Remerciements

Je tiens, en tout premier lieu, à remercier Benjamin Werner et Roberto Di Cosmo, mes directeurs de stage pour m'avoir particulièrement bien encadré tout en me laissant de l'initiative.

Un grand merci aussi à Hugo Herbelin et à Bruno Barras, dont la maîtrise du code de Coq m'a éclairé à plusieurs reprises.

Je remercie Cristina Cornes, Gilles Dowek et Christine Paulin-Mohring avec qui j'ai pu aborder certains points théoriques non triviaux.

De manière générale, toute ma gratitude va à l'ensemble de l'équipe Coq et aux thésards du projet Cristal qui ont su me réserver un accueil fort chaleureux dont je me souviendrai.

Enfin, je ne peux oublier Daniel de Rauglaudre qui m'a apporté un soutien permanent et une aide systématique en toutes circonstances.

Table des matières

Chapitre 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 `list_it` qui ne porte pas moins de quatre noms différents dans d'autres versions de ML.

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

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 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 oeuvre 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 faire un tel outil pour Coq.

1.2 Plan

Ce rapport s'articule en cinq parties distinctes. Dans un premier temps, nous ferons un tour d'horizon rapide mais précis du langage CAML et du système Coq en accentuant les différences qu'il peut y avoir au niveau du typage. Dans un deuxième temps, nous rentrerons dans le vif du sujet en donnant les notions de base relatives aux isomorphismes de types afin d'établir, par la suite, une théorie pour Coq. Nous parlerons aussi de l'implémentation en isolant bien les choix clés qui ont pu être faits au cours du codage et en exposant les principaux algorithmes. Enfin, des jeux d'essais judicieusement construits permettront non seulement d'esquisser la correction du prototype (dont le code se trouve en annexe) par rapport à la procédure de décision élaborée lors de l'implémentation mais aussi de connaître ses performances, ce qui est salutaire pour son utilisation future.

Chapitre 2

Notions de CAML et de Coq

Avant de présenter ces deux langages, il est important de comprendre qu'ils n'évoluent pas dans le même univers. CAML est un langage de programmation tandis que Coq est un langage de spécifications formelles. Nous ne chercherons donc pas à comparer ni leurs fonctionnalités ni leurs performances. Cela n'aurait aucun sens. Toutefois, nous mettrons en avant leurs systèmes de types de manière à mieux comprendre ultérieurement comment le problème de l'égalité de deux types se pose différemment suivant la richesse du typage.

2.1 Le langage CAML

2.1.1 Généralités

Le langage CAML est un langage très récent, conçu dans le cadre du projet formel à l'INRIA. Il fait partie de la grande famille des langages ML (ML pour Meta Language) tout droit issus de ISWIM (If you See What I Mean) de P.J.Landin en 1966 ([PL66]).

CAML est un langage fonctionnel. Cependant, il ne l'est pas purement et certains traits impératifs tels la séquence et les boucles subsistent. Il est fortement typé, de manière statique (à la compilation uniquement) et implicite (les types sont inférés). Enfin, il intègre des services à ce jour encore peu répandus comme les types concrets, le filtrage ou le garbage collecting.

2.1.2 Le langage des termes

La syntaxe du noyau de CAML est essentiellement les termes du λ-calcul avec constantes. La construction let...in est utilisée non seulement pour marquer certains β-rédex (termes de la forme ((fun $x \rightarrow a_1$) a_2)) afin de mieux les typer, mais aussi pour contrôler l'évaluation (qui est stricte). On peut les définir par la grammaire suivante:

$expr ::= ident$	
$\mid fun\ x \rightarrow expr$	Abstractions
$\mid expr\ expr$	Applications
$\mid let\ x = expr\ in\ expr$	Déclarations
$ident ::= x$	Variables
$\mid const$	Constantes
$const ::= C$	Constructeurs
$\mid f$	Primitives

Ce noyau s'étend par des constructions impératives comme les références, les exceptions ou les opérations de contrôle (boucles while et for) et par les enregistrements. On peut aussi réaliser des déclarations globales de types et de fonctions. Enfin, des structures de données sont disponibles soit dans l'environnement initial soit dans des modules. Parmi celles-ci, les entiers, les caractères, les chaînes de caractères, les booléens, les flottants, les listes, les tableaux, les tables de hash,...

2.1.3 Le système de type

Si l'on se place dans le noyau de CAML, on peut dire que, par rapport au cube de Barendregt, on se situe au niveau du système F. Cependant, comme l'inférence de type est indécidable dans le système F, on se restreint à des types polymorphes prénexes (les lieurs de variables de types sont placés au début du type exclusivement). Si l'on rajoute d'autres structures à ce noyau telles la mémoire avec le système de références ou les enregistrements, le typage se complique donnant lieu soit à des restrictions supplémentaires soit à des règles ad hoc.

Plus précisément, pour le noyau de CAML, on définit les expressions de types, de schémas de types et d'environnements de typage comme suit:

$\tau ::= \alpha \mid \tau \rightarrow \tau \mid g(\overrightarrow{\tau})$	Types
$\sigma ::= \forall \overrightarrow{\alpha}.\tau$	Schémas de types
$A ::= \emptyset \mid A \oplus (z \mapsto \sigma)$	Environnements de typage

Avec $g(\overrightarrow{\tau})$ représentant tous les types de base (entiers, chaînes de caractères,...). La relation de typage est alors la plus petite relation sur les triplets (A, a, τ), notée $A \vdash a : \tau$ satisfaisant les règles données par le tableau 2.1, où $Gen(\tau, A) = FV(\tau) \backslash FV(A)$.

Pour la suite, on prêtera une attention toute particulière au type paire noté *. Son constructeur (_,_) est considéré comme une constante et ne nécessite aucune règle supplémentaire de typage. On retiendra également le type unit avec un constructeur vide () qui est, par définition, le type des fonctions réalisant des effets de bords (impression, modification de la mémoire,...). Il est à noter que ces deux types sont parfaitement redéfinissables dans le langage. On pourrait les écrire comme suit:

```
let ('a,'b) my_pair = Cpl of ('a,'b);;
let my_unit = Ety of unit;;
```

$$\frac{z : \forall \overrightarrow{\alpha}.\tau_0 \in A}{A \vdash z : \tau_0[\overrightarrow{\alpha} \leftarrow \overrightarrow{\tau}]} \; Var\text{-}Inst$$

$$\frac{A \oplus (x \mapsto \tau_2) \vdash a : \tau_1}{A \vdash fun\ x \rightarrow a : \tau_2 \rightarrow tu_1} \; Fun$$

$$\frac{A \vdash a_1 : \tau_2 \rightarrow \tau_1 \;\; A \vdash a_2 : \tau_2}{A \vdash a_1\ a_2 : \tau_1} \; App$$

$$\frac{A \vdash a_1 : \tau_1 \;\; A \oplus (x \mapsto Gen(\tau_1, A)) \vdash a_2 : \tau_2}{A \vdash let\ x = a_1\ in\ a_2 : \tau_2} \; Gen\text{-}Let$$

Tableau 2.1 : Les règles de typage pour le noyau de CAML.

Pour plus de renseignements sur CAML, on pourra consulter [PW94] et [XL94].

2.2 Le prouveur Coq

2.2.1 Présentation

Coq est un outil d'aide à la preuve basée sur une logique d'ordre supérieure qu'est le calcul des constructions inductif. Coq permet non seulement de faire des preuves mathématiques mais aussi d'extraire des programmes à partir de spécifications prouvées et constructives. Il utilise un style déclaratif (pour exprimer les lemmes, théorèmes,...) et impératifs (pour réaliser les preuves). Ainsi, lors des preuves, l'utilisateur donne des ordres à Coq, qui peuvent être des actions simples de la déduction naturelle ou des tactiques, qui sont des procédures de démonstration automatique.

La première implémentation date de 1984 et fut réalisée en CAML par Gérard Huet et Thierry Coquand dans le cadre du projet formel à l'INRIA. Cette première mouture était basée sur le calcul des constructions et ne permettait pas les définitions inductives directes. Depuis, les versions se sont succédées par des portages sous-jacents (portage CAML-Camllight en 1992 par Daniel de Rauglaudre (Version 5.7), portage Camllight-Objective Caml par Bruno Barras (Version 6.1)) ou par l'introduction de nouvelles structures, outils et tactiques (types mutuellement inductifs par Christine Paulin-Mohring, extraction vers ML par Benjamin Werner, une procédure de décision pour la logique propositionnelle intuitionniste par César Muñoz,...). La version actuelle, la 6.2, est toujours codée en Objective Caml mais utilise les fonctionnalités de Camlp4 (de Daniel de Rauglaudre) pour définir plus facilement ses grammaires.

2.2.2 Les termes

Le langage des termes Coq se définit de la manière suivante:

$S ::= Prop \mid Set \mid Type(i)(i \in N)$	Sortes
$T ::= S \mid c_{env} \mid x \mid (x:T)T \mid [x:T]T \mid (T\ T)$	Termes

Où $Type_i$ est le $Type$ d'univers i (cette architecture en couches empêche l'imprédicativité de Type qui introduit une incohérence; voir à ce propos, le paradoxe de Girard dans [JG72]) et c_{env} représente l'ensemble des constantes de l'environnement.

