Langages et Systèmes de Preuve

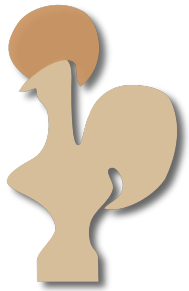
Christine Paulin-Mohring

Université Paris Sud & INRIA Saclay - Île-de-France

18 mars 2015

Objectifs

- Assistant de preuve



  - Coq
  - s'applique à d'autres systèmes de la même famille HOL, Isabelle, PVS, Agda, Matita...
- Paysage des mathématiques assistées par ordinateurs
- Concepts mis en œuvre : langage & architecture

Plan

Motivations

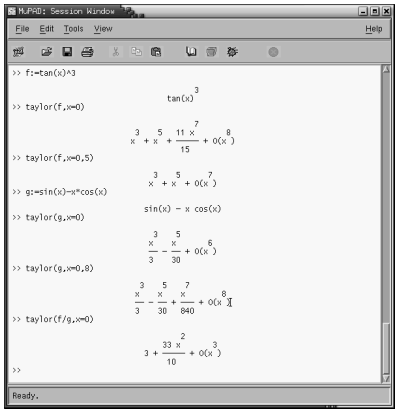
Pourquoi s'intéresser aux assistants de preuve ?

- Limitations liées à l'indécidabilité et la complexité
- Gérer des preuves infaisables à la main
- Intégrer raisonnement et calcul
- Maximiser la confiance

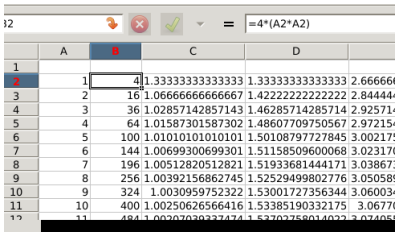
Emergence d'une nouvelle activité
Génie/Ingénierie des preuves

Outils pour le calcul

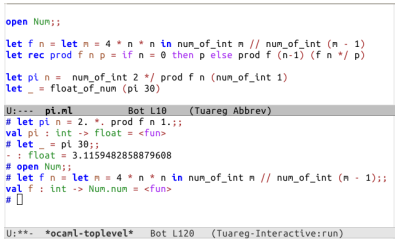
calculatrice
opérations de base



calcul formel
langage spécialisé,
fonctions avancées,
interface



feuille de calcul
présentation, édition



programmation
langage universel,
bibliothèques, spécialistes

problème : modéliser, combiner les méthodes

Calcul et Preuve

Calcul	Coq
Calculatrice	Noyau de vérification
Feuille de Calcul	Edition des preuves, tactiques
Calcul formel	Bibliothèques spécialisées
Programmation	Extension via plugin

On a désappris à calculer car les machines font cela mieux que nous

Pourra-t-on désapprendre la construction/vérification de preuves ?

Plan

Propriété du langage

- Il ne suffit pas que le calcul soit juste, il faut avoir posé les bonnes équations
  - les spécifications doivent pouvoir être remises en question : relecture, confronter plusieurs définitions, tester, exécuter, prouver
- Un problème bien posé est à moitié résolu
  - un programme bien spécifié est à moitié prouvé...
  - le langage comme support à l'expression des besoins
  - compréhensible par l'homme et la machine

Choix du langage

La logique est un langage universel pour représenter des propriétés

- les constructions de base

  `vrai`($\top$), `faux`($\bot$),
  `non`($\neg$), `et` ($\wedge$), `ou` ($\vee$), `implique`($\Rightarrow$), `équivalent`($\Leftrightarrow$)...
  `pour tout` ($\forall$), `il existe` ($\exists$)

- logiques de programme (ad-hoc)

$$
\begin{array}{lcl}
M, V \models Xv & \Leftrightarrow & M, (V_2, V_3, \ldots) \models v \\
M, V \models Fv & \Leftrightarrow & \exists i \geq 1, M, (V_i, V_{i+1}, \ldots) \models v \\
M, V \models Gv & \Leftrightarrow & \forall i \geq 1, M, (V_i, V_{i+1}, \ldots) \models v
\end{array}
$$

