Calcul des Constructions Inductives
D.E.A. Programmation: Sémantique, Preuves et Langages
2003-2004

Calcul des Constructions Inductives

Christine Paulin-Mohring, Benjamin Werner,

Bruno Barras, Hugo Herbelin,

Jean-Christophe Filliâtre, Claude Marché

Table des matières

8 Architecture des assistants à la démonstration **3**
- 8.1 Architecture de Coq 3
- 8.2 Critères de classification 3
- 8.3 Autres systèmes 5
- 8.4 Preuves par réflexion 5
  - 8.4.1 Utilisation de preuves de décidabilité 6
  - 8.4.2 Utilisation d'une structure abstraite 6
  - 8.4.3 Un exemple en Coq: l'associativité de l'addition sur les entiers naturels . . 6

Chapitre 8

Architecture des assistants à la démonstration

8.1 Architecture de Coq

Le système Coq repose sur un noyau de vérification du Calcul des Constructions Inductives. Les opérations de base de ce noyau consistent à ajouter une déclaration (variable, définition, déclaration inductive, module) dans l'environnement. Cela nécessite d'effectuer le typage des termes en particulier de contrôler des contraintes d'univers et de pouvoir tester la convertibilité de deux termes, ce qui passe par des étapes de réduction.

Au dessus de ce noyau, est construit un langage de description de haut niveau qui est compilé vers le CCI. Ce langage permet de laisser certaines informations implicites, par exemple d'utiliser des coercions entre différents types de données, d'utiliser les dépendances entre types pour omettre certains arguments de fonction qui seront calculés par unification. Le mécanisme de définition par filtrage est également compilé vers le filtrage atomique du CCI. On pourrait avoir des mécanismes de définition récursive de fonctions proches des langages de programmation et écrire des termes incomplets qui nécessitent des preuves complémentaires (par exemple pour traiter agréablement des fonctions partielles). La distinction de ces deux niveaux permet une souplesse d'expression sans modifier la théorie de base (en particulier sans risque de la rendre incohérente). Par contre elle introduit une distance entre ce que l'utilisateur écrit et ce que la machine prouve. Techniquement, cela complexifie l'interaction avec l'utilisateur (affichage, détection des erreurs).

Un second aspect de l'architecture de Coq est le mécanisme de construction de preuves qui se fait à l'aide de tactiques. Partant d'une propriété à montrer, on cherche à construire une preuve (dans le cas de Coq, un terme de ce type). Cela se fait à l'aide de tactiques qui travaillent sur un arbre de preuve dont la racine correspond à la propriété P à établir et les feuilles un ensemble de propriétés suffisantes pour prouver P. Les tactiques peuvent implanter des procédures arbitrairement complexes. Dans Coq, lorsque l'arbre n'a plus de feuilles, un terme de preuve est reconstruit puis vérifié par le noyau. On peut imaginer différents langages pour exprimer les étapes de preuve.

8.2 Critères de classification

Il est d'usage de classer les systèmes d'aide (ou assistants) à la démonstration selon les critères suivants [Bar81, Wieb, Wiea].

- Critère de de Bruijn
- Logique ou méta-logique
- Principe de Poincaré

- La représentation des preuves
- Développement interactif des preuves
- Degré d'automatisation

Dans la suite, on décrira ces différents critères et on analysera divers systèmes de développements de démonstration (ACL2, PVS, HOL, Isabelle, MetaPrl - anciennement NuPrl -, Mizar et Coq) à travers ces critères.

Le critère de de Bruijn Le critère de de Bruijn caractérise les systèmes dont la part dédiée à la certification de la correction des preuves est petite et bien délimitée.

Chacun des systèmes HOL, Isabelle et Coq a un « noyau » consacré à la certification des preuves et en ce sens vérifie le critère de de Bruijn. Toutefois, autant les noyaux de HOL et Isabelle restent assez petits, autant celui de Coq est assez conséquent (en particulier en raison de la gestion des types inductifs et de la réduction).

