Recherche dans une bibliothèque de preuves Coq en utilisant le type et modulo isomorphismes

David Delahaye* Benjamin Werner† Roberto Di Cosmo‡

20 octobre 1997

Résumé

Nous proposons une méthode pour rechercher un identificateur dans une bibliothèque de preuves Coq en utilisant le type (spécification totale ici) comme motif. Elle se base sur la notion d'isomorphisme de type développée dans le cadre de la programmation fonctionnelle. Nous introduisons une théorie caractérisant une certaine classe d'isomorphismes de Coq et qui est une généralisation de l'axiomatisation pour le λ-calcul typé associé aux Catégories Cartésiennes Fermées (CCC) en prenant en compte le polymorphisme et les types dépendants. De là, nous montrons quelques propriétés de base telles la correction, l'incomplétude (par rapport au calcul inductif des constructions), la décidabilité et mettons en évidence la nécessité de propager les termes de conversion dans les preuves d'isomorphismes (au niveau du passage au contexte). Ainsi, notre implémentation, dans un souci de performances, caractérise le sous-ensemble congruent de cette théorie permettant d'ignorer les convertisseurs.

1 Introduction

1.1 Motivations

Le problème consistant à retrouver facilement des composants logiciels dans une bibliothèque est fondamental. Il est relié à la réutilisabilité du code. En effet, un code réutilisable est un code non seulement suffisamment générique mais aussi pouvant être retrouvé rapidement lorsque l'on en a besoin. Cette seconde composante est souvent négligée car on la considère, à tort, comme peu théorique et peu informatique. Résultat, les outils de recherche actuels ne sont rien d'autre que des index d'identificateurs dans lesquels on prie souvent pour que le nom donnée à la fonction recherchée soit assez explicite pour la trouver rapidement. Ainsi et de manière générale, on perd du temps dans cette quête approximative qui, si elle se révèle infructueuse, oblige l'utilisateur à réécrire du code peut-être existant. Un exemple intéressant est celui de la fonction CAML ([PL66], [PW94], [XL94]) `list_it` (tableau 1) qui ne porte pas moins de quatre noms différents dans d'autres versions de ML.

On l'aura compris, un identificateur est parfaitement insuffisant pour permettre une recherche performante. L'idée est donc de prendre le type comme motif de la recherche et d'effectuer les comparaisons modulo une certaine équivalence. Ainsi, se pose la question suivante: quand deux

*INRIA-Rocquencourt, domaine de Voluceau, B.P. 105, 78153 Le Chesnay Cedex, France. David.Delahaye@inria.fr.

†INRIA-Rocquencourt, domaine de Voluceau, B.P. 105, 78153 Le Chesnay Cedex, France. Benjamin.Werner@inria.fr.

‡ENS-Ulm, DMI, 45 rue d'Ulm, 75230 Paris Cedex 05, France. dicosmo@dmi.ens.fr.

Langage	Nom	Type
ML (LCF)	itlist	$\forall X.\forall Y.(X \rightarrow Y \rightarrow Y) \rightarrow List(X) \rightarrow Y \rightarrow Y$
CAML	list_it	$\forall X.\forall Y.(X \rightarrow Y \rightarrow Y) \rightarrow List(X) \rightarrow Y \rightarrow Y$
Haskell	foldl	$\forall X.\forall Y.(X \rightarrow Y \rightarrow X) \rightarrow X \rightarrow List(Y) \rightarrow X$
SML of New Jersey	fold	$\forall X.\forall Y.(X \times Y \rightarrow Y) \rightarrow List(X) \rightarrow Y \rightarrow Y$
Edinburgh SML	fold_left	$\forall X.\forall Y.(X \times Y \rightarrow Y) \rightarrow Y \rightarrow List(X) \rightarrow Y$

Tableau 1 : Le `list_it` en ML.

types sont-ils équivalents? Il n'y a évidemment pas de réponse standard à cette question. Tout dépend de ce que l'on veut identifier. Un choix un peu brutal et pas très utile pourrait être de prendre l'égalité structurelle. Mais cette option est un peu trop restrictive et pour s'en apercevoir, il suffit de reprendre l'exemple de `list_it` qui possède au total quatre types distincts dans les différents ML. L'équivalence doit donc être plus large. Les travaux de Mikael Rittri ([MR91]) ont mis en évidence que la notion la plus propice pour la recherche en bibliothèques est celle de types isomorphes. Cette notion, formalisée et étudiée pendant de nombreuses années par Roberto Di Cosmo, Kim Bruce, Giuseppe Longo et Sergei Soloviev a déjà été mise en œuvre dans un outil baptisé CamlSearch de Jérôme Vouillon et Julien Jalon sous la direction de Roberto Di Cosmo au LIENS en 1994. Ce programme, externe à CAML, permet d'entrer un type avec métavariables (il réalise donc du filtrage) et effectue la recherche modulo une théorie Th^{ML} prouvée complète pour ML. L'objectif est de réaliser un tel outil pour Coq ([BW94], [BB$^+$97]).