Représentation des objets

- Théorie des ensembles

$$4 \equiv \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\}$$

- Arithmétique de Peano $(0, S(\_), \_ + \_, \_ \times \_)$
  - codage des suites par des entiers
  - $2^n = m$

$$\exists e, |e| = n + 1 \wedge e(0) = 1 \wedge \forall i < n, e(i+1) = 2 \times e(i) \wedge e(n) = m$$

Analyse de la situation

- Machines de Turing comme modèle universel de calcul
  - oui mais les machines modernes sont plus sophistiquées
- Concevoir les langages modernes de la logique

Le point de vue informatique

- Mots binaires de taille bornée
- Code = suite finie d'instructions
  - une recette pour calculer des valeurs
  - moyen fini de décrire des objets potentiellement infinis
  - définition intentionnelle calculatoire
  - abstraction (lambda-calcul)/fermeture (langages fonctionnels)
- Organisation de la mémoire en blocs
  - entête : taille, étiquette pour organiser les blocs
  - termes, signature (symboles associés à des arités)

Types

- Des expressions (τ) pour représenter des ensembles d'objets
- Une relation $t : \tau$ pour classer les objets
  booléens, entiers, fonctions, arbres ...
- Reconnaissable statiquement (à la compilation, avant le calcul)
- En général un type canonique par objet
- Stable par calcul

  si $t : \tau$ et t s'évalue en u alors $u : \tau$
- Une manière de limiter les erreurs automatiquement
- Mais forcément des limitations

$$\mathbb{N}^* \qquad 1 : \mathbb{N}^* \qquad x^2 - 2x + 1 : \mathbb{N}^* ?$$

les types ne sont pas des ensembles !

Types simples

- Types fonctionnels : $\tau \to \sigma$

$$\frac{f : \sigma \to \tau \qquad p : \sigma}{f\ p : \tau} \qquad [f(x) \stackrel{\text{def}}{=} t]\frac{x : \sigma \vdash t : \tau}{f : \sigma \to \tau}$$

- Types de base : `bool`, `nat`
  - constructeurs avec leur arité (typée)

```
Inductive bool := true | false.
Inductive nat := O | S : nat → nat.
```

Définition de fonctions

- propriétés des types construits
  - valeurs constructibles : viennent des constructeurs
  - constructeurs distincts
- analyse par cas

```
Definition isz (n:nat) :=
  match n with O ⇒ true | S m ⇒ false end.
```

- point-fixe structurel

```
Fixpoint exp2 (n:nat) :=
  match n with O ⇒ 1 | S m ⇒ 2 * exp2 m end.
```

Des objets qui calculent

```
Eval compute in exp2 4.
     = 16 : nat
```

- Tester (sur des valeurs closes)
- Si on peut calculer $f(n)$ en m alors $f(n) = m$ est trivial.

Limitations

- Toutes les fonctions ne peuvent pas se calculer
  - minimiser une fonction $f : \mathbb{N} \to \mathbb{N}$ $\quad \forall x : \mathbb{N}, f(\mathtt{min}(f)) \leq f(x)$
- Une valeur booléenne se calcule
  Une propriété logique se prouve

$bool \neq$ Prop

Quels programmes dans la logique ?

