Lot 5.1

Technologie de vérification

Ajouter la réécriture au noyau de Coq

Prototype d'extension du système Coq

Description :	Nous décrivons comment obtenir, installer et utiliser le prototype de Coq avec récriture basé sur la bibliothèque CiME. Nous donnons ensuite quelques détails sur son implantation: l'architecture de CiME d'une part, les modifications et nouveaux modules apportés à Coq d'autre part.
Auteur(s) :	Frédéric Blanqui
Référence :	Averroes / Lot 5.1 / Fourniture 3 / 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

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

Table des matières

1 Description du prototype **3**
1.1 Téléchargement . . . 3
1.2 Compilation . . . 3
1.3 Documentation . . . 3
1.4 Exemple . . . 3

2 Architecture de CiME **4**

3 Implantation **5**
3.1 Syntaxe . . . 5
3.2 Noyau . . . 6

1 Description du prototype

1.1 Téléchargement

Le prototype est accessible via l'archive CVS de Coq. Pour télécharger les sources, il suffit de faire les commandes unix suivantes. Si c'est la première fois que vous vous connectez au serveur CVS de Coq, faire :

```
cvs -d :pserver:anoncvs@coqcvs.inria.fr:/coq login
```

sans indiquer de mot de passe. Ensuite, faire :

```
cvs -d :pserver:anoncvs@coqcvs.inria.fr:/coq co -r recriture V7
```

crée un répertoire `V7` dans le répertoire courant.

1.2 Compilation

Suivre les indications du fichier `INSTALL`. Essentiellement, cela consiste à configurer le système (commande `configure`), le compiler (commande `make world`) et l'installer (commande `make install`).

La nouveauté par rapport à Coq est qu'une partie des sources de CiME [3] sont incluses avec les sources de Coq (répertoire `cime`), et sont compilées avant de compiler les sources de Coq afin de créer la bibliothèque `libcime` utilisée ensuite dans Coq.

1.3 Documentation

La syntaxe des commandes est décrite dans le fichier `README.REWRITING`. Il y a 4 nouvelles commandes :

- `Symbol` : déclare un objet de manière semblable à `Parameter` ou `Axiom` mais pouvant être ultérieurement défini par des règles de récriture. Il est nécessaire de lui donner une arité et un status (lexicographique, multi-ensemble ou une combinaison des deux). Optionnellement, on peut déclarer le symbole comme étant (anti)monotone par rapport à certains de ses arguments (ces conditions sont vérifiées au moment ou des règles sont données), et plus petit, équivalent ou plus grand que des symboles précédemment déclarés (la précédence sur les symboles est définie et sa bonne fondation vérifiée au fur et à mesure). Enfin, on peut également déclarer un symbole comme étant commutatif (C) ou associatif-commutatif (AC).
- `Rules` : permet de rajouter des règles dans l'environnement, en déclarant le type des variables libres. Le détail des conditions qui sont vérifiées est décrit dans [1]. En particulier, les membres gauches des règles doivent être algébriques et linéaires, les appels récursifs doivent se faire sur des arguments structurellement plus petits que les arguments du membres gauches (modulo le status du symbole ainsi défini), et les règles doivent préserver la confluence locale du système modulo les symboles C ou AC (propriété vérifiée par CiME).
- `Simpl_rew` : tactique de base semblable a `Simpl` mais n'utilisant que les règles de récriture présente dans l'environnement.
- `Print Rules` : affiche la liste des règles présentes dans l'environnement.

1.4 Exemple

```
Symbol plus AC : nat->nat->nat.

Rules [x,y:nat] {
  (plus O x) => x;
  (plus (S x) y) => (S (plus x y))
}.
```

```
Symbol mult AC > plus : nat->nat->nat.

Rules [x,y,z:nat] {
  (mult O x) => O;
  (mult (S x) y) => (plus y (mult x y));
  (mult (plus x y) z) => (plus (mult x z) (mult y z))
}.
```

2 Architecture de CiME

Pour réaliser dans le noyau les récritures, nous avons utilisé la version de CiME [3] du 30 juin 2002, dernière version de CiME avant l'introduction de “hash-consing”. Son adaptation à la dernière version de CiME ne devrait pas poser de difficultés.

Le seul fichier de CiME que nous ayons modifié est `orderings/finite_orderings` (renommé en `orderings/finite_ord`) : on a éliminé le polymorphisme du type de module `Finite_ordering` et du foncteur `Make`.

