M2 PLS. Introduction à Coq

Micaela Mayero

http://www-lipn.univ-paris13.fr/~mayero/
Université Paris 13
LIPN-LCR

28 novembre 2014

Assistant d'aide à la preuve

- Autres prouveurs : HOL, PVS, Mizar, Isabelle, B, ACL2, …*[Freek]*
- Pourquoi faire ? :
  - preuves de correction de "propriétés" / spécification :
    - ↪ propriétés mathématiques (théorèmes)
    - ↪ preuves de programmes
    - ↪ spécifications
  - exemples significatifs :
    - ↪ Javacard, compilateur C certifié
    - ↪ Meteor, NASA
    - ↪ Bibliothèque mathématique
- preuve formelle $\neq$ vérification

La logique

- théorie des ensembles (B)
- calcul des constuctions inductives (Coq)
- logique d'ordre supérieur (HOL,PVS)
- logique du premier ordre (Zenon)

$\rightarrow$ théorie des types / théorie des ensembles
$\rightarrow$ logique classique / logique intuitionniste
$\rightarrow$ isomorphisme de Curry-de Bruijn-Howard
↪ Extraction

Le Calcul des Constructions Inductives

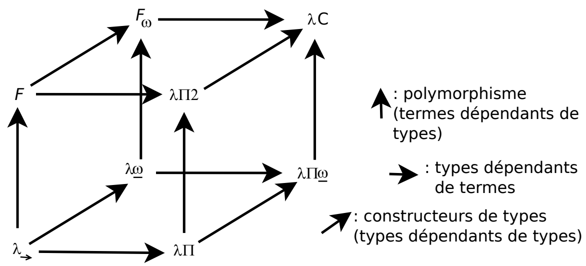
CoC (Calculus of Constructions) (Coquand-Huet, 85) +
types inductifs (Paulin, 88)

- λ-calcul simplement typé
- d'ordre supérieur
- déduction naturelle
- intuitionniste
- types inductifs
- types dépendants
- Curry-de Bruijn-Howard

(voir λ-cube de Barendregt)

$\hookrightarrow$ théorie des types de Martin-Löf avec égalité et univers

Le λ-cube de Barendregt



Le λ-calcul (généralités)

Church, 1936

- concepte de fonction et d'application
- définition :
  - les variables sont des λ-termes
  - (u v) est un λ-terme si u et v sont des λ-termes
  - λ x.v est un λ-terme si x est une variable et v un λ-terme
- exemple : λ x.x+2, (λ x.x+2)3, λ xy.x+y, λ x.(x x) (=Δ), ...
- conversions et réductions : α, β, ...
- normalisation, church-rosser, diamant (ou losange), ...
- λ-calcul simplement typé : rajout d'informations de typage ;
  λ x :nat.x+2 $\rightarrow$ terme bien typé $\rightarrow$ normalisation forte

$\hookrightarrow$ langages fonctionnels

Le λ-calcul (substitution, conversion et réduction)

- substitution : notée E[V := E′], définie inductivement :
  x[x := N] $\equiv$ N
  y[x := N] $\equiv$ y, si x $\neq$ y
  (M1 M2)[x := N] $\equiv$ (M1[x := N]) (M2[x := N])
  (λy.M)[x := N] $\equiv$ λy.(M[x := N]), si x $\neq$ y et y $\notin$ FV(N)
- α-conversion : λy.v $=_\alpha$ λz.v[y := z] (! substitution)
- β-réduction : (λx.xy)a $=_\beta$ (xy)[x := a] = ay
  Rq : on note $\rightarrow^*$ la fermeture réflexive transitive de la relation de réduction $\rightarrow$.
- ... (delta, iota, ...)