2.2.3 Le typage

Comme on l'a vu précédemment, le système de type de Coq est le plus riche de tous puisqu'il s'agit du calcul des constructions avec la récurrence. Il est aussi très expressif ce qui permet de donner aux identificateurs des spécifications totales, contrairement à CAML. Enfin, les termes typables sont fortement normalisables par rapport à la β-réduction, ce qui empêche les programmes extraits de boucler.

Plus précisément, le tableau 2.2 nous donne les principales règles de typage de Coq. Nous n'aborderons pas ici les règles de conversion, de définition et des types (co)inductifs, qui sont un peu techniques et nécessitent des notions et commentaires annexes.

Le lecteur, désireux d'en savoir plus, pourra se réferer à [BB$^+$97] pour le système Coq en lui-même et à [BW94] pour la théorie sous-jacente.

$$\frac{}{\mathcal{WF}([])[[]]}\,W\text{-}E$$

$$\frac{E[\Gamma] \vdash T : s \quad s \in S \quad x \notin \Gamma \cup E}{\mathcal{WF}(E)[\Gamma :: (x : T)]}\,W\text{-}s$$

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash Prop : Type(p)}\,Ax_1$$

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash Set : Type(q)}\,Ax_2$$

$$\frac{\mathcal{WF}(E)[\Gamma] \quad i < j}{E[\Gamma] \vdash Type(i) : Type(j)}\,Ax_3$$

$$\frac{\mathcal{WF}(E)[\Gamma] \quad (x : T) \in \Gamma}{E[\Gamma] \vdash x : T}\,Var$$

$$\frac{\mathcal{WF}(E)[\Gamma] \quad (c : T) \in E}{E[\Gamma] \vdash c : T}\,Const$$

$$\frac{E[\Gamma] \vdash T : s_1 \quad E[\Gamma :: (x : T)] \vdash U : s_2 \quad s_1 \in \{Prop, Set\} \text{ ou } s_2 \in \{Prop, Set\}}{E[\Gamma] \vdash (x : T)U : s_2}\,Prod_1$$

$$\frac{E[\Gamma] \vdash T : Type(i) \quad E[\Gamma :: (x : T)] \vdash U : Type(j) \quad i \leq k \quad j \leq k}{E[\Gamma] \vdash (x : T)U : s_2}\,Prod_2$$

$$\frac{E[\Gamma] \vdash (x : T)U : s \quad E[\Gamma :: (x : T)] \vdash t : U}{E[\Gamma] \vdash [x : T]t : (x : T)U}\,Lam$$

$$\frac{E[\Gamma] \vdash t : (x : U)T \quad E[\Gamma] \vdash u : U}{E[\Gamma] \vdash (t\ u) : T[x \leftarrow u]}\,App$$

Tableau 2.2 : Les principales règles de typage de Coq.

Chapitre 3

Isomorphismes de types

Ce chapitre a pour objectif d'exposer les travaux de Roberto Di Cosmo, Kim Bruce, Giuseppe Longo et Sergei Soloviev sur les types isomorphes. Il pose ainsi toutes les définitions et notions, pour la plupart formalisées dans [RC95], nécessaires à l'élaboration d'une théorie pour un langage en général et pour Coq en particulier.

3.1 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 3.1.1 (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 22 pour plus d'informations sur de tels types). tt est l'unique constructeur d'arité 0 du type inductif unit.

Pour montrer comment ces règles peuvent servir à justifier des isomorphismes définissables, voici un petit exemple complet:

Exemple 3.1.2 *On se place dans le λ-calcul polymorphe (système F) avec le symbole de type binaire $\_*\_$, une constante de terme binaire $(\_,\_)$ de type $\forall X.\forall Y.(X * Y)$ et les primitives de projection associées p_1 et p_2 respectivement de types $\forall X.\forall Y.(X * Y) \to X$ et $\forall X.\forall Y.(X * Y) \to Y$. Le but est de montrer que les types $A \to (B * C)$ et $(A \to B) * (A \to C)$ sont isomorphes de manière définissable. Pour ce, il faut construire les termes intervertibles et vérifier que leurs compositions donnent bien deux termes identité. Les termes de conversion s'écrivent comme suit:*

$$M = \lambda z : A \to (B * C).(\lambda x : A.(p_1(z\ x)), \lambda x : A.(p_2(z\ x)))$$
$$N = \lambda z : (A \to B) * (A \to C).\lambda x : A.((p_1\ z)\ x, (p_2\ z)\ x)$$

Et les compositions se font de la manière suivante:

$$\lambda x : A \to (B * C).N\ (M\ x)$$

$$\downarrow (\beta)$$

$$\lambda y : A \to (B * C).N\ T,\ \textit{avec}\ T = (\lambda x : A.p_1\ (y\ x), \lambda x : A.p_2\ (y\ x))$$

$$\downarrow (\beta)$$

$$\lambda y : A \to (B * C).((p_1\ T)\ x, (p_2\ T)\ x)$$

$$\downarrow (\pi^2) \times 2$$

$$\lambda y : A \to (B * C).\lambda x : A.((\lambda x : A.p_1\ (y\ x))\ x, (\lambda x : A.p_2\ (y\ x))\ x)$$

$$\downarrow (\beta) \times 2$$

$$\lambda y : A \to (B * C).\lambda x : A.(p_1\ (y\ x), p_2\ (y\ x))$$

$$\downarrow (SP^2)$$

$$\lambda y : A \to (B * C).\lambda x : A.y\ x$$

$$\downarrow (\eta)$$

$$\lambda y : A \to (B * C).y$$

$$\lambda x : (A \to B) * (A \to C).M\ (N\ x)$$

$$\downarrow (\beta)$$

$$\lambda y : (A \to B) * (A \to C).M\ T,\ \ avec\ T = \lambda x : A.((p_1\ y)\ x, (p_2\ y)\ x)$$

$$\downarrow (\beta)$$

$$\lambda y : (A \to B) * (A \to C).(\lambda x : A.p_1\ (T\ x), \lambda x : A.p_2\ (T\ x))$$

$$\downarrow (\beta) \times 2 + (\pi^2) \times 2$$

$$\lambda y : (A \to B) * (A \to C).(\lambda x : A.(p_1\ y)\ x, \lambda x : A.(p_2\ y)\ x)$$

$$\downarrow (\eta) \times 2$$

$$\lambda y : (A \to B) * (A \to C).(p_1\ y, p_2\ y)$$

$$\downarrow (SP^2)$$

$$\lambda y : (A \to B) * (A \to C).y$$

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 $A \to B$ 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 3.1.3 (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 dans l'exemple suivant:

Exemple 3.1.4 *Il est bien connu que les modèles du λ-calcul simplement typé avec produit et unit sont exactement les catégories cartésiennes fermées (ccc's en anglais). En 1981, Soloviev montra, en utilisant des techniques développés en théorie des nombres plusieurs années auparavant, que les isomorphismes valides dans les ccc's étaient les suivants:*

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

En utilisant la définition 3.1.3, on en déduit facilement que ces isomorphismes sont les seuls isomorphismes sémantiques dans le λ-calcul avec produit et unit.

Remarque 3.1.5 *En fait, pour tirer cette conclusion, on utilise une propriété implicite. La classe des isomorphismes syntaxiques est incluse dans la classe des isomorphismes sémantiques. Cela provient du fait que les termes fonctionnels du langage peuvent toujours s'interpréter par des morphismes (par définition de la sémantique) et que l'on peut, dans certains cas, imaginer des morphismes non définissables.*

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 comme nous le montre l'exemple suivant:

Exemple 3.1.6 *On se place dans les conditions de l'exemple 3.1.4. On assume que les isomorphismes sémantiques énumérés sont également syntaxiques. Comme ce sont les seuls isomorphismes et qu'il ne peut pas y avoir plus d'isomorphismes syntaxiques que sémantiques, cela montrera que les classes d'isomorphismes syntaxiques et sémantiques sont égales. Pour prouver cette assertion, le seul moyen est, par définition, de donner les termes intervertibles et de vérifier que leurs compositions donnent des identités. On se contentera ici de les énoncer en supposant que cela fonctionne bien, mais le lecteur curieux pourra se lancer dans l'exercice de réécriture qui demande un peu de concentration. On a donc:*

1. $\lambda x : A * B.(p_2\ x, p_1\ x)$
   $\lambda x : B * A.(p_2\ x, p_1\ x)$
2. $\lambda x : A * (B * C).((p_1\ x, p_1\ (p_2\ x)), p_2\ (p_2\ x))$
   $\lambda x : (A * B) * C.(p_1\ (p_1\ x), (p_2\ (p_1\ x), p_2\ x))$
3. $\lambda z : (A * B) \to C.\lambda x : A.\lambda y : B.z\ (x, y)$
   $\lambda z : A \to (B \to C).\lambda x : A * B.z\ (p_1\ x)\ (p_2\ x)$
4. $\lambda z : A \to (B * C).(\lambda x : A.(p_1\ (z\ x)), \lambda x : A.(p_2\ (z\ x)))$
   $\lambda z : (A \to B) * (A \to C).\lambda x : A.((p_1\ z)\ x, (p_2\ z)\ x)$
5. $\lambda x : A * unit.p_1\ x$
   $\lambda x : A.(x, tt)$
6. $\lambda x : (A \to unit).tt$
   $\lambda y : unit.\lambda x : A.tt$
7. $\lambda x : (unit \to A).x\ tt$
   $\lambda y : A.\lambda w : unit.y$

Il y a de nombreux autres exemples où les deux classes se correspondent. On peut citer le λ-calcul simplement typé, le sytème F, le λ-calcul simplement typé avec produit, le λ-calcul simplement typé avec unit,...