Pas n'importe quel calcul dans la logique

- problème de terminaison

$$\forall n, n = 0 \vee n \neq 0$$
$$0 \neq 1$$
$$f(x) = \textbf{if } f(x) = 0 \textbf{ then } 1 \textbf{ else } 0$$
$$f(x) = 0 \Leftrightarrow f(x) \neq 0$$

- calculs impurs
  - $p = p$ non valide si p fait des effets de bord

$$(x{+}{+} == x{+}{+}) \qquad \texttt{random} == \texttt{random}$$

Solutions

- fonctions représentées intentionnellement (relations logiques)
- expliciter les effets des constructions impératives (monades)
  - états, exception, entrées-sorties, aléatoire, non-déterminisme
- notations

Types étendus

- types polymorphes : `list` α, `tree` α

$$\frac{t : \forall \alpha, \tau[\alpha]}{t[\sigma] : \tau[\sigma]}$$

```
Inductive tree A :=
  leaf | node : A → tree A → tree A → tree A.
```

- types indicés : `vec` n

$$\frac{t : \forall x : \sigma, \tau[x] \qquad p : \sigma}{t\,p : \tau[p]}$$

```
Inductive vec A : nat → Type :=
  nil : vec A O | add : ∀ n,A → vec A n → vec A (S n).
```

- branchement indicé

```
Inductive BDT := T|F| var: nat → (bool → BDT) → BDT.
Inductive W A B := sup: ∀(a:A),(B a → W A B) → W A B.
```

Types dépendants

Définition récursive de types

```
Fixpoint prodn A (n:nat) : Type :=
  match n with O ⇒ unit | S m ⇒ A * prodn A m end.

Fixpoint pairn A (n:nat) (a:A) : prodn A n :=
  match n return prodn A n with
     O ⇒ tt | S m ⇒ (a,pairn m a)
  end.

Eval compute in pairn 4 true.
     = (true, (true, (true, (true, tt))))
     : prodn bool 4
```

Langage fonctionnel avancé

Typage dépendant du match

- Forme élémentaire

```
match p as i in I return P i with
 c1 x1...xn1 ⇒ t1 (* : P (c1 x1...xn1) *)
|...
|cp x1...xnp ⇒ tp (* : P (cp x1...xnp) *)
end
: P p
```

- Formes plus complexes compilées

```
Fixpoint leb n m : bool := match (n,m) with
       (0,_)         ⇒ true
     | (_,0)         ⇒ false
     | (S(n),S(m)) ⇒ leb n m
    end.
```

Récapitulatif objets et types

- Langage fonctionnel typé polymorphe
  - abstraction, application
- Types inductifs construits
  - constructeurs, filtrage, point-fixe
- Spécifique à Coq
  - les types sont des objets
  - types définis récursivement
  - une notion interne d'égalité modulo calcul

Preuves et vérité

S'il existe une preuve alors la formule est vraie
oui mais

- incomplétude : les preuves de cohérence sont inatteignables
  - systèmes logiques faux : paradoxe de Russel, système *Type* : *Type* (Martin-Löf)...
  - mais des systèmes rodés existent

    théorie des ensembles, arithmétique, logique d'ordre supérieur
- en pratique : si l'utilisateur introduit des hypothèses alors celles-ci peuvent être contradictoires
  - des axiomatiques manuelles incohérentes
  - interaction de plusieurs axiomes
  - raisonnement parfaitement correct mais qui ne sert à rien

Avoir un système assez puissant pour définir au lieu de supposer

Logique d'ordre supérieur

- Deux connecteurs primitifs : $P \Rightarrow Q \quad \forall x : \tau, P$
- Un type des propositions o, des relations unaires $\tau \to o$, ...
- Quantification sur les propositions

$$\forall X : o, X \Rightarrow X \qquad \forall X : o, X$$

- Représentation des autres connecteurs logiques

$$P \wedge Q \stackrel{\text{def}}{=} \forall X : o, (P \Rightarrow Q \Rightarrow X) \Rightarrow X$$

- Quantification existentielle

$$\exists x : \tau, P \stackrel{\text{def}}{=} \forall X : o, (\forall x, P \Rightarrow X) \Rightarrow X$$

- Egalité

$$x =_\tau y \stackrel{\text{def}}{=} \forall X : \tau \to o, X\, x \Rightarrow X\, y$$

