Le Calcul des Constructions Inductives

Joëlle Despeyroux
INRIA Sophia-Antipolis

Module "Mécanisation des Preuves"

Dea d'informatique, Nice, janvier-mars 2000

Les Systèmes de Types Purs (PTS)

$termes \quad M \quad ::= x \mid c \mid \lambda x : M.M' \mid (M \;\; M') \mid \Pi x : M.M'$

Un PTS est défini par la donnée d'un triplet $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ où $\mathcal{S} \subset \mathcal{C}$, $\mathcal{A}$ est un ens d'axiomes $(c : s)$ où $c \in \mathcal{C}$ et $s \in \mathcal{S}$, et $\mathcal{R}$ est un ens de triplets (s_1, s_2, s_3) avec $s_i \in \mathcal{S}$.

$$ax \;\; \vdash c : s \quad (c : s) \in \mathcal{A} \qquad var \; \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \; x \notin \Gamma$$

$$aff \; \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B} \; x \notin \Gamma$$

$$lam \; \frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash \lambda x : A.M \;:\; \Pi x : A.B}$$

$$app \; \frac{\Gamma \vdash M : \Pi x : A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash (M \;\; N) : B[N/x]}$$

$$prod \; \frac{\Gamma \vdash A \;:\; s_1 \qquad \Gamma \vdash B \;:\; s_2}{\Gamma \vdash \Pi x : A.B \;:\; s_3} \; (s_1, s_2, s_3) \in \mathcal{R}$$

$$conv \; \frac{\Gamma \vdash A \;:\; B \qquad \Gamma \vdash B' \;:\; s \qquad B =_\beta B'}{\Gamma \vdash A \;:\; B'}$$

Le λ-cube est un sous-ens des PTS, où $(s_1, s_2) = (s_1, s_2, s_2)$.

CCind

PTS:

Sortes: $\mathcal{S} = \{\mathrm{Prop}, \mathrm{Type}, \mathrm{Set}, \mathrm{TypeSet}\}$.

Axiomes: $\mathcal{A} = \{\mathrm{Prop} : \mathrm{Type}, \mathrm{Set} : \mathrm{TypeSet}\}$.

Règles: $\mathcal{R} = \mathcal{S} \times \mathcal{S}$.

+ **types inductifs** (constantes, règles de typage et de réduction)

Notations:

Dans Coq, l'abstraction $\lambda x : t.t'$ est notée: $[x : t]\ t'$, et
le produit $\Pi x : t.t'$ ou $\forall x : t.t'$ est noté: $(x : t)\ t'$.

Les entiers dans CCind

```
Inductive nat: Set := O: nat | S: nat->nat.
```

À partir de la définition de *nat*, Coq génère:

$$
\begin{array}{l}
nat\_ind : \forall P : nat \rightarrow \mathrm{Prop}. \\
\quad (P\ 0) \rightarrow \\
\quad (\forall m : nat.(P\ m) \rightarrow (P\ (S\ m))) \rightarrow \\
\quad \forall n : nat.(P\ n)
\end{array}
$$

Récursion sur les entiers

```
Recursive Definition plus: nat -> nat -> nat :=
      O   m => m
  | (S p) m => (S (plus p m)).

Fixpoint plus [n:nat]: nat -> nat :=
  [m:nat]<nat>Case n of
     (* O *)     m
     (* (S p) *) [p:nat] (S (plus p m)) end.

Fixpoint even [n:nat] : bool :=
  <nat>Case n of
     (* O *)     true
     (* (S p) *) [p:nat]<nat>Case p of
         (* O *)     false
         (* (S m) *) [m:nat](even m) end end.
```

Dans les versions précédentes de Coq:

```
Old_Definition plus: nat -> nat -> nat :=
  [n,m:nat]<nat>Match n with
     (* O *) m
     (* (S p) *) [p,r:nat] (S r) end.
```

Récursion sur les entiers (suite)

À partir de la définition de *nat*, Coq génère *nat_ind* et *nat_rec* :

$\forall P : nat \rightarrow \text{Set}.\ (P\ 0) \rightarrow (\forall m : nat.(P\ m) \rightarrow (P\ (S\ m))) \rightarrow \forall n : nat.(P\ n)$

avec: $\begin{array}{ll} (nat\_rec\ P\ x\ f\ 0) & \hookrightarrow x \\ (nat\_rec\ P\ x\ f\ (S\ n)) & \hookrightarrow (f\ n\ (nat\_rec\ P\ x\ f\ n)) \end{array}$

```
Definition nat_rec' := [A:Set](nat_rec [n:nat]A).
```