3.2 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. On peut remarquer que cette seconde méthode est générale et s'applique aussi bien au cas syntaxique que sémantique.

3.2.1 Intervertiblité

Pour trouver tous les isomorphismes syntaxiques du λ-calcul simplement typé, une possibilité est de s'appuyer sur les travaux de Dezani-Ciancaglini ([DC76]) qui a montré que les termes non typés intervertibles avaient une forme simple:

Théorème 3.2.1 (Termes intervertibles du λ-calcul non typé) *Soit M un terme normalisable. M est intervertible si et seulement si M est une permutation héréditaire finie (fhp en anglais).*

Une permutation héréditaire finie est définie comme suit:

Définition 3.2.2 (Permutations héréditaires finies) *Un terme non typé M est une permutation héréditaire finie si et seulement si:*

- *$M = \lambda x.x$, ou*
- *$M = \lambda z.\lambda v_1 ... \lambda v_n.z\ P_1 ... P_n$ et il existe une permutation $\Sigma : n \to n$, telle que $\lambda x_i.P_{\Sigma(i)}$ est une permutation héréditaire finie pour tout $1 \leq i \leq n$.*

Exemple 3.2.3 *$\lambda z.\lambda x_1.\lambda x_2.z\ x_2\ x_1$ est une fhp (avec $\Sigma = \{1 \to 2;\ 2 \to 1\}$). De même que $\lambda z.\lambda x_1.\lambda x_2.z\ x_2.\lambda x_3.\lambda x_4.x_1\ x_4\ x_3$ (avec $\Sigma_1 = id$ et $\Sigma_2 = \{1 \to 1;\ 2 \to 3;\ 3 \to 2\}$).*

Théorème 3.2.4 (Termes intervertibles pour le λ-calcul simplement typé) *Les fhp sont typables et sont exactement les termes intervertibles du λ-calcul simplement typé (modulo une fonction de gommage qui élimine toutes les indications de typage dans les termes).*

Ce résultat s'étend facilement au second ordre (système F). Utilisant le résultat de Dezani, Kim Bruce et Giuseppe Longo ont montré que les formes intéressantes étaient les permutations héréditaires finies du second ordre (2-fhp en anglais) définies directement dans le système F de la manière suivante:

Définition 3.2.5 (Permutations héréditaires finies du second ordre) *Un terme M du système F est une permutation héréditaire finie du second ordre si et seulement si:*

- $M = \lambda x : A.x$, *ou*
- $M = \lambda z : Z.\lambda x_1 : X_1 ... \lambda x_n : X_n.z\ P_1 ... P_n$ *et il existe une permutation* $\Sigma : n \rightarrow n$ *telle que pour tout* $1 \leq i \leq n$*:*
  - *si* X_i *est une variable de type alors* $P_{\Sigma(i)}$ *est* X_i
  - *sinon* $\lambda x_i : X_i.P_{\Sigma(i)}$ *est une permutation héréditaire finie du second ordre*

Théorème 3.2.6 (Termes intervertibles pour le système F) *Les 2-fhp sont exactement les termes intervertibles du système F.*

Remarque 3.2.7 *On peut noter que ces caractérisations:*

- *très simples et très utiles ne sont malheureusement pas toujours possibles. Les problèmes apparaissent notamment avec l'introduction de constantes comme le produit et unit qui ne permettent alors que la caractérisation d'un sous-ensemble des termes intervertibles.*
- *nous donnent un moyen de décider si deux types sont isomorphes dans le* λ*-calcul simplement typé et le système F.*
- *permettent des preuves de complétude de théories.*

3.2.2 Théories

Une autre méthode pour prouver que deux types sont isomorphes est de construire une théorie équationnelle. Mais comme on l'a déjà dit, cette méthode ne se cantonne pas uniquement aux isomorphismes définissables. 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 la correction et de la complétude. Ces propriétés s'expriment traditionnellement de la manière suivante:

Définition 3.2.8 (Correction, complétude) *On dira qu'une théorie équationnelle Th est correcte vis-à-vis des isomorphismes d'un calcul si et seulement si:*

$$\forall A, B \; Th \models A = B \Rightarrow A \cong_{d|s} B.$$

Inversement, on dira qu'une théorie équationnelle Th est complète vis-à-vis des isomorphismes d'un calcul si et seulement si:

$$\forall A, B \; A \cong_{d|s} B \Rightarrow Th \models A = B.$$

La notation $A \cong_{d|s} B$ signifie que A et B sont isomorphes soit de manière définissable soit sémantiquement.

Comme on peut s'y attendre, la correction est assez facile à montrer. Si l'équivalence de types est syntaxique, il suffit d'exhiber les termes intervertibles dans le calcul. On peut le faire soit directement soit indirectement par des techniques de preuves (et l'isomorphisme de Curry-Howard). Si l'équivalence est sémantique, c'est un peu plus difficile car il faut exhiber les bijections dans tous les modèles. Toutefois, comme on l'a vu dans l'exemple 3.1.4, on peut, dans certains cas, donner des modèles catégoriques et utiliser de fait des résultats généraux développés dans la théorie des catégories. Quant à la complétude, c'est une propriété bien plus difficile à prouver. Elle demande de construire une preuve à partir d'une équivalence valide de manière générale, ce qui n'est pas trivial.

Exemple 3.2.9 *On se place dans le système F. On peut donner la théorie suivante:*

1. $\Pi x : A.\Pi y : B.C = \Pi y : B.\Pi x : A.C$ *si* $x \notin FV(B)$ *et* $y \notin FV(A)$
2. $\Pi x : A.B = \Pi y : A.B[x \leftarrow y]$ *si* $y \notin FV(\Pi x : A.B)$

On peut montrer que cette théorie est correcte et complète par rapport à l'équivalence syntaxique et sémantique (qui sont identiques dans ce calcul).

Une dernière propriété à laquelle il faut prêter attention est la décidabilité de la théorie pour savoir quelle implémentation on peut en faire. Cela rejoint, en général, indirectement la question de la décidabilité de l'équivalence que la théorie tend à caractériser dans la mesure où l'on cherche à construire cette théorie de manière correcte et complète. Pour montrer la décidabilité, il faut chercher à réduire le problème à un calcul. Pour montrer l'indécidabilité, il faut coder un problème connu indécidable (problème de correspondance de Post, calcul des prédicats, filtrage d'ordre supérieur,...) dans son problème. Si décidabilité il y a, la procédure de recherche doit être correcte, complète (par rapport à la théorie) et terminante. Si le problème est indécidable, la procédure doit faire l'impasse sur la complétude.

Chapitre 4

Une théorie pour Coq

Le but de ce chapitre est 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.

4.1 Les équations

Avant de donner les équations et pour bien comprendre les généralisations et extensions qui ont pu être effectuées par rapport à CAML, nous allons faire un petit rappel de la théorie réalisée par Roberto Di Cosmo pour CAML. Cette théorie s'articule autour des constructeurs de types $\rightarrow$ et $\times$, du type unit et du second ordre. En utilisant une syntaxe plutôt à la Church qu'à la CAML et en notant $*$ un ensemble de types, les règles de Th^{CAML} peuvent s'écrire de la manière suivante:

1. $A \times B = B \times A$
2. $A \times (B \times C) = (A \times B) \times C$
3. $(A \times B) \rightarrow C = A \rightarrow (B \rightarrow C)$
4. $A \rightarrow (B \times C) = (A \rightarrow B) \times (A \rightarrow C)$
5. $A \times unit = A$
6. $A \rightarrow unit = unit$
7. $unit \rightarrow A = A$
8. $\Pi x : *.\Pi y : *.A = \Pi y : *.\Pi x : *.A$
9. $\Pi x : *.A = \Pi y : *.A[x \leftarrow y]$ si $y \notin FV(\Pi x : *.A)$
10. $\Pi x : *.A \times B = \Pi x : *.\Pi y : *.A \times (B[x \leftarrow y])$

Remarque 4.1.1 *On peut noter que:*

- *dans l'équation 6, A peut représenter $*$ en particulier*
- *les axiomes du second ordre respectent les formes prénexes imposées par le système de types de CAML*
- *le swap à l'ordre 1 se déduit de (1) et (3)*

- *ce système est correct et complet pour les isomorphismes de CAML (syntaxiques et sémantiques car les deux classes se correspondent). De plus, il définit une congruence décidable.*

La théorie que nous allons donner pour Coq s'appuie essentiellement sur celle de CAML 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. Plus précisément, on aura:

Définition 4.1.2 (Réécriture des types inductifs) *$\mathcal{R}_{Ind}$ est le système de réécriture constitué des règles suivantes:*
si on a:

$$\texttt{Inductive TypInd [}\overrightarrow{p} : \overrightarrow{P}\texttt{] : T :=}$$
$$\texttt{Cind: }\Pi\overrightarrow{x} : \overrightarrow{X}\texttt{.(TypInd }\overrightarrow{p}\texttt{).}$$

avec `T` *sans* $\rightarrow$ *et* `TypInd`, `Cind` $\notin \overrightarrow{X}$, *alors 3 cas sont possibles:*

- $\overrightarrow{x} = \overrightarrow{X} = \overrightarrow{0}$*:*

$$\texttt{TypInd} \overset{\mathcal{R}_{Ind}}{\longrightarrow} \lambda\overrightarrow{p} : \overrightarrow{P}.unit$$
$$\texttt{Cind} \overset{\mathcal{R}_{Ind}}{\longrightarrow} \lambda\overrightarrow{p} : \overrightarrow{P}.tt$$

- $\overrightarrow{x} = \{x\},\ \overrightarrow{X} = \{X\}$*:*

$$\texttt{TypInd} \overset{\mathcal{R}_{Ind}}{\longrightarrow} \lambda\overrightarrow{p} : \overrightarrow{P}.X$$
$$\texttt{Cind} \overset{\mathcal{R}_{Ind}}{\longrightarrow} \lambda\overrightarrow{p} : \overrightarrow{P}.\lambda x : X.x$$

- $\overrightarrow{x} = \{x_0, ..., x_{n-1}\}$, $\overrightarrow{X} = \{X_0, ..., X_n\}$:

  `TypInd` $\stackrel{\mathcal{R}_{Ind}}{\longrightarrow}$
  $\lambda \overrightarrow{p} : \overrightarrow{P}.\Sigma x_0 : X_0....\Sigma x_{n-1} : X_{n-1}.X_n$
  `Cind` $\stackrel{\mathcal{R}_{Ind}}{\longrightarrow}$
  $\lambda \overrightarrow{p} : \overrightarrow{P}.\lambda x_0 : X_0....\lambda x_{n-1} : X_{n-1}.\lambda x_n : X_n.(x_0, (...(x_{n-1}, x_n)))$

Remarque 4.1.3 *On peut noter que:*

- *les définitions mutuellement inductives ne sont pas traitées.*
- *cette réécriture peut créer des β-redex.*

Exemple 4.1.4 *Prenons le cas de* `sigS2` *(dans Specif) défini de la manière suivante:*

```
Inductive sigS2 [A:Set;P:A->Set;Q:A->Set] : Set :=
      existS2 : (x:A)(P x)->(Q x)->(sigS2 A P Q)
```

D'après la définition 4.1.2, les réécritures de `sigS2` *peuvent s'écrire comme suit:*

`sigS2` $\stackrel{\mathcal{R}_{Ind}}{\longrightarrow}$
$\lambda A : Set.\lambda P : A \rightarrow Set.\lambda Q : A \rightarrow Set.\Sigma x : A.\Sigma_{\_} : (P\ x).(Q\ x)$
`existS2` $\stackrel{\mathcal{R}_{Ind}}{\longrightarrow}$
$\lambda A : Set.\lambda P : A \rightarrow Set.\lambda Q : A \rightarrow Set.\lambda x : A.\lambda y : (P\ x).\lambda z : (P\ x).(x, (y, z))$

Théorème 4.1.5 (Terminaison) *$\mathcal{R}_{Ind}$ termine.*

Preuve *Etant donné que* `TypInd` *et* `Cind` *n'apparaissent pas dans les $\overrightarrow{X}$ et que les définitions mutuellement inductives ne sont pas prises en compte, on peut prendre l'ordre lexicographique (nombre d'occurences de* `TypInd`*,nombre d'occurences de* `Cind`*) pour conclure.*

□

Ce changement de syntaxe (justifié sémantiquement) nous permet d'obtenir une algèbre des types basés sur Σ[1] et Π (les autres types inductifs sont considérés comme des constantes). Th^{Coq} s'exprime alors de la manière suivante:

[1] Ces Σ sont codables dans le CCI et peuvent être vus comme une famille de termes (suivant le type des composantes) plutôt que comme un lieur au même titre que Π.

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. $\Pi x : unit.A = A[x \leftarrow tt]$

Remarque 4.1.6 *Notons que:*

- $\xrightarrow{\beta\iota}$ *permet en particulier les réductions suivantes:*

  $$\pi_1\ (x, y) \xrightarrow{\beta\iota} x$$
  $$\pi_2\ (x, y) \xrightarrow{\beta\iota} y$$

  Où π_1 et π_2 représentent les projections associées à Σ (n'oublions pas que nous nous sommes assurés de leur existence).

- *tt représente le constructeur du type unit (dans Datatypes).*

- *unit a un rôle assymétrique par rapport à Σ, contrairement à $*$ dans la théorie de CAML. Cette assymétrie s'explique par la présence d'une dépendance.*

4.2 Propriétés

4.2.1 Passage au contexte

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

Définition 4.2.1 (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).

L'exemple suivant nous illustre bien pourquoi cette propriété n'est pas vérifiée dans Th^{Coq}:

Exemple 4.2.2 *Soit l'équation E et le contexte C définis de la manière suivante:*

$$E \equiv \Pi x : (\Sigma y : A.B).C = \Pi x : A.\Pi y : B[y \leftarrow x].C[x \leftarrow (x, y)]$$
$$C[\Omega] = \Pi f : \Omega.\Pi c : (\Sigma y : A.B).(f\ c)$$

E est une équation valide (en fait c'est l'axiome 6 de curryfication). Pour que $C[\Omega]$ soit un type correct, on doit restreindre Ω à des types de la forme $\Pi x : (\Sigma y : A.B).C$. En appliquant C à l'équation E, on s'aperçoit que:

$$\Pi f : (\Pi x : (\Sigma y : A.B).C).\Pi c : (\Sigma y : A.B).(f\ c) \neq$$
$$\Pi f : (\Pi x : A.\Pi y : B[y \leftarrow x].C[x \leftarrow (x, y)]).\Pi c : (\Sigma y : A.B).(f\ c)$$

Ce résultat s'explique très bien, au-delà du fait que l'on ne peut pas obtenir de dérivation dans Th^{Coq}, dans la mesure où le terme de gauche est bien typé mais pas celui de droite.

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. Pour ce qui est de la justification syntaxique de Th^{Coq}, on va la supposer pour l'instant en attendant de la montrer à la section suivante. Quant à la modification du contexte, on va voir comment elle s'effectue suivant la forme du contexte. On posera donc:

Définition 4.2.3 (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 4.2.4 *Th^{Coq} est une pré-congruence faible.*

Preuve *Compte-tenu de l'algèbre des types que l'on considère, les contextes possibles sont:*

$$\mathcal{C}_{Coq} = \{\Xi x : \Omega.A,\ \Xi x : A.\Omega,\ \Xi x : \Omega.\Omega\}\ avec\ \Xi \in \{\Pi, \Sigma\}.$$

On va raisonner par cas sur $\mathcal{C}_{Coq}$ en supposant que toute égalité prouvée dans Th^{Coq} se justifie syntaxiquement par l'existence de deux termes intervertibles. On pose donc $B = B'$ avec $\sigma : B \to B'$ et $\tau : B' \to B$. En prenant des contextes C tels que $C[B]$ soit bien typé (pour les autres, le théorème est trivialement vrai), on a:

- $\Xi x : \Omega.A$*: deux cas se présentent suivant* Ξ*:*
  - $\Xi = \Pi$*: on a* $\Pi x : B.A = \Pi x : B'.A[x \leftarrow (\tau\ x)]$ *et les deux termes intervertibles sont* $\sigma' = \lambda f : (\Pi x : B.A).\lambda x : B'.f\ (\tau\ x)$ *et* $\tau' = \lambda f : (\Pi x : B'.A[x \leftarrow (\tau\ x)]).\lambda x : B.f\ (\sigma\ x)$
  - $\Xi = \Sigma$*: on a* $\Sigma x : B.A = \Sigma x : B'.A[x \leftarrow (\tau\ x)]$ *et les deux termes intervertibles sont* $\sigma' = \lambda c : (\Sigma x : B.A).(\sigma\ (\pi_1\ c),\ \pi_2\ c)$ *et* $\tau' = \lambda c : (\Sigma x : B'.A[x \leftarrow (\tau\ x)]).(\tau\ (\pi_1\ c),\ \pi_2\ c)$
- $\Xi x : A.\Omega$*: suivant* Ξ*, on a:*

  - $\Xi = \Pi$: *on a* $\Pi x : A.B = \Pi x : A.B'$ *et les deux termes intervertibles sont* $\sigma' = \lambda f : (\Pi x : A.B).\lambda x : A.\sigma\ (f\ x)$ *et* $\tau' = \lambda f : (\Pi x : A.B').\lambda x : A.\tau\ (f\ x)$
  - $\Xi = \Sigma$: *on a* $\Sigma x : A.B = \Sigma x : A.B'$ *et les deux termes intervertibles sont* $\sigma' = \lambda c : (\Sigma x : A.B).(\pi_1\ c, \sigma\ (\pi_2\ c))$ *et* $\tau' = \lambda c : (\Sigma x : A.B').(\pi_1\ c, \tau\ (\pi_2\ c))$

- $\Xi x : \Omega.\Omega$: *on a deux cas selon* Ξ:

  - $\Xi = \Pi$: *on a* $\Pi x : B.B = \Pi x : B'.B'$ *et les deux termes intervertibles sont* $\sigma' = \lambda f : (\Pi x : B.B).\lambda x : B'.\sigma\ (f\ (\tau\ x))$ *et* $\tau' = \lambda f : (\Pi x : B'.B').\lambda x : B.\tau\ (f\ (\sigma\ x))$
  - $\Xi = \Sigma$: *on a* $\Sigma x : B.B = \Sigma x : B'.B'$ *et les deux termes intervertibles sont* $\sigma' = \lambda c : (\Sigma x : B.B).(\sigma\ (\pi_1\ c), \sigma\ (\pi_2\ c))$ *et* $\tau' = \lambda c : (\Sigma x : B'.B').(\tau\ (\pi_1\ c), \tau\ (\pi_2\ c))$

$\square$

La preuve de ce théorème nous permet de modifier la couche de logique équationnelle à ajouter aux axiomes donnés en 4.1, au niveau du passage au contexte.

