Lot 5.1

Technologie de vérification

Ajouter la réécriture au noyau de Coq

Proposition d'architecture du moteur de test de conversion

Description :	Nous examinons l'architecture actuelle de Coq et discutons différentes solutions pour y ajouter la récriture. Ensuite, dans le but d'implanter un prototype rapidement, sans bouleverser le code de Coq, nous proposons une architecture simple utilisant CiME, une bibliothèque OCaml pour la récriture.
Auteur(s) :	Frédéric Blanqui
Référence :	Averroes / Lot 5.1 / Fourniture 2 / V1.0
Date :	30 mars 2004
Statut :	validé
Version :	1.0

Réseau National des Technologies Logicielles
Projet subventionné par le Ministère de la Recherche et des Nouvelles Technologies

CRIL Technology, France Télécom R&D, INRIA-Futurs, LaBRI (Univ. de Bordeaux – CNRS), LIX (École Polytechnique, CNRS) LORIA, LRI (Univ. de Paris Sud – CNRS), LSV (ENS de Cachan – CNRS)

Historique

10 février 2004	V 0.1	création du document
30 mars 2004	V 1.0	version finale

Table des matières

1 Architecture de Coq

Nous commençons par présenter l'architecture actuelle de Coq [7]. Ensuite, nous discutons différentes possibilités pour pouvoir y intégrer de la récriture. Nous considérons la version 7.3 (bugfix) de Coq qui date de mai 2002. La version suivante 7.4 (février 2003) et la version en cours de développement (version 8.0 beta, décembre 2003) ne remettent pas en cause le moteur de test de conversion. L'adaptation de la présente architecture à la future version 8.0 de Coq ne doit donc pas poser de difficultés.

Une brève description peut être trouvée dans [4, 9]. Le code de Coq est divisé en 9 répertoires :
- `lib` : modules implantant des structures de données et fonctions d'utilité générale.
- `kernel` : modules implantant le moteur de test de conversion et le vérificateur de type.
- `library` : modules implantant les méchanismes de “backtracking” (commande `Undo`).
- `pretyping` : modules implantant l'inférence de type.
- `interp` : modules transformant les AST[1] en termes non typés.
- `parsing` : modules implantant la lecture, le codage et l'affichage des commandes et termes fournis par ou à l'utilisateur.
- `proofs` : modules implantant le moteur de preuves.
- `tactics` : modules implantant le language de tactiques.
- `toplevel` : modules implantant le lancement et l'exécution de Coq.

1.1 Noyau

Le répertoire qui nous intéresse le plus, `kernel`, contient les fichiers suivants. Ceux relatifs à la définition des termes :
- `univ` : univers et contraintes d'univers.
- `names` : noms des objets.
- `term` : termes.

Ceux implantant le test de conversion :
- `esubst` : substitutions explicites.
- `closure` : structure de données utilisée pour la réduction.
- `conv_oracle` : fonction indiquant quelle δ-réduction réaliser en premier dans le cas où on doit faire un choix.
- `reduction` : fonctions de réduction et de test de conversion.

Ceux implantant les environnements :
- `sign` : séquences de déclarations.
- `entries` : structures de donnée utilisées pour rajouter un objet dans un environnement.
- `declarations` : structures de données utilisées en interne pour représenter les objets d'un environnement.
- `environ` : environnements.
- `cooking` : ?

Enfin, ceux implantant la vérification de type :
- `type_errors` : erreurs de typage.
- `indtypes` : vérifie l'admissibilité d'un type (co)inductif.
- `inductive` : vérifie l'admissibilité d'une fonction définie par (co)point fixe.
- `typeops` : inférence de type.
- `safe_typing` : environnements bien typés.

1.2 Réduction

Dans le système Coq, le moteur de test de conversion doit prendre en compte 4 types différents de définitions ou relations de réduction :
- β-réduction : dans l'application d'une fonction, remplacement dans le corps de la fonction des arguments formels par leur valeur.

[1] Arbre de Syntaxe Abstrait.

- ι-réduction : réduction engendrée par les définitions (co)inductives introduites par les mot-clés `Fixpoint`, `Fix`, `CoFixpoint` et `CoFix`.
- δ-réduction : réduction engendrée par les définitions introduites par les mot-clés `Definition` et `Local`.
- ζ-réduction : réduction engendrée par les définitions introduites par le mot-clé `let`.