En revanche, ni Mizar, ni PVS n'ont une notion de « noyau » bien délimité. En particulier, de nouvelles méthodes de preuves peuvent être ajoutées à ces systèmes et la correction des preuves nouvellement obtenues ne dépendra que de la correction de l'implantation de la nouvelle méthode, pas d'un « noyau » stable et préalablement bien circonscrit.

Logique ou méta-logique ? Le choix des systèmes Isabelle et MetaPrl est de fournir non pas une logique mais une méta-logique (« logical framework ») permettant de déclarer les signatures et règles d'inférences de logiques arbitraires. Dans la pratique, compte tenu du développement d'une bibliothèque minimale nécessaire à toute formalisation conséquente, seules peu de logiques sont effectivement implantées dans un système basé sur une méta-logique. Ainsi, Isabelle par exemple, offre essentiellement des bibliothèques pour la logique d'ordre supérieur de Church (HOL) et la théorie des ensembles de Zermelo-Fraenkel (ZF).

Le principe de Poincaré Le principe de Poincaré [Poi02] caractérise les systèmes qui distinguent entre simples vérifications calculatoires et étapes de preuve. Poincaré prend l'exemple de la propriété $2+2=4$ qui ne se justifie pas par une preuve mais plutôt par une vérification par calcul. Des systèmes comme Coq, Isabelle, MetaPrl, PVS et HOL utilisent une *règle de conversion* qui identifie des propriétés équivalentes modulo certaines règles de calcul. Dans HOL, cette règle de conversion ne prend en compte que la β-réduction dans un lambda-calcul simplement typé alors que dans Coq, on identifie les termes modulo la ι-réduction qui permet de calculer une grande classe de fonctions récursives. Mizar par contre n'intègre pas de notion de calcul.

Le principe de Poincaré peut être implanté à des degrés très divers. Par exemple, dans le Calcul des Constructions Inductives, une preuve de $0+n=n$ relève de la simple vérification (c'est la règle de conversion) tandis qu'une preuve de $n+0=n$ nécessite une étape d'induction et des étapes de réécriture. Autrement dit, dans le Calcul des Constructions Inductives, seul un quotient relativement à une évaluation séquentielle des programmes est introduit dans la logique. On pourrait ainsi imaginer d'étendre la règle de conversion à un quotient relativement à une évaluation non déterministe des programmes. C'est ce que se propose de faire les extensions du calcul des constructions avec des règles de réécriture.

Toutefois, il existe une manière de ramener toute procédure de simplification décidable à une application de la règle de conversion et une étape de réécriture. C'est le mécanisme de *réflexion*.

La représentation des preuves Un système de preuve peut soit valider une preuve sans garder aucune trace de la vérification autre que le source de la preuve. C'est le cas de PVS et ACL2. Ceci est forcément le cas lorsqu'aucun « noyau » ne vérifie les preuves. Toutefois, un système peut vérifier le critère de de Bruijn et ne pas garder de trace des preuves. C'est le cas de

HOL qui déduit de nouveaux théorèmes à partir de règles d'inférence bien définies sans garder de trace de quelles règles ont été appliquées pour valider tel ou tel théorème.

Un système peut valider les preuves sous la forme d'un terme de preuve. C'est le cas de Coq et Isabelle.

Le développement interactif des preuves Tous les systèmes n'offrent pas un mécanisme de développement interactif de preuves. Par exemple, Mizar et ACL2 ne peuvent que vérifier une preuve dans sa globalité. En cas de non validation, l'utilisateur doit apporter des modifications globales à la preuve.

De l'autre côté, les systèmes HOL, Coq, Isabelle et PVS, héritent tous de la notion de tactique introduite dans le système LCF et offrent ainsi la possibilité de développement de preuve interactifs.