1.2 Plan

Cet article s'articule en quatre parties distinctes. Nous exposerons, tout d'abord, les notions de base relatives aux isomorphismes de types afin d'établir, par la suite, une théorie pour Coq. Puis, nous parlerons de l'implémentation en isolant bien les choix et restrictions qui ont pu être faits au cours du codage de la procédure de décision et en abordant brièvement le principe de la recherche. Enfin, des jeux d'essais judicieusement construits permettront non seulement d'esquisser la correction du prototype par rapport à la théorie mais aussi de connaître ses performances, ce qui est salutaire pour son utilisation future.

2 Isomorphismes de types

Nous allons nous baser, ici, sur les travaux de Roberto Di Cosmo, Kim Bruce, Giuseppe Longo et Sergei Soloviev sur les types isomorphes. En particulier, nous allons donner toutes les définitions et notions, pour la plupart formalisées dans [RC95], qui sont nécessaires à l'élaboration d'une théorie pour un langage en général et pour Coq en particulier.

2.1 Types isomorphes

De manière naturelle, on peut dire que deux types sont isomorphes si il existe deux fonctions de conversion définissables dans le langage qui permettent de passer de l'un à l'autre et vice-versa. Plus précisément, on a la définition suivante:

Définition 2.1 (Isomorphismes définissables, termes intervertibles) *Deux types A et B sont isomorphes de manière définissable ($A \cong_d B$) si et seulement si il existe les fonctions (λ-*

termes) $M : A \rightarrow B$ *et* $N : B \rightarrow A$ *telles que* $M \circ N =_{\mathcal{L}} I_B$ *et* $N \circ M =_{\mathcal{L}} I_A$ *où* I_A *et* I_B *sont* $\lambda x : A.x$ *et* $\lambda x : B.x$*, les identités de type A et B. Les termes M et N sont dits intervertibles. On note aussi* $M : A \cong_d B : N$ *quand on veut rendre les termes intervertibles associés explicites.*

Ce point vue est essentiellement syntaxique. Il oblige à exhiber deux fonctions dans le langage. Toutefois, l'égalité $=_{\mathcal{L}}$ servant à comparer $M \circ N$ et I_B ainsi que $N \circ M$ et I_A n'est pas purement syntaxique et peut être plus large que les règles d'exécution (ou de réduction) du langage. Cette égalité dépend beaucoup du langage mais de manière générale, elle contient les axiomes suivants:

(β)	$(\lambda x.M)N = M[x \leftarrow N]$
(η)	$\lambda x.M\ x = M$ si $x \notin FV(M)$
(π^n)	$p_{c,i}(C\ M_1 ... M_n) = M_i$, i=1..n
(SP^n)	$(C\ (p_{c,1}\ M)...(p_{c,n}\ M)) = M$
(top)	$M = tt$ si $M : unit$

$FV(M)$ représente l'ensemble des variables libres de M. $p_{c,i}$ est la ième-projection associée au type inductif non effectif et sans arité dont C est l'unique constructeur (se reporter à la page 4 pour plus d'informations sur de tels types). tt est l'unique constructeur d'arité 0 du type inductif unit.

Au-delà de cette notion syntaxique d'isomorphisme de types, on peut adopter une vision plus sémantique en se penchant sur les modèles du langage. Si on peut donner au langage une sémantique ensembliste (les types sont des ensembles structurés et les termes de type $A \rightarrow B$ des morphismes préservant la structure de l'interprétation de A vers celle de B) telle que le typage soit correct (les programmes bien typés ne sont jamais bloqués et dans le cas particulier de Coq, les programmes bien typés terminent toujours) alors on peut utiliser la définition suivante:

Définition 2.2 (isomorphismes sémantiques) *Deux types A et B sont isomorphes dans un modèle* $\mathcal{M}$ *spécifique si leurs interprétations sont isomorphes dans* $\mathcal{M}$ *au sens traditionnel (i.e., il existe dans le modèle deux fonctions intervertibles f et g entre eux) et on écrit alors* $\mathcal{M} \models A \cong_s B$*. Deux types A et B seront sémantiquement isomorphes, noté* $A \cong_s B$*, si on a* $\mathcal{M} \models A \cong_s B$ *pour tout modèle* $\mathcal{M}$ *du calcul.*