La réduction est implantée par une machine abstraite. Plus de précisions peuvent être trouvées dans la thèse de Bruno Barras [1]. Le type des termes est `constr`. Le module `closure` définit un type `clos_infos` qui permet de définir quelles réductions on souhaite voir appliquées (on peut préciser les symboles que l'on veut δ-réduire individuellement), et un type `fconstr` pour représenter les termes dans la machine. Un état de la machine est donné par un `fconstr` (la tête) et une pile de `fconstr` (les arguments). Le module `closure` exporte en particulier les fonctions suivantes :

- `inject : constr -> fconstr` transforme un `constr` en `fconstr` pour pouvoir le réduire.
- `val whd_stack : clos_infos -> fconstr -> fconstr stack -> fconstr * fconstr stack` calcule la forme $\beta\delta\iota\zeta$-normale de tête d'un `fconstr` appliqué à une pile d'arguments.
- `term_of_fconstr : fconstr -> constr` transforme un `fconstr` en `constr`.
- `whd_val : clos_infos -> fconstr -> constr` calcule la forme $\beta\delta\iota\zeta$-normale de tête d'un `fconstr`.
- `norm_val : clos_infos -> fconstr -> constr` calcule la forme $\beta\delta\iota\zeta$-normale d'un `fconstr`.

1.3 Conversion

L'algorithme de test de conversion implanté dans la version 7.3 de Coq est dû à Bruno Barras [1]. L'idée de base est proche de celle de Thierry Coquand [8] qui ne considère que la β-réduction. Pour tester la β-équivalence de 2 termes u et v, on commence par réduire u et v jusqu'à leurs formes normales de tête qui, dans ce cas, doivent être une séquence d'abstractions suivie d'une constante ou d'une variable (la tête) appliquée à une séquence de termes. Si les têtes sont différentes, les termes sont nécessairement distincts car la tête est invariante par réduction. Si par contre les têtes sont égales, il convient alors de tester l'équivalence de leurs arguments deux à deux de manière récursive. Bruno Barras a étendu cet algorithme aux autres réductions et l'a implanté de manière efficace à l'aide de substitutions explicites.

Dans les fonctions de conversion décrites ci-après, un terme est décomposé en 2 ou 3 éléments : un terme de type `lift` (défini dans `esubst`) représentant des renumérotations des indices de de Bruijn ("lifts"), et un `fconstr` seul ou un `fconstr` (la tête) avec la pile de ses arguments. Le test de conversion produit des contraintes d'univers satisfiables si les deux termes sont convertibles, ou lève une exception.

- La fonction de conversion proprement dite est `fconv` et s'applique à des `constr`. Elle commence par tester si les deux `constr` sont syntaxiquement égaux (fonction `eq_constr` dans `term` efficace grâce au "hash-consing"). Dans le cas contraire, elle crée un `clos_infos` sans δ-réduction (les δ-réductions nécessaires au test de conversion seront rajoutées au fur et à mesure), transforme les `constr` en `fconstr` avec `inject`, et appelle `ccnv` avec des lifts égaux à `ELID` (identité).
- `ccnv` calcule les formes normales de tête avec `whd_stack`, et appelle `eqappr`.
- `eqappr` compare les têtes. Dans le cas où une tête seulement est une constante, elle la δ-réduit et rappelle `eqappr` avec le résultat. Dans le cas où les deux têtes sont des constantes, elle appelle `oracle_order` (dans module `conv_oracle`) pour savoir quelle constante δ-réduire. Enfin, dans le cas où les deux têtes sont égales, on compare les arguments.

2 Ajout de la récriture : discussion

L'implantation actuelle de Coq repose de manière importante sur le fait que, pour savoir si une application est réductible en tête, il suffit de regarder les sous-termes immédiats : si on a une abstraction, l'application est β-réductible, et si on a un constructeur, l'application est ι-réductible. Avec des règles de récriture a priori quelconque, et en particulier avec de la récriture modulo AC, il en va tout autrement. Il peut être nécessaire d'examiner des sous-termes à une profondeur supérieure à un pour savoir si un terme est réductible en tête. De plus, un terme non réductible en tête n'est pas nécessairement en forme normale de tête : des réductions en profondeur peuvent être nécessaires pour pouvoir réduire un terme en tête.