Le degré d'automatisation Le degré d'automatisation est un facteur clé dans la diffusion des assistants de preuve en dehors du milieu des logiciens. Souvent, l'automatisation, car elle applique des méthodes « de force brute » s'oppose au souci de lisibilité des preuves : un énoncé même simple aura facilement une preuve complexe, et en tout cas non directe. Une réponse à ce souci est une nouvelle fois le mécanisme de preuve par réflexion qui isole la méthode de preuve en une étape élémentaire (à l'échelle humaine) de démonstration.

PVS et ACL2 sont actuellement les systèmes les plus automatisés parmi ceux mentionnés ci-dessus. Viennent ensuite Isabelle, puis HOL et Coq.

8.3 Autres systèmes

On peut mentionner de nombreux autres systèmes dont les bibliothèques et la base d'utilisateur sont moins développées.

Lego [Pol] est une implémentation du Calcul des Constructions enrichie par des déclarations de types inductifs. Le système Plastic [Gro] est une continuation de Lego expérimentant divers mécanismes de coercions. Ces systèmes sont développés à Edimbourg et Durham au Royaume-Uni.

Alfa/Agda [Coq] est un système expérimental de développement de preuve « comme un programme »: les preuves sont des λ-termes qui sont saisis interactivement via une interface graphique. Ces systèmes sont développés à l'université de Chalmers en Suède.

PhoX [Raf] est un système basé sur l'arithmétique d'ordre 2 manipulant des programmes ML avec inférence de type polymorphe à la Milner. Son utilisation est importante en milieu enseignant. Il est développé par C. Raffali à l'université de Savoie.

Twelf [PS] est une méta-logique dotée d'un mécanisme d'unification d'ordre supérieur. Twelf est utilisé pour modéliser la sémantique de langages de programmation. Ce système est développé à Carnegie Mellon University dans l'équipe de Frank Pfenning.

8.4 Preuves par réflexion

La réflexion est une technique permettant de remplacer les étapes de preuves associées à une procédure de simplification ou de décision en la combinaison d'une unique étape de preuve et du calcul. En ce sens, la technique de réflexion suit le principe de Poincaré qui consiste à sortir du langage de preuve les méthodes de preuves qui se ramènent à un simple calcul.

8.4.1 Utilisation de preuves de décidabilité

Supposons qu'une propriété A sur un ensemble U soit décidable. On a alors une preuve `dec` de la propriété $\forall x : U.\{A\ x\}+\{\neg(A\ x)\}$. On en déduit aisément une fonction de décision booléenne : `dec_bool` : $U \to$ `bool` et sa propriété de correction :
`correct` : $\forall x : U.($`dec_bool` $u) =$ `true` $\to A\ u$.

Supposons que l'on veuille prouver la propriété $(A\ u)$ pour un terme u clos. Le terme (`dec_bool` u) est clos et de type `bool`, il doit s'évaluer vers la valeur `true`. Une preuve possible de $(A\ u)$ est donc :

`correct` u (`refl true`)

La vérification que ce terme est de type $(A\ u)$ nécessite de typer u puis de typer (`refl true`) de type (`dec_bool` u) = `true` ce qui revient à réduire (`dec_bool` u) en `true`, ce qui peut nécessiter un calcul complexe.

8.4.2 Utilisation d'une structure abstraite

En général, on souhaite montrer des propriétés sur des termes qui ne sont pas forcément clos et sur lesquels il n'est pas forcément simple de construire des procédures de décision. On introduit alors une structure abstraite intermédiaire qui va représenter la syntaxe des expressions à manipuler et qui va permettre des manipulations symboliques. On a alors besoin d'une fonction d'interprétation de la syntaxe vers les propriétés à montrer.