Cette méthode sémantique semble peu utilisable en pratique dans la mesure où l'on doit énumérer tous les modèles du langage (il y en a en général une infinité) pour y exhiber les termes de conversion. Préférer la notion syntaxique peut donc paraître une sage décision. Toutefois, dans certains cas, on peut s'en tirer avec succès comme pour le λ-calcul simplement typé avec produit et unit où l'on peut montrer, via les ccc's et le résultat de complétude de Soloviev ([SS83]), que les seuls isomorphismes sémantiques (et donc syntaxiques) sont les suivants:

1. $A * B = B * A$
2. $A * (B * C) = (A * B) * C$
3. $(A * B) \rightarrow C = A \rightarrow (B \rightarrow C)$
4. $A \rightarrow (B * C) = (A \rightarrow B) * (A \rightarrow C)$
5. $A * unit = A$
6. $A \rightarrow unit = unit$
7. $unit \rightarrow A = A$

Cela pose le problème de la correspondance des classes d'isomorphismes syntaxiques et sémantiques. Existe-t-il des langages où les deux classes sont identiques? La réponse est oui avec par exemple le cas que nous venons d'étudier mais aussi le λ-calcul simplement typé, le sytème F, le λ-calcul simplement typé avec produit, le λ-calcul simplement typé avec unit,...

2.2 Caractérisation des classes d'isomorphismes

Une fois que l'on sait que, dans un calcul donné, les isomorphismes syntaxiques et sémantiques se correspondent, il est plus facile de caractériser complètement cette classe unique en se servant de l'aspect syntaxique. En fait, on peut toujours le faire dans le cas général, mais, à un moment ou à un autre, on sera obligé soit de prouver la correspondance des classes sémantiques et syntaxiques si c'est possible, soit de chercher une autre caractérisation pour la partie sémantique.

Pour se lancer dans la caratérisation syntaxique, on peut voir deux méthodes. La première consiste à identifier une certaine forme de termes non typés et de faire le lien avec les termes typés intervertibles. La deuxième consiste tout simplement à construire une théorie pour juger de l'équivalence de deux types. Cette seconde méthode est générale et s'applique aussi bien au cas syntaxique que sémantique. Pour l'aspect sémantique, c'est un bon moyen de se ramener à de la syntaxe pure tandis que pour l'aspect syntaxique c'est un outil fort pratique pour construire progressivement (lors de la preuve) les termes intervertibles. Une fois la théorie réalisée, il est nécessaire de se poser les questions de correction, de complétude (par rapport à la notion d'isomorphismes considérée) et de décidabilité (de l'équivalence).

3 Une théorie pour Coq

Notre objectif est ici de donner la théorie que l'on a considéré pour Coq, de justifier ces axiomes et de s'intéresser à quelques propriétés de ce système de règles, en mettant systématiquement l'accent sur les différences avec les outils utilisés par CamlSearch lorsque cela a un sens.

3.1 Les équations

La théorie que nous allons donner pour Coq s'appuie essentiellement sur celle de CAML (voir [RC95]) en apportant deux généralisations importantes qui vont vite compliquer le système et ses propriétés. La première généralisation consiste à prendre en compte les types dépendants. On substitue Π à $\rightarrow$. La deuxième généralisation est de considérer le produit cartésien dépendant (le $\exists$). On remplace $*$ par Σ. Mais ce Σ ne correspond pas uniquement à ex (dans Logic), à sig ou à sigS (dans Specif). Il concerne également tous les types inductifs à un constructeur qui n'ont pas d'arité au sens où ils n'introduisent pas de contraintes sur leurs paramètres comme eq (dans Logic). Lorsque les constructeurs de tels types ont plus de deux arguments (en plus des paramètres), leurs types inductifs respectifs sont traités comme des Σ emboîtés (avec une associativité à droite au même titre que $\rightarrow$). Par ailleurs, en vue d'une justification syntaxique des isomorphismes que l'on va considérer, il est important de ne traiter que les types inductifs pour lesquels des projecteurs peuvent être écrits dans le langage Coq. Par exemple, `ex` dans Logic, ne peut pas être pris en compte. On va donc considérer non pas purement les types de Coq, mais les types de Coq après une réécriture de ces types inductifs. Th^{Coq} s'exprime alors de la manière suivante:

1. $A = B$ si $A \xrightarrow{\beta\iota} B$
2. $\Sigma x : A.B = \Sigma y : A.B[x \leftarrow y]$ si $y \notin FV(\Sigma x : A.B)$
3. $\Pi x : A.B = \Pi y : A.B[x \leftarrow y]$ si $y \notin FV(\Pi x : A.B)$
4. $\Sigma x : A.B = \Sigma x : B.A$ si $x \notin FV(A, B)$
5. $\Sigma x : (\Sigma y : A.B).C = \Sigma x : A.\Sigma y : B[y \leftarrow x].C[x \leftarrow (x, y)]$
6. $\Pi x : (\Sigma y : A.B).C = \Pi x : A.\Pi y : B[y \leftarrow x].C[x \leftarrow (x, y)]$
7. $\Pi x : A.\Sigma y : B.C = \Sigma y : (\Pi x : A.B).(\Pi x : A.C[y \leftarrow (y\ x)]$

8. $\Sigma x : A.unit = A$
9. $\Sigma x : unit.A = A[x \leftarrow tt]$
10. $\Pi x : A.unit = unit$
11. $\Pi x : unit.A = A[x \leftarrow tt]$

3.2 Propriétés

Nous allons nous attacher maintenant à établir les propriétés de base de Th^{Coq}. Dans la plupart des cas, nous ne donnerons aucune démonstration mais le lecteur désireux de connaître les preuves correspondantes pourra se référer à [DD97].

3.2.1 Passage au contexte

Th^{Coq} n'est pas une pré-congruence au sens de la définition suivante:

Définition 3.1 (Pré-congruence) *Une relation $\mathcal{R}$ sur un ensemble $\mathcal{E}$* défini inductivement *est une pré-congruence si et seulement si:*

$$\forall a, b \in \mathcal{E},\ \forall C \in \mathcal{C}[E],\ a\ \mathcal{R}\ b \Rightarrow (C[a] \in \mathcal{E} \Rightarrow C[a]\ \mathcal{R}\ C[b]).$$

($\mathcal{C}[E]$ représente l'ensemble des contextes sur E).

Toutefois, avec n'importe quel contexte, on peut réussir à conserver l'égalité à condition que les équations issues de Th^{Coq} puissent se justifier syntaxiquement et que l'on modifie le contexte. On posera donc:

Définition 3.2 (Pré-congruence faible) *Une relation $\mathcal{R}$ sur un ensemble $\mathcal{E}$* défini inductivement *est une pré-congruence faible si et seulement si:*

$$\forall a, b \in \mathcal{E},\ \forall C \in \mathcal{C}[E],\ a\ \mathcal{R}\ b \Rightarrow (C[a] \in \mathcal{E} \Rightarrow \exists C' \in \mathcal{C}[E],\ C[a]\ \mathcal{R}\ C'[b]).$$

($\mathcal{C}[E]$ représente l'ensemble des contextes sur E).

Théorème 3.3 *Th^{Coq} est une pré-congruence faible.*

La preuve de ce théorème nous permet de modifier la couche de logique équationnelle à ajouter aux axiomes donnés en 3.1, au niveau du passage au contexte.

3.2.2 Correction

On peut montrer que Th^{Coq} est une théorie correcte pour la classe des isomorphismes syntaxiques et de fait pour la classe des isomorphismes sémantiques. Ces propriétés s'expriment par le théorème et le corollaire suivants:

Théorème 3.4 (Correction syntaxique) *Th^{Coq} est une théorie correcte vis-à-vis de la classe des isomorphismes syntaxiques.*

Corollaire 3.5 (Correction sémantique) *Th^{Coq} est une théorie correcte vis-à-vis de la classe des isomorphismes sémantiques.*

La preuve du théorème 3.4 nous permet maintenant d'exprimer de manière complète Th^{Coq} comme nous le montre les tableaux 2 et 3.

$$\begin{array}{l} A = \\ \begin{pmatrix} \lambda x : A.x \\ \lambda x : B.x \end{pmatrix} \\ B \text{ si } A \stackrel{\beta\iota}{\longrightarrow} B \ (Th^{Coq}_{\beta\iota Rd}) \end{array}$$

$$\begin{array}{l} \Sigma x : A.B = \\ \begin{pmatrix} \lambda c : (\Sigma x : A.B).c \\ \lambda c : (\Sigma y : A.B[x \leftarrow y]).c \end{pmatrix} \\ \Sigma y : A.B[x \leftarrow y] \text{ si } x \notin FV(\Sigma x : A.B) \ (Th^{Coq}_{\Sigma Alp}) \end{array}$$

$$\begin{array}{l} \Pi x : A.B = \\ \begin{pmatrix} \lambda f : (\Pi x : A.B).\lambda y : A.(f\ y) \\ \lambda f : (\Pi y : A.B[x \leftarrow y]).\lambda x : A.(f\ x) \end{pmatrix} \\ \Pi y : A.B[x \leftarrow y] \text{ si } x \notin FV(\Pi x : A.B) \ (Th^{Coq}_{\Pi Alp}) \end{array}$$