Propriétés en Coq

proposition = type de ses preuves

- une seule construction : implication et quantification universelle

$$A \Rightarrow B \stackrel{\text{def}}{=} A \to B \equiv \forall \_ : A, B$$

- autres connecteurs définis

```
Inductive True : Prop := I : True.
Inductive False : Prop :=.
Definition not (A:Prop) := A → False.
Inductive and (A B:Prop) : Prop :=
   conj : A → B → A /\ B
Inductive or (A B:Prop) : Prop :=
  | or_introl : A → A \/ B | or_intror : B → A \/ B
Inductive ex A (P:A → Prop) : Prop :=
  ex_intro : ∀ x:A, P x → ex P.
```

Prédicats de base en Coq

```
Inductive eq A (x:A) : A → Prop := eq_refl : x = x.
```

$$\frac{t \equiv u}{\texttt{eq\_refl}\ t : t = u} \qquad \frac{t = u \quad P\,t}{P\,u}$$

```
Inductive ftrans A R (x y : A) : Prop :=
  one : R x y → trans R x y
| join : ∀ z, trans R x z → trans R z y → trans x y.

Inductive Acc A R (x : A) : Prop :=
    Acc_intro : (∀ y:A, R y x → Acc R y) → Acc R x.
Definition well_founded A R := ∀ a:A, Acc R a.
```

Termes de preuves

- Proposition : type dans la sorte Prop.
- Preuve de P : terme de type P.
- Vérifier une preuve : typer un terme.
- Le typage est décidable
  - le terme contient tous les détails de la preuve
  - en Coq : intègre des calculs qui peuvent être complexes
- Le terme de preuve peut avoir d'autres applications
  - analyse des dépendances
  - explication
  - revérification indépendante

Exemple de preuve

```
∀ P : nat → Prop,
  P 0 → P 1 → (∀ n, P n → P (S (S n))) → ∀ n, P n
```

Preuve classique par récurrence

```
Variable P : nat → Prop.
Variables (p0:P 0) (p1:P 1) (p2:∀ n, P n → P (S (S n))).
Lemma nind2 : ∀ n, P n.
intro n; assert (P n /\ P (S n)).
induction n; intuition.
destruct H; assumption.
Qed.
```

Preuves vues comme des programmes

```
Fixpoint nind2 (n:nat) : P n := match n with
   | 0 ⇒ p0 | 1 ⇒ p1 | S (S m) ⇒ p2 m (nind2 m) end.
```

Une idée qui fait son chemin dans la preuve de programme

```
let nind2 n (* requires 0 <= n, ensures P n *) =
k:=!n;
while 2<=!k (* invariant 0 <= k /\ P(k) ⇒ P(n) *)
do k:=!k-2 done;
if k=0 then return;
if k=1 then return;
```

Constructivité

- algorithme de décision de la propriété P
  $\Leftrightarrow$ preuve de $\forall x : A, P\,x \vee \neg P\,x$
- algorithme qui réalise la spécification Q
  $\Leftrightarrow$ preuve de $\forall x : A, \exists y : B, Q\,x\,y$

- Distinction $\mathrm{Prop} \neq \mathrm{Set}$
- Extraction de programmes exécutables (Ocaml)

Limites du langage de Coq

- Plusieurs égalités
  - $t \equiv u$: identiques par calcul $2+2 \equiv 4 \quad 0+x \equiv x$ décidable
  - $t = u$: prouvablement identiques par calcul $n+m=m+n$ échangeable dans n'importe quel contexte (bien typé)
  - $t \simeq u$: relation d'équivalence, substitution via congruence
- Langage intentionnel : $f\,x \stackrel{\text{def}}{=} x+0, g\,x \stackrel{\text{def}}{=} 0+x : \ f \neq g$
- utilisation des types dépendants délicate :

$$\mathtt{vec}(n+m) \not\equiv \mathtt{vec}(m+n)$$

- Théorie Types Homotopiques : meilleur traitement de l'égalité
- Condition de garde des points fixes