La technique de réflexion procède comme suit:

- Définition d'une structure abstraite S pour le type de problème auquel s'adresse la méthode de décision ou de simplification considérée
- Définition d'une fonction d'interprétation $x : S \mapsto |x|$ des objets de cette structure abstraite S en une expression concrète (un terme ou une formule) du système logique
- Définition de la fonction ϕ de décision ou de simplification par calcul sur les objets de la structure abstraite
- Preuve que la méthode est valide, c'est-à-dire que $\forall s : S, |s| = |\phi(s)|$ pour une procédure de simplification sur les termes, $\forall s : S, |s| \leftrightarrow |\phi(s)|$ pour une procédure de simplification sur les propositions (sachant qu'une procédure de décision peut être vue comme une procédure de simplification renvoyant soit la proposition vraie soit la proposition fausse)

Ces bases étant posées, la simplification d'un énoncé de la forme $\psi(t)$ en l'énoncé $\psi(\phi(t))$ où t a été simplifié procède par une simple application du lemme de validité de ϕ. En effet, par applicabilité de la méthode de décision, il existe un $s : S$ tel que $|s|$ est convertible avec t, qui par le lemme de validité est égal à $|\phi(s)|$ qui lui-même est convertible en la forme simplifiée de t.

En fait, plus généralement, on considère des structures abstraites avec des variables et des fonctions d'interprétation paramétrées par une substitution de ces variables par des sous-termes non traitables par la méthode de simplification. On a alors un lemme de validité qui a la forme $\forall s : S, \forall \sigma : Var \to Term, |s|_\sigma = |\phi(s)|_\sigma$.

8.4.3 Un exemple en Coq: l'associativité de l'addition sur les entiers naturels

On considère des expressions construites à partir de l'addition (`plus`, notée +) sur les entiers naturels que l'on souhaite normaliser sous une forme associative à droite en supprimant les zéros (par exemple, la normalisation de `(x+u)+((y*t)+0)` est `x+(u+(y*t))`).

Construction de la structure abstraite représentant les expressions construites à partir de + et 0 On prendra les entiers naturels eux-mêmes pour dénoter les variables.

```
Coq < Definition index := nat.

Coq < Inductive expr : Set :=
Coq <       | Plus : expr -> expr -> expr
Coq <       | Zero : expr
Coq <       | Var : index -> expr.
```

Construction de la fonction d'interprétation

```
Coq <     Require Import List.

Coq <     Require Import Plus.

Coq < (* Valeur par défaut de nth si la substitution n'était pas de bonne longueur *)
Coq <     Definition default := 0.

Coq < (* Fonction d'interprétation *)
Coq <     Fixpoint interp (s:list nat) (e:expr) {struct e} : nat :=
Coq <       match e with
Coq <       | Plus e1 e2 => interp s e1 + interp s e2
Coq <       | Zero => 0
Coq <       | Var i => nth i s default
Coq <       end.
```

Construction de la fonction de simplification

```
Coq < Fixpoint insere (e1 e:expr) {struct e1} : expr :=
Coq <       match e1 with
Coq <       | Plus e1 e2 => insere e1 (insere e2 e)
Coq <       | Zero => e
Coq <       | Var i => Plus e1 e
Coq <       end.

Coq < Fixpoint norm (e:expr) : expr :=
Coq <       match e with
Coq <       | Plus e1 Zero => norm e1
Coq <       | Plus e1 e2 => insere e1 (norm e2)
Coq <       | x => x
Coq <       end.
```

Construction de la preuve de correction de la simplification

```
Coq < Lemma validite_insere :
Coq <      forall (s:list nat) (e1 e2:expr),
Coq <        interp s (insere e1 e2) = interp s (Plus e1 e2).

Coq < Proof.

Coq < induction e1; intro e2; simpl; auto.

Coq < rewrite plus_assoc_reverse.

Coq < change (interp s e1_2 + interp s e2) with (interp s (Plus e1_2 e2)).

Coq < rewrite <- IHe1_2.
```



```
Coq < rewrite IHe1_1; trivial.

Coq < Qed.
```

```
Coq < Theorem validite :
Coq <      forall (s:list nat) (e:expr), interp s (norm e) = interp s e.

Coq < Proof.

Coq < induction e; simpl; auto.

Coq < destruct e2 as [e e0| | i].

Coq < rewrite validite_insere.

Coq < rewrite <- IHe2; trivial.

Coq < simpl interp; rewrite IHe1; auto.

Coq < rewrite validite_insere.

Coq < rewrite <- IHe2; trivial.

Coq < Qed.
```