$nat\_rec' : \ \forall A : \text{Set}.\ A \rightarrow (nat \rightarrow A \rightarrow A) \rightarrow nat \rightarrow A$ (Gödel)

- *nat_rec*: élimination dépendante ($P = [n : nat]A$ dépend de n)
- *nat_rec'*: élimination non dépendante ($P = A : Set$)

```
Definition plus:=
  [n,m:nat](nat_rec' nat m [p,r:nat](S r) n)

Old_Definition plus: nat -> nat -> nat :=
  [n,m:nat]<nat>Match n with
     (* O *) m
     (* (S p) *) [p,r:nat] (S r) end.
```

Évaluation de plus:

$(plus\ x\ 0) = (nat\_rec'\ nat\ x\ f\ 0) \hookrightarrow x$

$(plus\ x\ (S\ n)) = (nat\_rec'\ nat\ x\ f\ (S\ n))$

$\hookrightarrow (f\ n\ (nat\_rec'\ nat\ x\ f\ n) = S\ (nat\_rec'\ nat\ x\ f\ n)$

$= S\ (plus\ x\ n))$

λT: le système de Gödel

λ-calcul avec types munis d'un opérateur de récursion.

Exemple: nat avec 0, S et R_a pour chaque const de type a.

$$
\begin{array}{ll}
0 & : nat \\
S & : nat \rightarrow nat \\
R_a & : a \rightarrow (nat \rightarrow a \rightarrow a) \rightarrow nat \rightarrow a
\end{array}
$$

Extension des règles de réduction:

$$(R_a\ M\ N\ 0) \hookrightarrow M$$
$$(R_a\ M\ N\ (S\ x)) \hookrightarrow (N\ x\ (R_a\ M\ N\ x))$$

Exemple:

$$plus = \lambda x, y : nat.(R_{nat}\ y\ \lambda p, r_p : nat.(S\ r_p)\ x)$$

Dans CCind fix et $case$ (ou nat_rec') implémentent R_{nat}:

```
Fixpoint plus [x:nat]: nat -> nat :=
  [y:nat]<nat>Case x of
     (* O *)      y
     (* (S p) *) [p:nat] (S (plus p y)) end.
```

Les listes

```
Inductive list: Set :=
    nil: list
  | cons: A -> list -> list.
```

$list\_ind :$
$\forall P : list \to \text{Prop}.\ (P\ nil) \to$
$(\forall a : A.\forall l : list.(P\ l) \to (P\ (cons\ a\ l))) \to$
$\forall l : list.(P\ l)$

```
Inductive list [A:Set]: Set :=
    nil: (list A)
  | cons: A -> (list A) -> (list A).
```

$list\_ind :$
$\forall A : Set.\forall P : (list\ A) \to \text{Prop}.\ (P\ nil\ A) \to$
$(\forall a : A.\forall l : (list\ A).(P\ l) \to (P\ (cons\ A\ a\ l))) \to$
$\forall l : (list\ A).(P\ l)$

```
Inductive list [A:Set]: nat -> Set :=
    nil: (list A O)
  | cons: (n:nat) A -> (list A n) -> (list A (S n)).
```

$list\_ind :$
$(A : Set, P : ((n : nat)(list\ A\ n) \to Prop))$
$(P\ 0\ (nil\ A)) \to$
$((n : nat, y : A, l : (list\ A\ n))(P\ n\ l) \to (P\ (S\ n)\ (cons\ A\ n\ y\ l))) \to$
$(n : nat, l : (list\ A\ n))(P\ n\ l)$

Le calcul des predicats

```
Inductive True : Prop := I:True.
Inductive False: Prop := .

   False_ind: (P:Prop) False->P

Inductive and [A:Prop;B:Prop]: Prop
  := conj: A->B->A/\B.

   and_ind: (A,B,P:Prop) (A->B->P)->A/\B->P

Inductive or [A:Prop;B:Prop]: Prop
  := or_il: A-> A\/B | or_ir: B->A\/B.

   or_ind : (A,B,P:Prop) (A->P)->(B->P)->A\/B->P

Inductive exists [A:Set;P:A->Prop]: Prop
  := ex: (x:A) (P x)->(exists A P).

   exists_ind: (A:Set)(P:A->Prop)(Q:Prop)
      ((x:A)(P x)->Q)->(exists A P)->Q
```

Les définitions inductives formalisent les règles d'introduction.