$$\begin{array}{l} \Sigma x : A.B = \\ \begin{pmatrix} \lambda c : (\Sigma x : A.B).(\pi_2\ c, \pi_1\ c) \\ \lambda c : (\Sigma x : B.A).(\pi_2\ c, \pi_1\ c) \end{pmatrix} \\ \Sigma x : B.A \text{ si } x \notin FV(A, B) \ (Th^{Coq}_{\Sigma Com}) \end{array}$$

$$\begin{array}{l} \Sigma x : (\Sigma y : A.B).C = \\ \begin{pmatrix} \lambda z : (\Sigma x : (\Sigma y : A.B).C).(\pi_1\ (\pi_1\ z), (\pi_2\ (\pi_1\ z), \pi_2\ z)) \\ \lambda z : (\Sigma x : A.\Sigma y : B[y \leftarrow x].C[x \leftarrow (x, y)]).((\pi_1\ z, \pi_1\ (\pi_2\ z)), \pi_2\ (\pi_2\ z)) \end{pmatrix} \\ \Sigma x : A.\Sigma y : B[y \leftarrow x].C[x \leftarrow (x, y)] \ (Th^{Coq}_{\Sigma Ass}) \end{array}$$

$$\begin{array}{l} \Pi x : (\Sigma y : A.B).C = \\ \begin{pmatrix} \lambda f : (\Pi x : (\Sigma y : A.B).C).\lambda x : A.\lambda y : B[y \leftarrow x].f\ (x, y) \\ \lambda f : (\Pi x : A.\Pi y : B[y \leftarrow x].C[x \leftarrow (x, y)]).\Pi x : (\Sigma y : A.B).f\ (\pi_1\ x)\ (\pi_2\ x) \end{pmatrix} \\ \Pi x : A.\Pi y : B[y \leftarrow x].C[x \leftarrow (x, y)] \ (Th^{Coq}_{\Sigma Cur}) \end{array}$$

$$\begin{array}{l} \Pi x : A.\Sigma y : B.C = \\ \begin{pmatrix} \lambda f : (\Pi x : A.\Sigma y : B.C).(\Pi x : A.(\pi_1\ (f\ x)), \Pi x : A.(\pi_2\ (f\ x))) \\ \lambda c : (\Sigma y : (\Pi x : A.B).(\Pi x : A.C[y \leftarrow (y\ x)]).\lambda x : A.((\pi_1\ c)\ x, (\pi_2\ c)\ x) \end{pmatrix} \\ \Sigma y : (\Pi x : A.B).(\Pi x : A.C[y \leftarrow (y\ x)] \ (Th^{Coq}_{\Pi Dis}) \end{array}$$

$$\begin{array}{l} \Sigma x : A.unit = \\ \begin{pmatrix} \lambda c : (\Sigma x : A.unit).\pi_1\ c \\ \lambda x : A.(x, tt) \end{pmatrix} \\ A \ (Th^{Coq}_{\Sigma Unr}) \end{array}$$

$$\begin{array}{l} \Sigma x : unit.A = \\ \begin{pmatrix} \lambda c : (\Sigma x : unit.A).(\pi_2\ c) \\ \lambda a : A[x \leftarrow tt].(tt, a) \end{pmatrix} \\ A[x \leftarrow tt] \ (Th^{Coq}_{\Sigma Unl}) \end{array}$$

$$\begin{array}{l} \Pi x : A.unit = \\ \begin{pmatrix} \lambda f : (\Pi x : A.unit).tt \\ \lambda u : unit.\lambda x : A.tt \end{pmatrix} \\ unit \ (Th^{Coq}_{\Pi Unr}) \end{array}$$

$$\begin{array}{l} \Pi x : unit.A = \\ \begin{pmatrix} \lambda f : (\Pi x : unit.A).f\ tt \\ \lambda a : A[x \leftarrow tt].\lambda x : unit.a \end{pmatrix} \\ A[x \leftarrow tt] \ (Th^{Coq}_{\Pi Unl}) \end{array}$$