CiME admet de nombreux modules. Les répertoires de CiME qui nous sont utiles sont :
- `compat` : structures de données générales
- `completion` : calcul des paires critiques, confluence et complétion
- `dioph_caml` : résolution d'équations diophantiennes
- `integer_solver` : arithmétique de Presburger
- `matching` : filtrage (avec listes “paresseuses”)
- `orderings` : généralités sur les ordres
- `rewriting` : récriture
- `standard_matching_and_unification` : filtrage et unification syntaxique
- `strategies` : stratégie “innermost”
- `terms` : signatures et termes
- `unification` : unification modulo différentes théories équationnelles
- `words` : récriture sur les mots

Nous avons écrit un ensemble d'outils qui nous permettent de ne récupérer que les fichiers qui sont utiles à l'extension de Coq :
- `cime/Makefile.dev` : un Makefile à inclure dans le Makefile principal à la place de `Makefile.cime` qui permet de travailler avec une autre version de CiME en instanciant la variable `CIMEHOME`. La commande `make build-release` permet également de faire une copie des fichiers nécessaires de `$CIMEHOME` dans `cime` en vue de faire une release.
- `dev/copy-cime` : script réalisant la copie des fichiers de `$CIMEHOME` à `cime`.
- `dev/get_files` : programme renvoyant sur `stdout` toutes les sous-chaînes des chaînes de caractères lues sur `stdin` filtrant l'expression régulière donnée en argument.
- `dev/order_deps` : lit sur `stdin` un ensemble de dépendances au format de `make` (fichier1 : fichier2 .. fichier$k$, $k \geq 0$) et renvoie sur `stdout`, dans l'ordre de compilation (de celui qui a le moins de dépendance à celui qui en a le plus) la liste des fichiers dont dépend les fichiers donnés en argument. Il ne renvoit rien s'il y a un cycle.

C'est ainsi que la version actuelle du prototype utilisent 89 fichiers CiME, soit environ 22000 lignes de code, alors que le noyau de Coq ne contient qu'environ 10000 lignes de code. Cependant, seulement une moitié environ est vraiment utile au prototype. Par exemple, du fait que la procédure principale d'unification de CiME prend en argument le type de théorie équationnelle modulo laquelle l'unification doit être réalisée, et que ce type peut prendre 8 valeurs différentes (Empty, C, AC, ACU, ACI, AG, ACUN, BR) avec, pour chaque théorie, des développements spécifiques, le prototype, qui n'utilise que Empty, C et AC, dépend inutilement du code pour les autres théorie. En récrivant une petite partie de CiME, il est donc possible de réduire la taille du code de CiME nécessaire au prototype, chose qu'à commencer à entreprendre Evelyne Contejean. Cependant, si on veut un jour disposer dans Coq des autres théories équationnelles, il faudra bien inclure tout le code précédent.

Les points d'entrée dans CiME qui nous sont utiles sont :
- `terms/signatures` : défini ce qu'est une signature.
- `terms/variables` : défini le type (abstrait) des variables.
- `terms/gen_terms` : défini le type des termes. Dans CiME, les arguments d'un symbole sont représentés par une liste alors que, dans Coq, ils sont représentés par un tableau. De plus, pour faire le filtrage modulo AC de manière efficace, les termes sont "aplatis" et les arguments triés (en prenant un ordre total sur les variables) : $+(+yx)z$ est représenté par $+xyz$ si $x < y < z$.
- `terms/rewrite_rules` : défini le type des règles de récriture.
- `rewriting/naive_dnet` : défini une fonction qui, à partir d'un ensemble de règles de récriture, construit ("compile" dans CiME) une structure ("discrimination net") qui permet de faire le filtrage de manière efficace.
- `rewriting/rewriting` : défini une fonction testant la réductibilité (modulo AC) en tête.
- `strategies/innermost` : défini une fonction qui, à partir d'une fonction testant la réducbilité en tête, calcule la forme normale d'un terme selon la stratégie "innermost", *i.e.* en normalisant d'abord les arguments.
- `completion/abstract_rewriting` : défini un type de module `AbstractRewriting` réunissant la plupart des types et fonctions de base nécessaires à la complétion (*e.g.* "compilation", normalisation ou calcul des paires critiques), et différents foncteurs qui, à partir d'un type pour les symboles, fournit un module de type `AbstractRewriting`.
- `completion/confluence` : défini un foncteur qui, à partir d'un module de type `AbstractRewriting`, fournit une fonction testant la confluence locale (appelée "confluence" dans CiME).
- `orderings/orderings_generalities` : généralités sur les ordres.
- `orderings/finite_orderings` modifié et renommé en `orderings/finite_ord` : définition et extension de précédences.

3 Implantation

L'architecture de Coq est présentée dans [2]. Nous donnons ici quelques détails sur les fichiers ajoutés ou modifiés.