4.2.2 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 4.2.5 (Correction syntaxique) Th^{Coq} *est une théorie correcte vis-à-vis de la classe des isomorphismes syntaxiques.*

Preuve *On se convainc sans mal de la validité syntaxique des règles de la logique équationnelle modifiée pour* Th^{Coq} *(on montre que les prémisses impliquent la conclusion). Ainsi, pour la reflexivité, on prend l'identité correspondante, pour la symétrie, on prend les mêmes termes intervertibles, pour la transitivité, on compose les termes intervertibles, et pour le passage au contexte, il suffit de se rapporter à la preuve du théorème 4.2.4. Pour les axiomes énumérés en 4.1, il nous faut exhiber les termes intervertibles, ce qui nous donne:*

1. $\lambda x : A.x^1$
   $\lambda x : B.x^1$
2. $\lambda c : (\Sigma x : A.B).c$
   $\lambda c : (\Sigma y : A.B[x \leftarrow y]).c$
3. $\lambda f : (\Pi x : A.B).\lambda y : A.(f\ y)$
   $\lambda f : (\Pi y : A.B[x \leftarrow y]).\lambda x : A.(f\ x)$
4. $\lambda c : (\Sigma x : A.B).(\pi_2\ c, \pi_1\ c)$

$\lambda c : (\Sigma x : B.A).(\pi_2\ c, \pi_1\ c)$

5. $\lambda z : (\Sigma x : (\Sigma y : A.B).C).(\pi_1\ (\pi_1\ z), (\pi_2\ (\pi_1\ z), \pi_2\ z))$
   $\lambda z : (\Sigma x : A.\Sigma y : B[y \leftarrow x].C[x \leftarrow (x, y)]).((\pi_1\ z, \pi_1\ (\pi_2\ z)), \pi_2\ (\pi_2\ z))$
6. $\lambda f : (\Pi x : (\Sigma y : A.B).C).\lambda x : A.\lambda y : B[y \leftarrow x].f\ (x, y)$
   $\lambda f : (\Pi x : A.\Pi y : B[y \leftarrow x].C[x \leftarrow (x, y)]).\Pi x : (\Sigma y : A.B).f\ (\pi_1\ x)\ (\pi_2\ x)$
7. $\lambda f : (\Pi x : A.\Sigma y : B.C).(\Pi x : A.(\pi_1\ (f\ x)), \Pi x : A.(\pi_2\ (f\ x)))$
   $\lambda c : (\Sigma y : (\Pi x : A.B).(\Pi x : A.C[y \leftarrow (y\ x)]).\lambda x : A.((\pi_1\ c)\ x, (\pi_2\ c)\ x)$
8. $\lambda c : (\Sigma x : A.unit).\pi_1\ c$
   $\lambda x : A.(x, tt)$
9. $\lambda c : (\Sigma x : unit.A).(\pi_2\ c)$
   $\lambda a : A[x \leftarrow tt].(tt, a)$
10. $\lambda f : (\Pi x : A.unit).tt$
    $\lambda u : unit.\lambda x : A.tt$
11. $\lambda f : (\Pi x : unit.A).f\ tt$
    $\lambda a : A[x \leftarrow tt].\lambda x : unit.a$

□

Corollaire 4.2.6 (Correction sémantique) *Th^{Coq} est une théorie correcte vis-à-vis de la classe des isomorphismes sémantiques.*

Preuve *Utiliser la remarque 3.1.5.*

□

Le théorème 4.2.5 nous permet maintenant d'exprimer de manière complète Th^{Coq} comme nous le montre les tableaux 4.1 et 4.2.

4.2.3 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 4.2.7 (Incomplétude syntaxique) *Th^{Coq} est une théorie incomplète vis-à-vis de la classe des isomorphismes syntaxiques.*

Preuve *Pour montrer l'incomplétude syntaxique de Th^{Coq}, il suffit d'exhiber deux types isomorphes syntaxiquement tels que Th^{Coq} ne puisse pas dériver leur égalité. Prenons par exemple les deux types inductifs* `nat` *et* `newnat` *définis en Coq par:*

```
Inductive nat:Set:=O:nat
                  |S:nat->nat.

Inductive newnat:Set:=ON:newnat
                     |SN:newnat->newnat.
```

[2] *On utilise la conversion du CCI.*

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

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

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

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

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

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

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

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

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

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

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

Tableau 4.1 : Les axiomes de Th^{Coq}

$$\frac{}{A =_{\begin{pmatrix} \lambda x : A.x \\ \lambda x : A.x \end{pmatrix}} A}\, Th^{Coq}_{Ref}$$

$$\frac{A =_{\begin{pmatrix} \sigma \\ \tau \end{pmatrix}} B}{B =_{\begin{pmatrix} \tau \\ \sigma \end{pmatrix}} A}\, Th^{Coq}_{Sym}$$

$$\frac{A =_{\begin{pmatrix} \sigma \\ \tau \end{pmatrix}} B \quad B =_{\begin{pmatrix} \sigma' \\ \tau' \end{pmatrix}} C}{A =_{\begin{pmatrix} \sigma \circ \sigma' \\ \tau \circ \tau' \end{pmatrix}} C}\, Th^{Coq}_{Trs}$$

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

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

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

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

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

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

Tableau 4.2 : La logique équationnelle de Th^{Coq}

Ces deux types sont syntaxiquement isomorphes et les termes intervertibles sont les suivants:

```
Fixpoint nat2newnat [n:nat]:newnat:=
  Cases n of
     O => ON
    |(S n) => (SN (nat2newnat n))
  end.

Fixpoint newnat2nat [nn:newnat]:nat:=
  Cases nn of
     ON => O
    |(SN nn) => (S (newnat2nat nn))
  end.
```

Toutefois, Th^{Coq} ne peut pas prouver que `nat` *et* `newnat` *sont égaux, dans la mesure où elle ne traite que les types inductifs à un constructeur et sans arité.*

□

Corollaire 4.2.8 (Incomplétude sémantique) *Th^{Coq} est une théorie incomplète vis-à-vis de la classe des isomorphismes sémantiques.*

Preuve *Utiliser la remarque 3.1.5.*

□

4.2.4 Décidabilité

On peut montrer que Th^{Coq} est décidable. Pour prouver cette décidabilité, on doit trouver une procédure correcte, complète (vis-à-vis de Th^{Coq}), et terminante. On va donc, dans un premier temps, décrire la procédure sensée implémenter Th^{Coq}, puis, dans un deuxième temps, on s'intéressera aux propriétés de correction, complétude et terminaison.

On considère le système de réécriture $\mathcal{R}^{Coq}$ donné par le tableau 4.3. Tout comme Th^{Coq}, $\mathcal{R}^{Coq}$ ne passe pas au contexte et n'est qu'une pré-congruence faible. On pose donc de plus les règles du tableau 4.4.

Remarque 4.2.9 *Les contextes C utilisés dans les règles d'inférence du tableau 4.4 sont supposés tels que $C[A]$ soit bien typé.*

Nous pouvons maintenant étudier les principales propriétés de $\mathcal{R}^{Coq}$. En particulier, nous allons nous intéresser à la confluence, à la terminaison et à la forme des formes normales.