Le λ-calcul (propriétés)

- normalisation : t est normalisable s'il existe un terme u tel que t $=_\beta$ u (u est appelé la forme normale de t)
- normalisation forte : t est fortement normalisable si toutes les réductions à partir de t sont finies
  Ex : $\Omega = (\lambda x.xx)(\lambda x.xx) = \Delta\Delta$ n'est pas fortement normalisable
- church-rosser : soient t et u deux termes tels que t $=_\beta$ u. Il existe un terme v tel que t $\rightarrow^*$ v et u $\rightarrow^*$ v.
- diamant (confluence) : soient t, u1 et u2 des termes tels que t $\rightarrow^*$ u1 et t $\rightarrow^*$ u2, alors il existe un terme v tel que u1 $\rightarrow^*$ v et u2 $\rightarrow^*$ v.
- ... (lemme de la bande, ...)

Déduction naturelle

Définition : système de règles

$$\frac{A \in \Gamma}{\Gamma \vdash A}(\text{Axiome}) \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}(\Rightarrow\!-\text{intro}) \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}(\wedge\!-\text{ intro})$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}(\wedge\!-\text{ elim}) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}(\wedge\!-\text{ elim}) \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash \neg A}{\Gamma \vdash \bot}(\neg\!-\text{ elim})$$

(..., voir *[Dowek]* par exemple)

L'ordre supérieur

- logique du premier ordre : variables, fonctions, prédicats, quantificateurs
  on ne quantifie que sur les variables
- logique du second ordre :
  on peut également quantifier sur les prédicats et les fonctions
- filtres, ultrafiltres, ... (3ème ordre)

l'intuitionnisme

Brouwer, 1910

a réfuté 2 principes de la logique classique :

- le tiers exclus : $\varphi \vee \neg\varphi$
- le *il existe* non constructif : $\exists x, P\ x$

Exemple : montrer de manière classique et de manière intuitionniste qu'il existe deux nombres x et y irrationnels t.q. x^y soit rationnel.
(cf TP1)

Les types inductifs

Récursivité

- les entiers :

```
Inductive nat : Set :=  O : nat | S : nat -> nat
nat_rect P: forall P : nat -> Set,
  P 0 ->
  (forall n : nat, P n -> P (S n)) ->
  forall n : nat, P n
```

- les listes :

```
Inductive list (A : Type) : Type :=
    nil : list A | cons : A -> list A -> list A
list_rect P : forall (A : Type) (P : list A -> Set),
  P nil ->
  (forall (a : A) (l : list A), P l -> P (a :: l)) ->
  forall l : list A, P l
```


Les types dépendants

- les tableaux de taille n
- les listes de taille n
- ...

notation : $\Pi n : nat(tab\ n)$
rq : $A \rightarrow B$ est un cas particulier de $\Pi n : A\ B$
(où n n'intervient pas dans B)

l'isomorphisme de Curry-de Bruijn-Howard

Correspondance entre :

type et proposition
programme et preuve

Par exemple, nous pouvons faire une correspondance entre :

il existe un programme P de type T
et
il existe une preuve P' de la proposition T'.

En d'autres termes : les preuves sont des objets, les propositions sont des types, une preuve d'une proposition P est un objet p de type P.
(BHK interpretation)

Interprétation de Brouwer–Heyting–Kolmogorov

- une preuve de P $\wedge$ Q est une couple (a,b) où a est une preuve de P et b une preuve de Q ;
- une preuve de P $\vee$ Q est un couple (a,b) où a est 0 et b une preuve de P ou a est 1 et b une preuve de Q ;
- une preuve de P $\rightarrow$ Q est une fonction f qui convertit une preuve de P en une preuve de Q ;
- une preuve de $\exists$ x $\in$ S : ϕ(x) est un couple (a,b) où a est un élément de S et b est une preuve de ϕ(a) ;
- une preuve de $\forall$ x $\in$ S : ϕ(x) est une fonction f qui convertit un élément a de S en une preuve de ϕ(a) ;
- la formule $\neg$ P est définie par P $\rightarrow \perp$, une preuve de $\neg$ P es une fonction f qui convertit une preuve de P en une preuve de $\perp$;
- $\perp$ est le faux. Il n'y a pas de preuve du faux.