Récapitulatif langage logique

Logiques d'ordre supérieur

- ingrédients de base : fonctions et structures construites
- définition récursive par calcul ou définition implicite par règles
- preuve par récurrence sur les termes et sur les définitions

Coq

- langage fonctionnel calculatoire puissant
- interprétation constructive des propositions
- représentation des preuves par des termes typés

Plan

Assistant de preuve

- Passer d'un langage (la calculatrice) à un système
- Langage de plus haut niveau
  - Sucre syntaxique, notations
  - Précompilation de constructions avancées
    - arguments implicites
    - filtrage complexe
  - Théories prédéfinies (logique, entiers, booléens, listes . . .)
- Assistance à la construction de preuve
  - mode interactif dirigé par le but
  - tactiques automatiques
  - bibliothèques (propriétés, tactiques)
- Efficacité
  - modules (compilation séparée)
- Interface
  - chercher dans les bibliothèques

Noyau de confiance

- Etat : ensemble de définitions correctement typées
  - déclaration de types inductifs
  - hypothèses
  - termes représentant des objets
  - termes représentant des preuves
- Vérification
  - ensemble de règles de typage élémentaires
  - règles de calcul
- Extensions garanties
  - contributions distribuées
  - bibliothèques
  - tactiques

Automatisation

- Tactiques élémentaires qui se composent
- Programmation de procédures complexes
  - doivent produire un terme de preuve revérifiable
- Approche par trace
  - recherche de preuve externe
  - vérification d'un certificat
- Approche réflexive : stratégie programmée et vérifiée en Coq
  - vérifier la procédure plutôt que chacun des résultats
  - ramener la vérification au calcul
  - termes de preuves compacts

Exemple

Spécification inductive

```
Inductive pair : nat → Prop :=
  pair0 : pair 0
| pair2 : ∀ n, pair n → pair (S (S n)).
```

Spécification récursive

```
Fixpoint pairb (n:nat) : bool := match n with
  0 ⇒ true | 1 ⇒ false | S (S m) ⇒ pairb m end.
Lemma pairbok : ∀ n, pairb n = true → pair n.
```

Type des entiers pairs

```
Record deuxN := mk2N {val:>nat; isp : pair val}.
```

```
Notation "n ':2N'":= mk2N n (pairbok n (eq_refl true)).
Definition p400 : deuxN := 400 :2N.
```

Base du langage SSREFLECT (bibliothèques, tactiques).

Reflexion avancée

Stratégie de preuve programmée en Coq et utilisée sur les propositions Coq

- Type concret : C (formules, expressions, traces...)
- Interpétation : $I : C \to \mathrm{Prop}$
- Décision : $d : C \to \mathtt{bool}$
- Correction : $\mathtt{cor} : \forall x, d\,x = \mathtt{true} \Rightarrow I\,x$
- Utilisation pour prouver A
  - réification : p (clos) tel que $A \equiv I\,p$
  - si $d\,p \equiv \mathtt{true}$ alors $\mathtt{cor}\,p\,(\mathtt{eq\_refl}\,\mathtt{true}) : A$

A propos de preuves par réflexion

- Possibilité de programmer les stratégies dans Coq
- La réification est un processus externe
- Langage de programmation limité
- Il faut prouver le programme *d*
- Exploite les capacités de calcul de Coq

Plan

Applications de Coq

- Calcul scientifique (continu)
- Arithmétique des ordinateurs (flottants)
- Cryptographie (certicrypt)
- Mathématiques (constructives ou non)
- Sémantique des langages
- Algorithmes avancés
- Outils certifiés
  - compilateurs, analyse statique, outils de vérification
- Cible pour des outils de vérification (Why3, edukera)

Conclusion

Les ingrédients de la réussite de Coq

- Un langage avec un nombre limité de concepts mais puissant
- Intégration du calcul et du raisonnement
- Correspondance programme/preuve
- Architecture sécurisée mais ouverte
- Des outils matures (encore pour spécialistes)
- Une pénétration dans d'autres communautés

Questions ?