On voit donc que l'implantation efficace du test de conversion en présence de récriture et de β-réduction posent des problèmes complexes. Ceux-ci n'ayant pas encore été explorés, davantage de recherche est nécessaire avant de pouvoir proposer un algorithme efficace pour l'implantation de Coq avec récriture, la résolution de ces problèmes risquant de remettre profondément en cause l'implantation actuelle du noyau de Coq. Cependant, l'intégration de récriture à Coq pose un certain nombre de questions auxquelles il nous semble nécessaire de répondre avant de choisir une architecture particulière :

Distinguer récriture et ι-réduction ? Bien que, de manière générale, la récriture englobe certaines des réductions déjà présentes dans Coq (en particulier la ι-réduction, mais la δ-réduction aussi), il a été décidé dans la définition de la classe de récriture à intégrer [2] que la récriture serait rajoutée à Coq sans pour autant se substituer aux réductions déjà présentes, et qu'elle s'appliquerait à des objets habituellement considérés comme constants dans Coq (`Parameter` et `Axiom`). Cependant, on peut se demander si, en terme de maintenance et de sécurité, il est bien raisonable de traiter de manière particulière ces réductions (en particulier la ι-réduction). Mais traiter la ι-réduction comme un cas particulier de récriture requiert de changer une partie importante du code de Coq.

(Ré)implanter la récriture soi-même ou faire appel à des outils spécialisés ? L'implantation efficace de la récriture, et en particulier de la récriture modulo AC, est quelque chose de non trivial. Cela représente de 10 à 20000 lignes de code dans CiME par exemple, alors que le noyau de Coq fait environ 10000 lignes de code. La question se pose alors s'il n'est pas raisonnable de faire appels à des outils spécialisés en récriture (*e.g.* CiME [6], Elan [3], TOM [12], Maude [5]), mais Coq n'est alors pas à l'abris d'erreurs dans le système utilisé. D'autre part, il y a plusieurs manières de faire appel à un tel système. Réutiliser son code pose des problèmes de maintenance : que faire si une erreur est découverte dans le code importé ? Appeler le système de manière dynamique pose des problèmes d'installation : l'utilisateur doit avoir installé une version approprié du système pour pouvoir installer Coq. Par ailleurs, le système doit/peut-il prendre en charge la β-réduction aussi (qui peut-être vue comme de la récriture du premier ordre si des substitutions explicites sont utilisées) ?

Interpréter ou compiler ? Enfin, on peut distinguer deux manières d'implanter la récriture qui peuvent se combiner. L'approche par interprétation consiste à disposer d'un code générique permettant de calculer la forme normale d'un terme à partir d'un ensemble de règles de récriture (c'est l'approche développée dans CiME par exemple). L'approche par compilation consiste à produire un programme qui, une fois compilée en binaire, calcule la forme normale d'un terme pour un ensemble de règles de récriture particulier (c'est l'approche développée dans Elan par exemple [11, 10]). La combinaison des deux approches semble naturelle et souhaitable puisque, d'une part, l'approche par interprétation est nécessaire dans une session interactive et, d'autre part, l'approche par compilation fournit une procédure de normalisation plus efficace que l'approche par interprétation. Cependant, l'utilisation de code compilé dans le noyau pose deux problèmes importants. Le premier est d'ordre technique : il nécessite de faire appel à des programmes contruits et compilés de manière dynamique. Le second est relatif à la sécurité : quelle confiance peut-on accorder à du code compilé ? D'utiliser du code extrait de preuves Coq pourrait être une solution.

3 Proposition d'architecture

L'objectif d'Averroes étant de réaliser un prototype pour tester l'utilisation de récriture en Coq et non de réaliser une nouvelle version de Coq incluant de la récriture tout en étant sûre et efficace, il est naturel d'adopter une architecture qui nécessite de modifier Coq le moins possible. C'est ainsi que nous avons choisi de distinguer récriture et ι-réduction, d'utiliser le code de CiME [6] pour $\mathcal{R}$-normaliser les termes (on appellera $\mathcal{R}$-réduction la réduction engendrée par les règles de récriture déclarées dans l'environnement), et donc d'interpréter les règles de récriture. CiME fournit également une fonction vérifiant la confluence locale (modulo AC) d'un ensemble de règles de récriture.