Après ce travail préalable, chaque application de la méthode de simplification se déroule comme suit.

```
Coq < Variable P : nat -> Prop.

Coq < Lemma exemple : forall x y t u:nat, P (0 + x + (u + y * t)).

Coq < intros.
1 subgoal

  x : nat
  y : nat
  t : nat
  u : nat
  ============================
   P (0 + x + (u + y * t))

Coq < pose (sigma := x :: u :: y * t :: nil);
Coq < change (P (interp sigma (Plus (Plus Zero (Var 0)) (Plus (Var 1) (Var 2))))).
1 subgoal

  x : nat
  y : nat
  t : nat
  u : nat
  sigma := x :: u :: y * t :: nil : list nat
  ============================
   P (interp sigma (Plus (Plus Zero (Var 0)) (Plus (Var 1) (Var 2))))

Coq < rewrite <- validite.
1 subgoal

  x : nat
  y : nat
  t : nat
```

```
  u : nat
  sigma := x :: u :: y * t :: nil : list nat
  ============================
   P
     (interp sigma
        (norm (Plus (Plus Zero (Var 0)) (Plus (Var 1) (Var 2)))))
Coq < simpl interp; simpl norm; clear sigma.
1 subgoal

  x : nat
  y : nat
  t : nat
  u : nat
  ============================
   P (x + (u + y * t))
```

Idéalement le travail consistant à trouver s tel que $|s| = t$ devrait être automatisé. Cela peut se faire en ML ou en utilisant le langage de tactique.

Un exemple plus consistant de tactique par réflexion disponible dans Coq est la tactique `Ring`. Un model-checker utilisant des BDDs a également été construit selon ce modèle [VGLPAK00].

Il n'est pas toujours commode ni très efficace de programmer dans le langage de Coq des procédures de recherche de preuve complexes. Les outils de preuve automatique peuvent souvent être adaptés pour produire une trace de preuve. On peut alors simplement internaliser dans Coq la notion de trace et la preuve de correction de cette trace. Ainsi la preuve vérifiée par Coq effectue un calcul sur la trace et sa taille est proportionnelle à cette taille. Cette approche a été utilisée pour construire une interface entre Coq et le système de réécriture Elan, ou dans une version réflexive de la tactique de preuve en arithmétique `Omega`.

Bibliographie

[Bar81] H.P. Barendregt. *The Lambda Calculus its Syntax and Semantics*. North-Holland, 1981.

[Coq] C. Coquand. Agda. `http://www.cs.chalmers.se/~catarina/agda/`.

[Gro] Computer Assisted Reasoning Group. The PLASTIC proof assistant. `http://www.dur.ac.uk/CARG/plastic.html`.

[Poi02] H. Poincaré. *La Science et l'Hypothèse*. Flammarion, 1902.

[Pol] R. Pollack. The LEGO proof assistant. `http://www.dcs.ed.ac.uk/home/lego/`.

[PS] F. Pfenning and C. Schürmann. The Twelf project. `http://www-2.cs.cmu.edu/~twelf/`.

[Raf] C. Raffalli. The PhoX proof assistant. `http://www.lama.univ-savoie.fr/sitelama/Membres/pages_web/RAFFALLI/af2.%html`.

[VGLPAK00] K. Neeraj Verma, J. Goubault-Larrecq, S. Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *ASIAN'2000*, volume 1961 of *LNCS*, pages 162–181, 2000.

[Wiea] F. Wiedijk. Comparing mathematical provers. `http://www.cs.kun.nl/~freek/comparison/index.html`.

[Wieb] F. Wiedijk. The fifteen provers of the world. `http://www.cs.kun.nl/~freek/comparison/index.html`.