurde) ou un unique constructeur dont les arguments sont dans **Prop** (*élimination faible et forte*)
    Cas de l'égalité, des conjonctions . . .
  - 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**.

Exemples

```
Inductive True : Prop := I : True.
True_rect : ∀P : Type, P → True → P
True_rec : ∀P : Set, P → True → P
True_ind : ∀P : Prop, P → True → P

Inductive unit : Type := tt : unit.
unit_rect : ∀P : unit → Type, P tt → ∀u : unit, P u
unit_rec : ∀P : unit → Set, P tt → ∀u : unit, P u
unit_ind : ∀P : unit → Prop, P tt → ∀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.
True_indd
     : ∀P : True → Prop, P I → ∀t : True, P t
```

Inductifs avec dépendances internes

```
Inductive ex (A:Type) (P:A → Prop) : Prop :=
  ex_intro : ∀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 : ∀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:∀o : ord, P o → P (S o))
    (f1 : ∀o:nat→ord, (∀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
 : ∀P : ord → Type,
    P O → (∀o:ord, P o→P (S o)) →
    (∀o:nat→ord,(∀n:nat,P (o n)) → P (lim o)) →
    ∀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

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

  - les deux premiers paramètres sont des paramètres de famille
  - le troisième paramètre est un "index"
- règle d'élimination sans dépendance avec le terme filtré : réécriture !

$$\frac{\Gamma \vdash t : eq\, A\, a\, b \quad \Gamma, c : A \vdash A(c) : s \quad \Gamma \vdash u : A(a)}{\Gamma \vdash \left(\begin{array}{l} \texttt{match}\ t\ \texttt{in}\ eq\ \_\ \_\ c\ \texttt{return}\ A(c)\ \texttt{with} \\ \quad refl\_equal \Rightarrow u \\ \texttt{end} \end{array} \right) : A(b)}$$

Remarque : élimination vers toutes les sortes parce que type singleton

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_for (A:Type) : bool → Type :=
  | node : A → tree_for A false → tree_for A true
  | empty : tree_for A false
  | add : tree_for A true → tree_for A false
          → tree_for A false.

Definition tree (A:Type) := tree_for A true.
Definition forest (A:Type) := tree_for 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 O
| vcons : ∀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 ⇒ O
    | 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 : ∀x:A, (∀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 msort (l:list nat)(H:Acc le (length l)){struct H}
 : list nat :=
  match H with Acc n Hn ⇒
  ..msort l1 (Hn (length l1) (* proof of |l1|<|l| *)..
  ..msort l2 (Hn (length l2) (* proof of |l1|<|l| *)..
  end.
```

En fait il faut écrire

```
msort l1
    (match H with
      Acc n Hn ⇒ Hn (length l1) (* proof of |l1|<|l| *)
     end)
```

Décroissance non structurelle

Coq a une macro pour faire cela : `Function`.

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

Function msort (l:list nat) {wf R l} : list nat :=
  match H with Acc n Hn ⇒
  ..msort l1 (Hn (length l1) (* proof of |l1|<|l| *)..
  ..msort l2 (Hn (length l2) (* proof of |l1|<|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 : (∀y:A, R y x → Acc R y) → Acc R x.
```

Inductifs dépendants : exemple

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

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

équivalent à

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

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

Inversion

Principe d'inversion

```
prove Gamma C →
(∃A, ∃B, C=A-→B ∧ prove (A::Gamma) B) ∨
(∃A, prove Gamma (A -→ B) ∧ prove Gamma A) ∨
(In C Gamma)
```

Gratuit si on choisit la formulation complètement paramétrée.

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.