Nous décrivons maintenant comment modifier la procédure de conversion de Coq pour y inclure la récriture. Dans `eqappr`, quand une tête au moins est une constante définie par récriture, il est peut-être nécessaire de faire quelques récritures pour montrer que les deux termes sont équivalents. Cependant, pour être déclenchables, ces récritures peuvent nécessiter que les arguments soient d'abord $\beta\delta\iota\zeta$-normalisés. De plus, en présence de symboles commutatifs ou associatifs-commutatifs, il ne suffit plus de tester l'égalité syntaxique des arguments : comme dans CiME, il faut "aplatir" et ordonner les arguments de façon à tester leur AC-équivalence (par exemple, dans CiME, si $x < y < z$ alors $+(+yx)z$ est représenté par $+xyz$).

On voit donc deux inconvénients importants à cette approche. Premièrement, le fait d'avoir à $\beta\delta\iota\zeta$-normaliser avant de $\mathcal{R}$-normaliser fait perdre l'avantage d'une normalisation progressive (passage par les formes normales de tête) qui permet de détecter plus tôt une erreur de type. Deuxièmement, l'utilisation du code de CiME pour normaliser un terme Coq nécessite de nombreuses conversions entre structures de données. Etant donné que la structure de `fconstr` utilisée pour la $\beta\delta\iota\zeta$-réduction est complexe, il est plus aisé de définir une conversion entre le type `constr` de Coq et le type `term` de CiME qui, à la différence de Coq, utilise des listes pour les arguments d'un symbole, les "aplatit" et les ordonne. On a donc, à chaque $\mathcal{R}$-normalisation, deux double conversions : de `fconstr` à `constr` (avec la fonction `term_of_fconstr`), de `constr` à `term`, et la même chose dans l'autre sens après la $\mathcal{R}$-normalisation.

La conversion entre `constr` et `term` pourrait être éliminée si CiME avait une interface plus générique et était capable de travailler sur n'importe quelle structure de données (ce qui ne paraît pas difficile à réaliser au moins en absence de symboles AC). C'est par exemple ce que permet TOM [12]. Pour éliminer la conversion entre `fconstr` et `constr`, il faut en plus être capable d'exprimer l'effet d'appliquer une règle de récriture (exprimée par des `constr`) sur un `fconstr`.

Références

[1] B. Barras. *Auto-validation d'un système de preuves avec familles inductives.* PhD thesis, Université Paris VII, France, 1999.

[2] F. Blanqui. Définition de la classe de réécriture à intégrer. Averroes, lot 5.1, fourniture 1, 2004.

[3] P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. *ELAN User Manual.* INRIA Nancy, France, 2000. `http://www.loria.fr/ELAN/`.

[4] J. Chrząszcz. Implementation of modules in the Coq system. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science 2758, 2003.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude : Specification and Programming in Rewriting Logic.* Computer Science Laboratory, SRI International, United States, 1999. `http://maude.csl.sri.com/`.

[6] E. Contejean, C. Marché, B. Monate, and X. Urbain. CiME, 2000. `http://cime.lri.fr/`.

[7] Coq-Development-Team. *The Coq Proof Assistant Reference Manual – Version 7.4.* INRIA Rocquencourt, France, 2003. `http://coq.inria.fr/`.

[8] T. Coquand. An algorithm for testing conversion in type theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.

[9] J.-C. Filliâtre. Design of a proof assistant : Coq version 7. Technical Report 1369, Université Paris Sud, France, 2000.

[10] H. Kirchner and P.-E. Moreau. Promoting rewriting to a programming language : A compiler for non-deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 11(2) :207–251, 2001.

[11] P.-E. Moreau. *Compilation de règles de réécriture et de stratégies non-déterministes*. PhD thesis, Université Nancy I, France, 1999.

[12] P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *Proceedings of the 12th International Conference on Compiler Construction*, Lecture Notes in Computer Science 2622, 2003.