Les principes d'induction formalisent les règles d'élimination dans le calcul des séquents.

Les termes de preuve

```
and_ind = [A,B,P:Prop; f:(A->B->P); a:(A/\B)]
            Cases a of (conj x x0) => (f x x0) end
        : (A,B,P:Prop) (A->B->P) -> A/\B -> P

or_ind = [A,B,P:Prop; f:(A->P); f0:(B->P); o:(A\/B)]
           Cases o of
             (or_il x) => (f x)
           | (or_ir x) => (f0 x)
           end
       : (A,B,P:Prop) (A->P) -> (B->P) -> A\/B -> P
```

Règles de la déduction naturelle:

```
Theorem and_elim1: (A,B:Prop) (A /\ B ) -> A.

Theorem or_elim :
   (A,B:Prop) (A\/B) -> (C:Prop) (A->C) -> (B->C) -> C.

nat_ind
  = [P:(nat->Prop); f:(P O); f0:((n:nat)(P n)->(P (S n)))]
    Fix F {F [n:nat] : (P n) :=
        Cases n of
          O => f
        | (S n0) => (f0 n0 (F n0))
        end}
  : (P:(nat->Prop))(P O)->((n:nat)(P n)->(P (S n)))->
    (n:nat)(P n).
```

Les prédicats inductifs

```
Inductive even: nat -> Prop :=
    even0: (even 0)
  | evenS: (n:nat) (even n) -> (even (S (S n))).

even_ind: (P:nat->Prop)
   (P 0) ->
   ((n:nat) (even n)->(P n)->(P (S (S n))))->
   (n:nat) (even n)->(P n)

Inductive le [n:nat]: nat -> Prop :=
    len: (le n n)
  | leS: (m:nat) (le n m) -> (le n (S m)).

le_ind: (n:nat)(P:nat->Prop)
   (P n) ->
   ((m:nat) (le n m)->(P m)->(P (S m)))->
   (x:nat) (le n x)->(P x)
```

Égalité de Leibnitz:

```
Inductive eq [A:Set;x:A]: A->Prop :=
  refl: (eq A x x).

eq_ind: (A:Set)(x:A)(P:A->Prop)
  (P x)->(y:A)(eq A x y)->(P y)
```

Types inductifs dans CCind

Types inductifs:

$$\mathrm{Ind}(X : A)\{T_1(X), \ldots, T_n(X)\}$$

Exemples:

```
Inductive nat: Set := 0: nat | S: nat->nat.
```

$$nat = \mathrm{Ind}(X : \mathrm{Set})\{X,\ X \to X\} : \mathrm{Set}$$

```
Inductive list [A:Set]: Set :=
    nil: (list A)
  | cons: A -> (list A) -> (list A).
```

$$list = \lambda A : \mathrm{Set}.\mathrm{Ind}(X : \mathrm{Set})\{X,\ A \to X \to X\} : \mathrm{Set} \to \mathrm{Set}$$

```
Inductive even: nat -> Prop :=
    even0: (even 0)
  | evenS: (n:nat) (even n) -> (even (S (S n))).
```

$$\begin{aligned} even = {} & \mathrm{Ind}(X : nat \to \mathrm{Prop}) \\ & \{(X\ 0),\ \forall n : nat.(X\ n) \to (X\ (S\ (S\ n)))\} \\ & : nat \to \mathrm{Prop} \end{aligned}$$

```
Inductive eq [A:Set;x:A]: A->Prop :=
  refl: (eq A x x).
```

$$\begin{aligned} eq = {} & \lambda A : \mathrm{Set}.\lambda x : A.\mathrm{Ind}(X : A \to \mathrm{Prop})\{(X\ x)\} \\ & : \forall A : \mathrm{Set}.A \to A \to \mathrm{Prop} \end{aligned}$$