Théorème 4.2.10 (Confluence) *$\mathcal{R}^{Coq}$ est confluent.*

$$A \overset{\mathcal{R}^{Coq}}{\longrightarrow}$$
$$\begin{pmatrix} \lambda x : A.x \\ \lambda x : B.x \end{pmatrix}$$
$$B \text{ si } A \overset{\beta\iota}{\longrightarrow} B \; (\mathcal{R}^{Coq}_{\beta\iota Rd})$$

$$\Sigma x : (\Sigma y : A.B).C \overset{\mathcal{R}^{Coq}}{\longrightarrow}$$
$$\begin{pmatrix} \lambda z : (\Sigma x : (\Sigma y : A.B).C).(\pi_1 \; (\pi_1 \; z), (\pi_2 \; (\pi_1 \; z), \pi_2 \; z)) \\ \lambda z : (\Sigma x : A.\Sigma y : B[y \leftarrow x].C[x \leftarrow (x,y)]).((\pi_1 \; z, \pi_1 \; (\pi_2 \; z)), \pi_2 \; (\pi_2 \; z)) \end{pmatrix}$$
$$\Sigma x : A.\Sigma y : B[y \leftarrow x].C[x \leftarrow (x,y)] \; (\mathcal{R}^{Coq}_{\Sigma Ass})$$

$$\Pi x : (\Sigma y : A.B).C \overset{\mathcal{R}^{Coq}}{\longrightarrow}$$
$$\begin{pmatrix} \lambda f : (\Pi x : (\Sigma y : A.B).C).\lambda x : A.\lambda y : B[y \leftarrow x].f \; (x,y) \\ \lambda f : (\Pi x : A.\Pi y : B[y \leftarrow x].C[x \leftarrow (x,y)]).\Pi x : (\Sigma y : A.B).f \; (\pi_1 \; x) \; (\pi_2 \; x) \end{pmatrix}$$
$$\Pi x : A.\Pi y : B[y \leftarrow x].C[x \leftarrow (x,y)] \; (\mathcal{R}^{Coq}_{\Sigma Cur})$$

$$\Pi x : A.\Sigma y : B. \overset{\mathcal{R}^{Coq}}{\longrightarrow} C$$
$$\begin{pmatrix} \lambda f : (\Pi x : A.\Sigma y : B.C).(\Pi x : A.(\pi_1 \; (f \; x)), \Pi x : A.(\pi_2 \; (f \; x))) \\ \lambda c : (\Sigma y : (\Pi x : A.B).(\Pi x : A.C[y \leftarrow (y \; x)]).\lambda x : A.((\pi_1 \; c) \; x, (\pi_2 \; c) \; x) \end{pmatrix}$$
$$\Sigma y : (\Pi x : A.B).(\Pi x : A.C[y \leftarrow (y \; x)] \; (\mathcal{R}^{Coq}_{\Pi Dis})$$

$$\Sigma x : A.unit \overset{\mathcal{R}^{Coq}}{\longrightarrow}$$
$$\begin{pmatrix} \lambda c : (\Sigma x : A.unit).\pi_1 \; c \\ \lambda x : A.(x, tt) \end{pmatrix}$$
$$A \; (\mathcal{R}^{Coq}_{\Sigma Unr})$$

$$\Sigma x : unit.A \overset{\mathcal{R}^{Coq}}{\longrightarrow}$$
$$\begin{pmatrix} \lambda c : (\Sigma x : unit.A).(\pi_2 \; c) \\ \lambda a : A[x \leftarrow tt].(tt, a) \end{pmatrix}$$
$$A[x \leftarrow tt] \; (\mathcal{R}^{Coq}_{\Sigma Unl})$$

$$\Pi x : A.unit \overset{\mathcal{R}^{Coq}}{\longrightarrow}$$
$$\begin{pmatrix} \lambda f : (\Pi x : A.unit).tt \\ \lambda u : unit.\lambda x : A.tt \end{pmatrix}$$
$$unit \; (\mathcal{R}^{Coq}_{\Pi Unr})$$

$$\Pi x : unit.A \overset{\mathcal{R}^{Coq}}{\longrightarrow}$$
$$\begin{pmatrix} \lambda f : (\Pi x : unit.A).f \; tt \\ \lambda a : A[x \leftarrow tt].\lambda x : unit.a \end{pmatrix}$$
$$A[x \leftarrow tt] \; (\mathcal{R}^{Coq}_{\Pi Unl})$$

Tableau 4.3 : La base de $\mathcal{R}^{Coq}$

$$\frac{A \overset{\mathcal{R}^{Coq}}{\underset{\binom{\sigma}{\tau}}{\longrightarrow}} A'}{\Pi x : A.B \overset{\mathcal{R}^{Coq}}{\underset{\binom{\lambda f : (\Pi x : A.B).\lambda x : A'.f\ (\tau\ x)}{\lambda f : (\Pi x : A'.B[x \leftarrow (\tau\ x)]).\lambda x : A.f\ (\sigma\ x)}}{\longrightarrow}} \Pi x : A'.B[x \leftarrow (\tau\ x)]}\ \mathcal{R}^{Coq}_{\Pi L}$$

$$\frac{A \overset{\mathcal{R}^{Coq}}{\underset{\binom{\sigma}{\tau}}{\longrightarrow}} A'}{\Sigma x : A.B \overset{\mathcal{R}^{Coq}}{\underset{\binom{\lambda c : (\Sigma x : A.B).(\sigma\ (\pi_1\ c),\ \pi_2\ c)}{\lambda c : (\Sigma x : A'.B[x \leftarrow (\tau\ x)]).(\tau\ (\pi_1\ c),\ \pi_2\ c)}}{\longrightarrow}} \Sigma x : A'.B[x \leftarrow (\tau\ x)]}\ \mathcal{R}^{Coq}_{\Sigma L}$$

$$\frac{A \overset{\mathcal{R}^{Coq}}{\underset{\binom{\sigma}{\tau}}{\longrightarrow}} A'}{\Pi x : B.A \overset{\mathcal{R}^{Coq}}{\underset{\binom{\lambda f : (\Pi x : B.A).\lambda x : B.\sigma\ (f\ x)}{\lambda f : (\Pi x : B.A').\lambda x : B.\tau\ (f\ x)}}{\longrightarrow}} \Pi x : B.A'}\ \mathcal{R}^{Coq}_{\Pi R}$$

$$\frac{A \overset{\mathcal{R}^{Coq}}{\underset{\binom{\sigma}{\tau}}{\longrightarrow}} A'}{\Sigma x : B.A \overset{\mathcal{R}^{Coq}}{\underset{\binom{\lambda c : (\Sigma x : B.A).(\pi_1\ c, \sigma\ (\pi_2\ c))}{\lambda c : (\Sigma x : B.A').(\pi_1\ c, \tau\ (\pi_2\ c))}}{\longrightarrow}} \Sigma x : B.A'}\ \mathcal{R}^{Coq}_{\Sigma R}$$

$$\frac{A \overset{\mathcal{R}^{Coq}}{\underset{\binom{\sigma}{\tau}}{\longrightarrow}} A'}{\Pi x : A.A \overset{\mathcal{R}^{Coq}}{\underset{\binom{\lambda f : (\Pi x : A.A).\lambda x : A'.\sigma\ (f\ (\tau\ x))}{\lambda f : (\Pi x : A'.A').\lambda x : A.\tau\ (f\ (\sigma\ x))}}{\longrightarrow}} \Pi x : A'.A'}\ \mathcal{R}^{Coq}_{\Pi B}$$

$$\frac{A \overset{\mathcal{R}^{Coq}}{\underset{\binom{\sigma}{\tau}}{\longrightarrow}} A'}{\Sigma x : A.A \overset{\mathcal{R}^{Coq}}{\underset{\binom{\lambda c : (\Sigma x : A.A).(\sigma\ (\pi_1\ c), \sigma\ (\pi_2\ c))}{\lambda c : (\Sigma x : A'.A').(\tau\ (\pi_1\ c), \tau\ (\pi_2\ c))}}{\longrightarrow}} \Sigma x : A'.A'}\ \mathcal{R}^{Coq}_{\Sigma B}$$

Tableau 4.4 : Le passage au contexte de $\mathcal{R}^{Coq}$

Preuve *Pour prouver la confluence de $\mathcal{R}^{Coq}$, il suffit de prouver la confluence de ses paires critiques. Les paires critiques de $\mathcal{R}^{Coq}$ sont nombreuses et nous ne les traiterons pas toutes. Le lecteur intéressé pourra les vérifier facilement de manière exhaustive. Pour notre part, nous allons nous attacher au cas où l'on a le choix entre $\mathcal{R}^{Coq}_{\Sigma Cur}$ et $\mathcal{R}^{Coq}_{\Pi Dis}$:*

$$\Pi x : (\Sigma y : A.B).\Sigma z : C.D$$
$$\downarrow (\mathcal{R}^{Coq}_{\Sigma Cur})$$
$$\Pi x : A.\Pi y : B[y \leftarrow x].\Sigma z : C[x \leftarrow (x,y)].D[x \leftarrow (x,y)]$$
$$\downarrow (\mathcal{R}^{Coq}_{\Pi Dis}) \times 2$$
$$\begin{array}{c}\Sigma z : (\Pi x : A.\Pi y : B[y \leftarrow x].C[x \leftarrow (x,y)]).\Pi x : A.\Pi y : B[y \leftarrow x]. \\ D[x \leftarrow (x,y)][z \leftarrow (z\ x\ y)]\end{array} \quad \textit{(1)}$$

$$\Pi x : (\Sigma y : A.B).\Sigma z : C.D$$
$$\downarrow (\mathcal{R}^{Coq}_{\Pi Dis})$$
$$\Sigma z : (\Pi x : (\Sigma y : A.B).C).\Pi x : (\Sigma y : A.B).D[z \leftarrow (z\ x)]$$
$$\downarrow (\mathcal{R}^{Coq}_{\Sigma Cur}) \times 2$$
$$\begin{array}{c}\Sigma z : (\Pi x : A.\Pi y : B[y \leftarrow x].C[x \leftarrow (x,y)]).(\Pi x : A.\Pi y : B[y \leftarrow x]. \\ D[z \leftarrow (z\ x)][z \leftarrow \lambda c : (\Sigma y : A.B).z\ (\pi_1\ c)\ (\pi_2\ c)][x \leftarrow (x,y)]\end{array} \quad \textit{(2)}$$