Tableau 2 : Les axiomes de Th^{Coq}

$$\frac{}{A =_{\left(\begin{smallmatrix} \lambda x : A.x \\ \lambda x : A.x \end{smallmatrix}\right)} A}\; Th^{Coq}_{Ref}$$

$$\frac{A =_{\left(\begin{smallmatrix} \sigma \\ \tau \end{smallmatrix}\right)} B}{B =_{\left(\begin{smallmatrix} \tau \\ \sigma \end{smallmatrix}\right)} A}\; Th^{Coq}_{Sym}$$

$$\frac{A =_{\left(\begin{smallmatrix} \sigma \\ \tau \end{smallmatrix}\right)} B \quad B =_{\left(\begin{smallmatrix} \sigma' \\ \tau' \end{smallmatrix}\right)} C}{A =_{\left(\begin{smallmatrix} \sigma \circ \sigma' \\ \tau \circ \tau' \end{smallmatrix}\right)} C}\; Th^{Coq}_{Trs}$$

$$\frac{A =_{\left(\begin{smallmatrix} \sigma \\ \tau \end{smallmatrix}\right)} A'}{\Pi x : A.B =_{\left(\begin{smallmatrix} \lambda f : (\Pi x : A.B).\lambda x : A'.f\ (\tau\ x) \\ \lambda f : (\Pi x : A'.B[x \leftarrow (\tau\ x)]).\lambda x : A.f\ (\sigma\ x) \end{smallmatrix}\right)} \Pi x : A'.B[x \leftarrow (\tau\ x)]}\; Th^{Coq}_{\Pi Cnl}$$

$$\frac{A =_{\left(\begin{smallmatrix} \sigma \\ \tau \end{smallmatrix}\right)} A'}{\Sigma x : A.B =_{\left(\begin{smallmatrix} \lambda c : (\Sigma x : A.B).(\sigma\ (\pi_1\ c),\ \pi_2\ c) \\ \lambda c : (\Sigma x : A'.B[x \leftarrow (\tau\ x)]).(\tau\ (\pi_1\ c),\ \pi_2\ c) \end{smallmatrix}\right)} \Sigma x : A'.B[x \leftarrow (\tau\ x)]}\; Th^{Coq}_{\Sigma Cnl}$$

$$\frac{A =_{\left(\begin{smallmatrix} \sigma \\ \tau \end{smallmatrix}\right)} A'}{\Pi x : B.A =_{\left(\begin{smallmatrix} \lambda f : (\Pi x : B.A).\lambda x : B.\sigma\ (f\ x) \\ \lambda f : (\Pi x : B.A').\lambda x : B.\tau\ (f\ x) \end{smallmatrix}\right)} \Pi x : B.A'}\; Th^{Coq}_{\Pi Cnr}$$

$$\frac{A =_{\left(\begin{smallmatrix} \sigma \\ \tau \end{smallmatrix}\right)} A'}{\Sigma x : B.A =_{\left(\begin{smallmatrix} \lambda c : (\Sigma x : B.A).(\pi_1\ c, \sigma\ (\pi_2\ c)) \\ \lambda c : (\Sigma x : B.A').(\pi_1\ c, \tau\ (\pi_2\ c)) \end{smallmatrix}\right)} \Sigma x : B.A'}\; Th^{Coq}_{\Sigma Cnr}$$

$$\frac{A =_{\left(\begin{smallmatrix} \sigma \\ \tau \end{smallmatrix}\right)} A'}{\Pi x : A.A =_{\left(\begin{smallmatrix} \lambda f : (\Pi x : A.A).\lambda x : A'.\sigma\ (f\ (\tau\ x)) \\ \lambda f : (\Pi x : A'.A').\lambda x : A.\tau\ (f\ (\sigma\ x)) \end{smallmatrix}\right)} \Pi x : A'.A'}\; Th^{Coq}_{\Pi Cnb}$$

$$\frac{A =_{\left(\begin{smallmatrix} \sigma \\ \tau \end{smallmatrix}\right)} A'}{\Sigma x : A.A =_{\left(\begin{smallmatrix} \lambda c : (\Sigma x : A.A).(\sigma\ (\pi_1\ c), \sigma\ (\pi_2\ c)) \\ \lambda c : (\Sigma x : A'.A').(\tau\ (\pi_1\ c), \tau\ (\pi_2\ c)) \end{smallmatrix}\right)} \Sigma x : A'.A'}\; Th^{Coq}_{\Sigma Cnb}$$

Tableau 3 : La logique équationnelle de Th^{Coq}

3.2.3 Complétude

On peut démontrer que Th^{Coq} n'est complète ni pour la classe des isomorphismes syntaxiques ni pour la classes des isomorphismes sémantiques. Ainsi, on a le théorème et le corollaire suivants:

Théorème 3.6 (Incomplétude syntaxique) *Th^{Coq} est une théorie incomplète vis-à-vis de la classe des isomorphismes syntaxiques.*

Corollaire 3.7 (Incomplétude sémantique) *Th^{Coq} est une théorie incomplète vis-à-vis de la classe des isomorphismes sémantiques.*

3.2.4 Décidabilité

On peut montrer que Th^{Coq} est décidable comme l'exprime le théorème suivant:

Théorème 3.8 (Décidabilité) *Th^{Coq} est décidable.*

Pour le prouver, on élabore une procédure correcte, complète (vis-à-vis de Th^{Coq}), et terminante. Cette procédure, que nous appellerons Dec^{Coq}, passe par la construction d'un système de réécriture $\mathcal{R}^{Coq}$ qui se constitue des axiomes de Th^{Coq} (orienté de la gauche vers la droite) privée de $Th^{Coq}_{\Sigma Alp}$, $Th^{Coq}_{\Pi Alp}$, $Th^{Coq}_{\Sigma Com}$, et passe au contexte conformément au tableau 3 ($\mathcal{R}^{Coq}$ n'est donc que faiblement pré-congruent). Les $\mathcal{R}^{Coq}$-formes normales sont des Σ emboîtés et pour chacune des composantes, il n'y a ni β-redex ni Σ ni unit. Dec^{Coq} consiste donc à $\mathcal{R}^{Coq}$-normaliser les deux types, puis à les comparer modulo $Th^{Coq}_{\Sigma Alp}$, $Th^{Coq}_{\Pi Alp}$, $Th^{Coq}_{\Sigma Com}$, $Th^{Coq}_{\Sigma Ass}$ et l'axiome suivant:

$$\begin{array}{l}\Pi x : A.\Pi y : B.C = \\ \left(\begin{array}{l}\lambda f : (\Pi x : A.\Pi y : B.C).\Pi y : B.\Pi x : A.f\ x\ y \\ \lambda f : (\Pi y : B.\Pi x : A.C).\Pi x : A.\Pi y : B.f\ y\ x\end{array}\right) \\ \Pi y : B.\Pi x : A.C \text{ si } x \notin FV(B) \text{ et } y \notin FV(A)\ (Th^{Coq}_{\Pi Swa})\end{array}$$

Notons que cet axiome se déduit de Th^{Coq} par $Th^{Coq}_{\Sigma Com}$ et $Th^{Coq}_{\Sigma Cur}$.

4 Implémentation

Nous allons étudier quelle implémentation on peut faire de Dec^{Coq} et comment s'effectue la recherche dans Coq.

4.1 La procédure de décision

L'implémentation de Dec^{Coq} passe inévitablement par l'implémentation de $\mathcal{R}^{Coq}$. Or, comme on a pu le voir précédemment, $\mathcal{R}^{Coq}$ n'est pas pré-congruent (en fait il l'est seulement faiblement), ce qui nous oblige à calculer et à mémoriser systématiquement les fonctions de conversion à chaque étape de réécriture. Ce type de réécriture n'est pas forcément difficile à mettre en œuvre (on sait bien ce qu'il y a à faire et comment le faire) mais le processus est très lourd. On ne peut donc pas se permettre d'implémenter un tel système qui correspond certes exactement à Th^{Coq} mais qui prendra probablement beaucoup de place (les termes de conversion grossissent lors de la réécriture et ne sont réduits qu'une fois appliqués) et surtout beaucoup de temps à l'exécution (n'oublions pas que la procédure est destinée à être appliquée aux quelques centaines de types de la bibliothèque Coq) si bien qu'il pourra décourager l'utilisateur.

Un compromis reste donc à trouver essentiellement entre la possibilité d'identifier un maximum de types vis-à-vis de Th^{Coq} et le temps d'exécution. L'idée, pour laquelle nous avons opté, consiste à se débarasser des termes intervertibles et à garder la partie congruente de $\mathcal{R}^{Coq}$. Ainsi, Dec^{Coq} ne change pas foncièrement si ce n'est que la réécriture est contrainte à se faire soit à la racine, soit à droite d'un lieur pour toutes les règles sauf pour la $\beta\iota$-réduction (qui peut donc se faire n'importe où) et ce, dans l'ordre partiel curryfication, règles pour *unit* et distribution.

4.2 La recherche

La recherche s'effectue en scannant un environnement Coq. Il y a donc deux outils indépendants. Le premier est intégré au toplevel de Coq (`coqtop`) et se contente de faire la recherche dans le contexte courant. Le deuxième (`coq_search2_top`) contient toute la bibliothèque (sous une forme allégée en chargeant et filtrant les fichiers objets `.vo`) et est exclusivement dédié à la recherche.