CCind

Types inductifs:

$$\mathrm{Ind}(X : A)\{T_1(X), \ldots, T_n(X)\}$$

Constructeurs (intro):

$$\mathrm{Constr}(i, I)$$

Exemples: $\begin{array}{ll} 0 & = \mathrm{Constr}(1, nat) : nat \\ S & = \mathrm{Constr}(2, nat) : nat \to nat \end{array}$

Opérateurs de récursion (élim):

$$[Q]\mathrm{Case}\ m : t \text{ of } \{\vec{f}\} \quad \text{et}$$

$$\mathrm{Fix}\ f : t := m$$

Exemple:

```
Fixpoint plus [n:nat]: nat -> nat :=
  [m:nat]<nat>Case n of
     (* O *)      m
     (* (S p) *) [p:nat] (S (plus p m)) end.
```

PTS:

Sortes: $\mathcal{S} = \{\mathrm{Prop}, \mathrm{Type}, \mathrm{Set}, \mathrm{TypeSet}\}$.

Axiomes: $\mathcal{A} = \{\mathrm{Prop} : \mathrm{Type}, \mathrm{Set} : \mathrm{TypeSet}\}$.

Règles: $\mathcal{R} = \mathcal{S} \times \mathcal{S}$.

Termes $t ::= x \mid c \mid \lambda x : t.t' \mid (t\ t') \mid \forall x : t.t' \mid (\mathrm{Co})\mathrm{Ind}(x : t)\{\vec{t}\}$
$\mid \mathrm{Constr}(n, t) \mid [Q]\mathrm{Case}\ m : t \text{ of } \{\vec{f}\} \mid (\mathrm{Co})\mathrm{Fix}\ f : t := m$

INRIA

CCind: Règles de réduction

$$[Q]\text{Case } (Constr(i,I)\ \vec{t}) : Q \text{ of } \{\vec{f}\} \hookrightarrow (f_i\ \vec{t})$$

$$\text{Soit } F = (\text{Fix } f : t := M)$$
$$(F\ \vec{u}\ (Constr(i,I)\ \vec{v})) \hookrightarrow (M\{F/f\}\ \vec{u}\ (Constr(i,I)\ \vec{v}))$$

Exemple:

```
Fixpoint plus [n:nat]: nat -> nat :=
  [m:nat]<nat>Case n of
     (* O *)      m
     (* (S p) *) [p:nat] (S (plus p m)) end.
```

$F = \lambda n : nat.\ (Fix\ plus : nat \rightarrow nat := M)$
où $M = \lambda m : nat.\ Case(n, m)$

$Case(0, m) \hookrightarrow m$
$Case(S(p), m) \hookrightarrow (\lambda p : nat.\ (S\ (plus\ p\ m))\ \ p) = (S\ (plus\ p\ m))$

$plus(0, m) \hookrightarrow m$
$plus(S(p), m) \hookrightarrow (S\ (plus\ p\ m))$

CCind - notations

Def. *(Occurence positive)* $t = \forall \vec{x} : \vec{t}.(X\ \vec{t'})$ est strictement positif en X $[Pos(X,t)]$ si $X \notin FV(\vec{t}, \vec{t'})$.

Def. *(Forme de constructeur)* Un terme C est une forme de constructeur en X $[Construc_X(C)]$ ssi

$C = (X\ \vec{t})$ avec $X \notin FV(\vec{t})$, ou

$C = \forall x : t.C'$ avec $Construc_X(C')$ et $X \notin FV(t)$, ou

$C = P \to C'$ avec $Construc_X(C')$ et $Pos(X, P)$.

Exemples:

$Construc_X(A \to (A \to X) \to X)$

$\neg Construc_X((X \to A) \to X)$ et $\neg Construc_X((X \to X) \to X)$

Def. *(petit constructeur)* Soit $(C\ X)$ une forme de constructeur. $(C\ X) = \forall \vec{x} : \vec{t}.(X\ \vec{t'})$. $(C\ X)$ est dit petit si $\forall i\ t_i$: Prop ou Set.