Pour que cette paire critique conflue, les D[...] doivent se réécrire sur un type commun. Pour la substitution de x, il n'y a pas de problème car c'est la même dans les deux cas. Quant à z, prenons une de ses occurences où nous avons l'application $z\ a_1 ... a_n$ avec $FV(a_i) = \emptyset$. Après application des substitutions, on a:

(1) $z\ x\ y\ a_1 ... a_n$
(2) $(\lambda c : (\Sigma y : A.B).z\ (\pi_1\ c)\ (\pi_2\ c))\ (x,y)\ a_1 ... a_n$

A ce moment-là, on peut encore appliquer $\mathcal{R}^{Coq}_{\beta\iota Rd}$ sur (2), ce qui nous donne:

(1) $z\ x\ y\ a_1 ... a_n$
(2) $z\ (\pi_1\ (x,y))\ (\pi_2\ (x,y)))\ a_1 ... a_n$

Enfin, $\mathcal{R}^{Coq}_{\beta\iota Rd}$ encore appliquée deux fois sur (2) nous donne:

(1) $z\ x\ y\ a_1 ... a_n$
(2) $z\ x\ y\ a_1 ... a_n$

Les deux expressions sont identiques, ce qui montre que cette paire critique conflue. Les autres preuves sont du même ordre.

□

Théorème 4.2.11 (Terminaison et formes normales) *$\mathcal{R}^{Coq}$ termine et les formes normales sont de la forme:*

$$\Sigma \overrightarrow{x} : \overrightarrow{X}.T$$

Où $\overrightarrow{X}$ et T ne contiennent ni β-redex ni Σ ni unit.

Preuve *Pour montrer que $\mathcal{R}^{Coq}$ termine, il suffit de montrer que tout type admet une $\mathcal{R}^{Coq}$-forme normale étant donné que $\mathcal{R}^{Coq}$ est confluent. Soit A un type de Coq. On peut le β-normaliser (la β-réduction est fortement normalisable dans CCI). Par la suite, d'autres β-redex peuvent apparaître lors de l'application des règles $\mathcal{R}^{Coq}_{\Pi L}$ et $\mathcal{R}^{Coq}_{\Sigma L}$, mais on peut β-normaliser les parties droites des lieurs (où sont apparus les β-redex) en appliquant une succession de $\mathcal{R}^{Coq}_{\Pi R}$ ou $\mathcal{R}^{Coq}_{\Sigma R}$ qui elles ne créent pas de β-redex. Une fois cette β-normalisation préliminaire effectuée, on applique en alternance $\mathcal{R}^{Coq}_{\Sigma Cur}$ et $\mathcal{R}^{Coq}_{\Pi Dis}$ au maximum. Ce processus termine car $\mathcal{R}^{Coq}_{\Sigma Cur}$ ne peut s'appliquer qu'un nombre fini de fois (elle élimine des Σ) et une fois que le terme est en $\mathcal{R}^{Coq}_{\Sigma Cur}$-forme normale, $\mathcal{R}^{Coq}_{\Pi Dis}$ ne s'applique aussi qu'un nombre fini de fois (le nombre de Π à distribuer). On obtient alors un type où les Π ne sont plus à gauche d'un Σ et où les Σ ne sont plus arguments d'un Π, c'est-à-dire un type de la forme:*

$$\begin{array}{rl} type_0 ::= & \Sigma x : type_1.type_1 \\ type_1 ::= & type_0 \\ & \mid type_2 \\ type_2 ::= & \Pi x : type_2.type_2 \\ & \mid unit \\ & \mid others \end{array}$$

Où others désigne tout autre type sans Σ, sans Π, sans unit et sans β-redex. Ensuite, on peut utiliser au maximum $\mathcal{R}^{Coq}_{\Sigma Ass}$, ce qui termine dans la mesure où l'on diminue le nombre de Σ à gauche des Σ. D'où, la forme suivante:

$$\begin{array}{rl} type_0 ::= & \Sigma x : type_2.type_1 \\ type_1 ::= & type_0 \\ & \mid type_2 \\ type_2 ::= & \Pi x : type_2.type_2 \\ & \mid unit \\ & \mid others \end{array}$$

Enfin, il ne reste plus qu'à utiliser les règles relatives à unit. On applique d'abord $\mathcal{R}^{Coq}_{\Pi Unr}$ *dans toutes les composantes des* Σ *(on termine car le nombre de* Π *diminuent). Puis, toutes les autres règles à savoir* $\mathcal{R}^{Coq}_{\Sigma Unr}$, $\mathcal{R}^{Coq}_{\Sigma Unl}$ *et* $\mathcal{R}^{Coq}_{\Pi Unl}$ *peuvent être utilisées (le processus termine car le nombre de unit diminue). Finalement, on obtient une* $\mathcal{R}^{Coq}$*-forme normale définie comme suit:*

$$
\begin{aligned}
type_0 &::= \Sigma x : type_2.type_1 \\
type_1 &::= type_0 \\
&\quad\mid type_2 \\
type_2 &::= \Pi x : type_2.type_2 \\
&\quad\mid others
\end{aligned}
$$

□

On est en mesure maintenant de donner une possible implémentation pour Th^{Coq}:

Définition 4.2.12 (Procédure de décision) *On se donne la procédure de décision* Dec^{Coq} *qui étant donné deux types, les* $\mathcal{R}^{Coq}$*-normalise, puis les compare modulo* $Th^{Coq}_{\Sigma Alp}$, $Th^{Coq}_{\Pi Alp}$, $Th^{Coq}_{\Sigma Com}$, $Th^{Coq}_{\Sigma Ass}$ *et l'axiome suivant déductible de* Th^{Coq}*:*

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

Théorème 4.2.13 (Correction) Dec^{Coq} *est correcte vis-à-vis de* Th^{Coq}.

Preuve *La correction est immédiate. Pour la réduction,* Dec^{Coq} *utilise les règles de* $\mathcal{R}^{Coq}$*, qui ne sont autres que des axiomes de* Th^{Coq}*. Quant à la comparaison,* Dec^{Coq} *se sert de quatre axiomes de* Th^{Coq} *et de* $Th^{Coq}_{\Pi Swa}$ *qui se déduit de* Th^{Coq} *(par* $Th^{Coq}_{\Sigma Com}$ *et* $Th^{Coq}_{\Sigma Cur}$*).*

□

Théorème 4.2.14 (Complétude) Dec^{Coq} *est complète vis-à-vis de* Th^{Coq}.

Proposition de preuve *On effectue un raisonnement par récurrence sur la longueur de la dérivation dans* Th^{Coq}*:*

- $Th^{Coq}_{\beta Red}$, $Th^{Coq}_{\Sigma Ass}$, $Th^{Coq}_{\Sigma Cur}$, $Th^{Coq}_{\Pi Dis}$, Th^{Coq}_{*Un*}*:* A *se réécrit en* B *par les règles correspondantes de* $\mathcal{R}^{Coq}$ *et la comparaison est triviale;*
- Th^{Coq}_{*Alp}*: après avoir* $\mathcal{R}^{Coq}$*-normalisé* A *et* B*, on a égalité car on compare modulo* Th^{Coq}_{*Alp}*;*

- $Th^{Coq}_{\Sigma Com}$: *A et B sont deux produits cartésiens non dépendants. La* $\mathcal{R}^{Coq}$*-normalisation peut se faire dans chacune de leurs composantes puis à la racine à la fin. Ainsi, soit on obtient encore un produit cartésien et on utilise* $Th^{Coq}_{\Sigma Com}$ *pour conclure, soit il ne reste qu'une composante (issue d'une simplification avec unit) et la comparaison est triviale;*
- Th^{Coq}_{Ref}, Th^{Coq}_{Sym}, Th^{Coq}_{Trs}: *trivial en utilisant le fait que l'égalité finale utilisée par* Dec^{Coq} *est réflexive, symétrique et transitive;*
- Th^{Coq}_{*Cnl}: *on sait que:*

$$A =_{\binom{\sigma}{\tau}} A'$$

et que $A \overset{Dec^{Coq}}{=} A'$*, ce qui signifie que:*

$$A \overset{(\mathcal{R}^{Coq})^*}{\longrightarrow}_{\binom{\sigma_A}{\tau_A}} nf_{\mathcal{R}^{Coq}}(A)^1$$

$$A' \overset{(\mathcal{R}^{Coq})^*}{\longrightarrow}_{\binom{\sigma_{A'}}{\tau_{A'}}} nf_{\mathcal{R}^{Coq}}(A')$$