CCI

Rappel : CoC + types inductifs
Un terme du calcul des constructions est ainsi construit :

- T est un terme (appelé Type)
- P est un terme (appelé Prop, le type de toutes les propositions)
- Si A et B sont des termes, le sont aussi :
  - $(A\ B)$
  - $(\lambda x : A.B)$
  - $(\forall x : A.B)$
  - + les règles d'inférence...

(voir par exemple *[CoqRefMan]*)

Règles d'inférence du CoC

$$\frac{}{\vdash \mathbb{P} : \mathbb{T}}(\text{Sorte}) \qquad \frac{\Gamma \vdash A : K}{\Gamma, x : A \vdash x : A}(\text{Variable})$$

$$\frac{\Gamma, x : A \vdash t : B : K}{\Gamma \vdash (\lambda x : A.t) : (\forall x : A.B) : K}(\text{Abstraction})$$

$$\frac{\Gamma \vdash M : (\forall x : A.B) \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B(x := N)}(\text{Application})$$

$$\frac{\Gamma \vdash M : A \qquad A =_\beta B \qquad \Gamma \vdash B : K}{\Gamma \vdash M : B}(\text{Conversion})$$

Les sortes

- *Prop* : la sorte des propositions (imprédicative)
- *Set* : la sorte des types de données de base (prédicative, depuis la V8)
- *Type* : la hiérarchie *cumulative* d'univers (prédicative)

$Set = Type_0$, $Set : Type_1$, $Prop : Type_1$, $Type_i : Type_{i+1}$,
$Type_i \subseteq Type_{i+1}$, $Prop \subseteq Type_1$

Imprédicativité et prédicativité :

$$\frac{x : A \vdash B : Prop}{\forall x : A, B : Prop} \qquad \frac{\vdash A : s \qquad x : A \vdash B : Type_i}{\forall x : A, B : Type_j (i \leq j)}$$

avec $s = Set$ ou $s = Type_i$

Tactiques élémentaires

Tactique : "commande" donnée au prouveur pour l'aider à "faire la preuve".

Exemples de correspondance avec la déduction naturelle :

`assumption`	*Axiome*
`intros,intros`	$\Rightarrow$— intro, $\forall$— intro, $\neg$— intro
`apply`	$\Rightarrow$— elim, $\forall$— elim, $\neg$— elim
`split`	$\wedge$— intro
`left,right`	$\vee$— intro
`exists`	$\exists$— intro
...	

Définitions Inductives

Exemple : écrire une fonction qui calcule la somme des éléments d'une liste d'entiers.
En Ocaml :

```
let rec somme l=match l with
    [] -> 0
   |a::tl -> a+(somme tl);;
```

En Coq (`Require Export List`) :

```
Fixpoint somme (l: list nat):nat:= match l with
 |nil => 0
 |cons a tl =>a+(somme tl)
end.
```

Contraintes en Coq : les fonctions doivent terminer ; conditions de garde pour assurer la terminaison.

Std-lib (17)

- Init (core)
- Logic
- ZArith
- QAtith
- Reals
- FSets
- List
- Bool
- Setoids
- String
- Wellfounded
- ...

- Les nombres
- Ecrire des tactiques : le langage $\mathcal{L}_{tac}$
- Extraction
- Preuve de programmes

- *[Bertot]* http://fuscia.inrialpes.fr/cours/coq/
- *[CoqArt]* www.labri.fr/perso/casteran/CoqArt/index.html
- *[CoqRefMan]* http://coq.inria.fr/distrib/current/refman/
- *[Dowek]* https://who.rocq.inria.fr/Gilles.Dowek/Cours/Pit/pit.pdf
- *[Freek]* www.cs.ru.nl/~freek/comparison/
- *[Hardin]* http://pagesperso-systeme.lip6.fr/Mathieu.Jaume/cours_lambda_th.ps
- *[Miquel]* www.pps.jussieu.fr/~miquel/enseignement/mpri/guide.html
- *[Proofweb]* proofweb.cs.ru.nl/