CCind: Typage

$$\forall i\ Construc_X(C_i(X))$$

$$\frac{A = \forall \vec{x} : \vec{A}.s \quad \forall i\ (\Gamma, (X : A) \vdash C_i(X) : s)}{\Gamma \vdash \mathrm{Ind}(X : A)\{\overrightarrow{C_i(X)}\} : A} \quad s \in \{\mathrm{Set}, \mathrm{Prop}\}$$

$$\frac{I = \mathrm{Ind}(X : A)\{\overrightarrow{C_i(X)}\} \qquad \Gamma \vdash I : A}{\Gamma \vdash \mathrm{Constr}(n, I) : C_n(I)}$$

Exemples:

```
Inductive nat: Set := O: nat | S: nat->nat.

Inductive list [A:Set]: nat -> Set :=
    nil: (list A O)
  | cons: (n:nat) A -> (list A n) -> (list A (S n)).
```

CCind: Typage (suite)

Def. $C(X)$ une forme de constructeur en X

$$\mathcal{S}\{(X\ \vec{t}), Q, c\} = (Q\ \vec{t}\ c)$$

$$\mathcal{S}\{\forall x : t.(C\ X), Q, c\} = \forall x : t.\mathcal{S}\{(C\ X), Q, (c\ x)\}$$

$$\mathcal{S}\{(P\ X) \to (C\ X), Q, c\} = \forall y : (P\ X).\mathcal{S}\{(C\ X), Q, (c\ y)\}$$

Exemples:

$\mathcal{S}\{nat, Q, 0\} = (Q\ 0)$

$\mathcal{S}\{nat, Q, S\} = \forall n : nat.(Q\ (S\ n))$

$$A = \forall \vec{x} : \vec{A}.s \quad I = \mathrm{Ind}(X : A)\{\overrightarrow{C_i(X)}\} \quad \Gamma \vdash t : (I\ \vec{u})$$

$$\frac{\Gamma \vdash Q : \forall \vec{x} : \vec{A}.(I\ \vec{x}) \to s' \quad \Gamma \vdash f_i : \mathcal{S}\{(C_i\ I), Q, Constr(i, I)\}}{\Gamma \vdash [Q]\mathrm{Case}\ t : (I\ \vec{u})\ \mathrm{of}\ \{\vec{f}\} : (Q\ \vec{u}\ t)}\ s, s' \in \mathcal{S}$$

$$t = \forall \vec{x} : \vec{t}.\forall x : (I\ \vec{x}).t' \quad m = \lambda \vec{x} : \vec{t}.\lambda x : (I\ \vec{x}).m'$$

$$\frac{I = \mathrm{Ind}(X : A)\{\vec{C}\} \quad \Gamma \vdash t : s \quad \Gamma, f : t \vdash m : t \quad \mathcal{D}\{f, |\vec{x}|, x, m'\}}{\Gamma \vdash (\mathrm{Fix}\ f : t := m) : t}$$

```
Fixpoint plus [n:nat]: nat -> nat :=
  [m:nat]<nat>Case n of
     (* O *)      m
     (* (S p) *)  [p:nat] (S (plus p m)) end.
```

Le système Coq

Système de développement de preuves basé sur CCind

Eliminations autorisées:

s / s'	Prop	Set	Type	TypeSet
Prop	○			
Set	○	○	●	●
Type	○	○	○	
TypeSet	○	○	○	○

○ élimination autorisée.

● élimination dépendante autorisée sur les petits types:
élimination forte.

Set/Type → prouver $0 \neq (S\ x)$.

Set/TypeSet → définir le type LL, alternative à $(list\ n\ L)$:

```
Fixpoint LL [L:Set] : nat -> Set :=
  [m:nat]<Set>Case m of
     (* O *)      L
     (* (S n) *) [n:nat] L -> (LL n) end.
```

Tactiques:
apply, intro, elim, exists, rewrite, induction, auto, try, ...

Extraction de programmes /
développement de programmes certifiés.

Bibliographie

References

[Gim95] Eduardo Gimenez. Codifying guarded definitions with recursive schemes. In P. Dybjer, B. Nordström, and J. Smith, editors, *Proceedings of the Workshop on Types in Proofs and Programs, Bastad, Sweden, June 1994*, Springer-Verlag LNCS 996, 1995. also available as ENS Lyon Report 95-07, dec 94.

[HKPM97] G. Huet, G. Kahn, and Ch. Paulin-Mohring. The Coq proof assistant - a tutorial, version 6.1. Technical Report 204, INRIA, Août 1997. Version révisée distribuée avec Coq.

[PW99] Christine Paulin and Benjamin Werner. Introduction to Coq. Lectures notes, DEA Paris University and Types summer school'99, August 1999.

[Wer94] Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris VII, Mai 1994.