3.1 Syntaxe

Séquence des fichiers modifiés pour étendre la syntaxe de Coq :
- `toplevel/vernacexpr.ml` : types pour représenter les commandes rentrées par l'utilisateur.
- `parsing/g_vernac.ml4` : grammaire.
- `toplevel/vernacentries` : pour chaque commande possible, appelle la fonction d'interpretation correspondante dans `toplevel/command`.
- `toplevel/command` : interprète les commandes à l'aide entre autres de :
  - `interp/topconstr` : type des termes rentrées par l'utilisateur,
  - `interp/constrintern` : transforme les `constr_expr` en `constr`.

  et appelle les fonctions de déclaration nécessaires dans `library/declare`.
- `library/declare` : définie les fonctions permettant de charger dans l'environnement courant un objet, appelle `library/global`.
- `library/global` : environnement courant, appelle les fonctions de `kernel/safe_typing` définies pour les environnements bien typés.
- `kernel/safe_typing` : environnements bien typés, appelle les fonctions de typage de `kernel/term_typing` et les fonctions sur les environnements quelconques de `kernel/environ`.

- `kernel/environ` : type des environnements.
- `kernel/term_typing` : appelle les fonctions vérifiant la bonne formation d'un objet.

3.2 Noyau

Pour le prototype, nous avons en particulier rajouter dans `kernel` les modules suivants :

- `precedence` (Figure 1) : module permettant de définir et d'étendre des précédences sur le type `kernel_name` des identificateurs de Coq. Implanté à l'aide des fichiers CiME `cime/orderings/finite_ord` (modifié) et `cime/orderings/orderings_generalities`.
- `symbol` (Figure 2) : module définissant les caractéristiques d'un symbole (théorie équationnelle, statut, arguments (anti)monotones, précédence, etc.).
- `positivity` (Figure 3) : fonctions indiquant si un symbole ou un inductif apparait positivement dans un `constr`.
- `cime` (Figure 4) : interface avec CiME. Un type environnement propre à cette interface est définit auquel il faut déclarer les ajouts de symbole, d'inductif ou de règles. Il permet de maintenir une structure de données propre à CiME pour optimiser le filtrage ("discrimination nets" dans `cime/rewriting/naive_dnet`). Par ailleurs, une fonction est fournie pour vérifier la confluence locale d'un ensemble de règles, et une autre pour calculer la forme normale d'un `constr` par rapport aux règles de récriture déclarées dans l'environnement.
- `rules` (Figure 5) : fonctions vérifiant si une déclaration de symbole ou de règles est admissible.

Références

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

[2] F. Blanqui. Prototype d'extension du système Coq. Averroes, lot 5.1, fourniture 2, 2004.

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

FIG. 1 – Interface du module `precedence`

```
type precedence (* on kernel_name *)

type result = Equivalent | Smaller | Greater | Uncomparable

(* precedence where equal names are Equivalent
   and distinct names are Uncomparable *)
val empty_prec : precedence

val compare : precedence -> kernel_name -> kernel_name -> result

val is_equiv : precedence -> kernel_name -> kernel_name -> bool
val is_smaller : precedence -> kernel_name -> kernel_name -> bool
val is_greater : precedence -> kernel_name -> kernel_name -> bool
val is_smaller_eq : precedence -> kernel_name -> kernel_name -> bool
val is_greater_eq : precedence -> kernel_name -> kernel_name -> bool
val are_uncomparable : precedence -> kernel_name -> kernel_name -> bool

exception Incompatible

val add_equiv : precedence -> kernel_name -> kernel_name -> precedence
val add_greater : precedence -> kernel_name -> kernel_name -> precedence
```

FIG. 2 – Interface du module `symbol`

```
(* equational theories *)
type eqth = Free | C | AC

(* statuses *)
type status = Mul | Lex | RLex | Comb of (int list) list

val select : 'a array -> (int list) list -> ('a list) list
val select_from_status : 'a array -> status -> ('a list) list

(* kinds of occurrences *)
type delta = Pos | Neg | Nul

val opp : delta -> delta

(* termination criteria *)
type termin = General_Schema

(* information about symbols *)
type symbol_info = {
  symb_arity : int;
  symb_eqth : eqth;
  symb_status : status;
  symb_mons : delta array;
  symb_termin : termin;
  symb_acc : bool array;
  symb_prec_defs : prec_def list;
}
```

FIG. 3 – Interface du module `positivity`

```
val occur_const : env -> symbol -> delta -> constr -> bool

val occur_mind : env -> mutual_inductive -> delta -> constr -> bool
```

FIG. 4 – Interface du module `cime`

```
type env

val empty_env : env

val add_symbol : constant_body KNmap.t -> env -> env
val add_inductive : mutual_inductive_body KNmap.t -> env -> env
val add_rules : rules_body -> env -> env

val is_locally_confluent : env -> (constr * constr) list -> bool

val normalize : env -> constr -> constr option
```

FIG. 5 – Interface du module `rules`

```
val check_symbol : env -> types -> symbol_entry -> symbol_info

val check_rules : env -> rules_entry -> rules_body
```