5 Jeu d'essais

On va lancer `coq_search2_top` et imaginer une requête possible d'un utilisateur:

```
Welcome to Coq_Search2 V1.0 (July 10 1997)

Searching for .vi ... Done
Loading .vi ... Done
Normalizing types ... Done

Coq_Search2 < Time.

Coq_Search2 < Search2 (b,a:nat)(gt b O)->{q:nat&{r:nat|
Coq_Search2 < a=(plus (mult q b) r)/\(gt b r)}}.
#Div#--* [div1 :
(b:nat)(gt b O)->(a:nat)(#GENTERM (MUTIND {#Div#diveucl.cci} {0}) a b)]
#Div#--* [div2 :
(b:nat)(gt b O)->(a:nat)(#GENTERM (MUTIND {#Div#diveucl.cci} {0}) a b)]
#Euclid_proof#--* [eucl_dev : (b:nat)(gt b O)->(a:nat)(diveucl a b)]
#Euclid_proof#--* [quotient :
(b:nat)
 (gt b O)->(a:nat){q:nat | (Ex [r:nat]a=(plus (mult q b) r)/\(gt b r))}]
#Euclid_proof#--* [modulo :
(b:nat)
 (gt b O)->(a:nat){r:nat | (Ex [q:nat]a=(plus (mult q b) r)/\(gt b r))}]
User Time: 1.11
System Time: 0
```

La syntaxe abstraite qui apparaît à l'impression des types de `div1` et `div2` lors de la recherche de la division euclidienne est due à un clash d'identificateurs entre le `diveucl` de `Div` et le `diveucl` de `Euclid_proof`. En ces circonstances, Coq a pour principe de ne prendre en considération (au niveau de ses caches) que le dernier identificateur importé (ici le `diveucl` de `Euclid_proof`) bien que l'ancien identificateur reste dans l'environnement.

6 Conclusion

6.1 Récapitulatif

En résumé, notre démarche a été la suivante:

- élaborer, pour Coq, une théorie Th^{Coq} faiblement pré-congruente, correcte, incomplète et décidable par rapport à la classe des isomorphismes définissables (et donc par rapport à la classe des isomorphismes sémantiques) et construire une procédure Dec^{Coq} correcte, complète pour Th^{Coq} et terminante.
- développer une autre procédure de décision correcte, incomplète pour Th^{Coq} (mais correspondant à un sous-ensemble congruent) et terminante, qui nous a permis de réaliser l'implémentation de deux outils de recherche, l'un interne et l'autre externe à Coq.
- documenter et tester les prototypes ainsi développés.

6.2 Travaux futurs

Prochainement, des modifications importantes sont à attendre, notamment:

- implémentation du filtrage: une telle fonctionnalité semble en effet inévitable pour que Search2 soit un outil réellement puissant. Il pourrait alors fort bien subsumer le Search actuel de Coq. Tout comme pour ML, il faudra certainement opérer une réduction du problème d'unification qui, si il se fait directement modulo les règles de Th^{Coq}, est indécidable (voir à ce propos et à titre d'exemple [PN97]).
- réécriture à gauche lorsqu'il n'y a pas de dépendance: en effet, seules les réécritures à gauche d'un lieur non anonyme entraînent des modifications à droite, ce qui oblige à garder les termes de conversion lors des normalisations. On peut donc conserver un système congruent en autorisant toute réécriture à gauche d'un lieur à condition que celui-ci soit anonyme.
- identifier plus de types inductifs: on aimerait bien parvenir à identifier les types inductifs à plusieurs constructeurs ayant la même structure ou possédant une forme différente mais construisant sémantiquement le même ensemble.

Références

[BW94] Benjamin Werner. *Une théorie des constructions inductives.* Thèse, Université Paris 7, Mai 1994.

[DD97] David Delahaye. *Search2: un outil de recherche dans une bibliothèque de preuves Coq modulo isomorphismes.* Rapport de DEA (Sémantique, Preuves et Programmation), Université Paris 6, Septembre 1997.

[MR91] Mikael Rittri. *Using types as search keys in function libraries.* Journal of Functional Programming, 1(1):71-89, 1991.

[PL66] P.J.Landin. *The Next 700 Programming Languages.* Communications of ACM Vol.9/Num.3/March 1966.

[PN97] Paliath Narendran, Frank Pfenning and Richard Statman. *On the unification problem for cartesian closed categories.* The Journal of Symbolic Logic Vol.62/Num.2/June 1997.

[PW94] Pierre Weis et Xavier Leroy. *Le langage Caml.* InterEditions, Mars 1994.

[RC95] Roberto Di Cosmo. *Isomorphisms of types: from λ-calculus to information retrieval and language design.* Birkhäuser, 1995.

[SS83] Sergei Soloviev. *The category of finite sets and cartesian closed categories.* Journal of Soviet Mathematics, 22(3):1387–1400, 1983.

[XL94] Xavier Leroy et Pierre Weis. *Manuel de référence du langage Caml.* InterEditions, Mars 1994.

[BB+97] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi et Benjamin Werner. *The Coq Proof Assistant Reference Manual Version 6.1.* INRIA, Rapport technique, Mai 1997.