Avec $nf_{\mathcal{R}^{Coq}}(A)$ *et* $nf_{\mathcal{R}^{Coq}}(A')$ *égaux modulo la comparaison finale de* Dec^{Coq} *avec* σ' *et* τ' *comme fonctions de conversion. On veut prouver que* $*x : A.B \overset{Dec^{Coq}}{=} *x : A'.B[x \leftarrow (\tau\ x)]$. *On va supposer que* $\mathcal{R}^{Coq}_{\Sigma Cur}$ *ne s'applique pas dans B ni dans* $B[x \leftarrow (\tau\ x)]$ *(ce qui est sans perte de généralité car il suffit alors d'appliquer* $\mathcal{R}^{Coq}_{\Sigma Cur}$ *dans B avant* $\mathcal{R}^{Coq}_{\Pi Dis}$ *dans A et A'). On peut donc écrire en utilisant* $\mathcal{R}^{Coq}_{*L}$:

$$*x : A.B \overset{\mathcal{R}^{Coq}}{\longrightarrow}_{\binom{\sigma''}{\tau''}} *x : (nf_{\mathcal{R}^{Coq}}(A)).B[x \leftarrow (\tau_A\ x)]$$

$$*x : A'.B[x \leftarrow (\tau\ x)] \overset{\mathcal{R}^{Coq}}{\longrightarrow}_{\binom{\sigma'''}{\tau'''}} *x : (nf_{\mathcal{R}^{Coq}}(A')).B[x \leftarrow (\tau\ (\tau_{A'}\ x))]$$

τ *peut s'exprimer comme* $\sigma_{A'} \circ \tau' \circ \tau_A$*, d'où, en remplaçant et en simplifiant, l'expression suivante:*

$$*x : A'.B[x \leftarrow (\tau\ x)] \overset{\mathcal{R}^{Coq}}{\longrightarrow}_{\binom{\sigma'''}{\tau'''}} *x : (nf_{\mathcal{R}^{Coq}}(A')).B[x \leftarrow (\tau_A\ (\tau'\ x))]$$

En continuant la réécriture, on obtiendra deux formes normales que Dec^{Coq} *identifiera car le terme obtenu précédemment montre qu'on a mémorisé (par* τ'*) l'égalité de* $nf_{\mathcal{R}^{Coq}}(A)$ *et de* $nf_{\mathcal{R}^{Coq}}(A')$*;*

[1] $nf_{\mathcal{R}^{Coq}}(A)$ représente la $\mathcal{R}^{Coq}$-forme normale de A.

- $Th_{\Pi Cnr}^{Coq}$: *on suppose que* $A \stackrel{Dec^{Coq}}{=} A'$ *et on veut montrer que* $\Pi x : B.A \stackrel{Dec^{Coq}}{=} \Pi x : B.A'$. *Comme* A *et* A' *sont des types corrects,* x *n'intervient ni dans* A *ni dans* A'. *On peut donc* $\mathcal{R}^{Coq}$*-normaliser* B *sans interférer avec la* $\mathcal{R}^{Coq}$*-normalisation de* A *et* A'. *A la fin, il ne reste plus qu'à distribuer* $nf_{\mathcal{R}^{Coq}}(B)$ *sur* $nf_{\mathcal{R}^{Coq}}(A)$ *et* $nf_{\mathcal{R}^{Coq}}(A')$ *par* $\mathcal{R}_{\Pi Dis}^{Coq}$. *Comme* $nf_{\mathcal{R}^{Coq}}(A)$ *et* $nf_{\mathcal{R}^{Coq}}(A')$ *sont égaux modulo la comparaison finale de* Dec^{Coq}, *les termes ainsi obtenus sont ègaux (on distribue les mêmes* Π*);*

- $Th_{\Sigma Cnr}^{Coq}$: *c'est le même principe que pour l'axiome précédent excepté qu'à la fin on applique* $\mathcal{R}_{\Pi Ass}^{Coq}$. *La comparaison consiste alors en deux comparaisons: les composantes de* $nf_{\mathcal{R}^{Coq}}(B)$ *avec celles de* $nf_{\mathcal{R}^{Coq}}(B)$ *après* $\mathcal{R}_{\Pi Ass}^{Coq}$ *(trivialement égales entre elles), les composantes de* $nf_{\mathcal{R}^{Coq}}(A)$ *avec celles de* $nf_{\mathcal{R}^{Coq}}(A')$ *après* $\mathcal{R}_{\Pi Ass}^{Coq}$ *(elles sont égales par hypothèse de récurrence et du fait que* $\mathcal{R}_{\Pi Ass}^{Coq}$ *a eu les mêmes effets sur* $nf_{\mathcal{R}^{Coq}}(A)$ *que sur* $nf_{\mathcal{R}^{Coq}}(A')$*);*

- Th_{*Cnb}^{Coq}: *ce sont les deux cas précédents dans lesquels on utilise l'hypothèse de récurrence pour comparer ce qui vient de la partie gauche.*

□

Théorème 4.2.15 (Terminaison) Dec^{Coq} *termine.*

Preuve *La* $\mathcal{R}^{Coq}$*-normalisation termine et les formes normales obtenues sont des* Σ *parenthésés à droite (d'après le théorème 4.2.11). Appliquer* $Th_{\Sigma Com}^{Coq}$ *et* $Th_{\Sigma Ass}^{Coq}$ *revient à générer toutes les permutations possibles (il y a des dépendances) des* Σ. *Ce nombre est à multiplier par le nombre de permutations des* Π *dans les composantes dont se charge la règle* $Th_{\Pi Swa}^{Coq}$. *Le résultat obtenu (dans le pire des cas) reste fini, le nombre de* Σ *et de* Π *étant fini. Reste le cas des règles* $Th_{\Sigma Alp}^{Coq}$ *et* $Th_{\Pi Alp}^{Coq}$. *Il est clair que l'on peut appliquer ces règles à l'infini sur chacune des deux formes normales pour obtenir l'égalité. L'astuce consiste donc soit à ramener ces règles à des problèmes d'unification d'ordre 1 triviaux soit à supprimer les noms de variables et à considérer uniquement des lieurs et des hauteurs de liaison (comme De Bruijn pour représenter les* λ*-termes dans AUTOMATH). Ainsi, la comparaison termine.*

□

Nous possédons désormais tous les éléments pour revenir à la propriété essentielle de cette section à savoir la décidabilité de Th^{Coq}:

Théorème 4.2.16 (Décidabilité) Th^{Coq} *est décidable.*

Preuve *Utiliser les théorèmes 4.2.13, 4.2.14 et 4.2.15.*

□

Chapitre 5

Implémentation

Dans ce chapitre, deux points seront abordés. Le premier consistera à comprendre les difficultés qu'engendre l'implémentation de Dec^{Coq} (étudiée au chapitre précédent) et à exposer l'algorithme de décision en mettant l'accent sur les choix et limitations qui ont pu être entrepris. Le deuxième s'attachera à la recherche en elle-même en décrivant, lorsque c'est nécessaire, certaines parties du code du sytème Coq, notamment son environnement.

5.1 La procédure de décision

5.1.1 Choix et limitations

L'implémentation de Dec^{Coq} passe inévitablement par l'implémentation de $\mathcal{R}^{Coq}$. Or, comme on a pu le voir au chapitre 4, $\mathcal{R}^{Coq}$ 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. En se référant au tableau 4.4, on constate que l'on ne peut même pas faire l'économie du calcul des fonctions σ (en effet seul τ intervient dans les termes réécrits) car dans les règles $\mathcal{R}^{Coq}_{\Pi L}$ et $\mathcal{R}^{Coq}_{\Pi B}$, l'expression du τ de conclusion utilise le σ en prémisse. Par ailleurs, on voit bien que l'on devra conserver, tout au long de la réécriture, le terme global pour appliquer progressivement les modifications imposées par la réécriture des sous-termes. Ainsi, on s'aperçoit que ce type de réécriture n'est pas forcément difficile à mettre en oeuvre (on sait bien ce qu'il y a à faire et comment le faire) mais on réalise aussi à quel point le processus est lourd. On ne peut pas se permettre d'implémenter un tel système qui correspond certes exactement à la théorie élaborée précédemment 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 certaines règles et ce, dans un ordre donné. Plus précisément:

Définition 5.1.1 (Réécriture contrainte) *$\mathcal{STR}^{Coq}_{Imp}$ est le système de réécriture basé sur $\mathcal{R}^{Coq}$ et vérifiant les propriétés suivantes:*

- *les règles $\mathcal{R}^{Coq}_{\Pi L}$, $\mathcal{R}^{Coq}_{\Sigma L}$, $\mathcal{R}^{Coq}_{\Pi B}$, $\mathcal{R}^{Coq}_{\Sigma B}$ sont supprimées;*
- *$\mathcal{R}^{Coq}_{\beta\iota Rd}$ est contextuelle;*
- *on a l'ordre partiel $\mathcal{R}^{Coq}_{\Sigma Cur}$, $\mathcal{R}^{Coq}_{*Un*}$ et $\mathcal{R}^{Coq}_{\Pi Dis}$.*

Remarque 5.1.2 *$\mathcal{R}^{Coq}_{\beta\iota Rd}$ peut être contextuelle car lorsqu'elle se fait à gauche d'un lieur, elle applique une fonction de conversion identité à la variable liée. Ainsi, une fois tous les β-redex éliminés aux occurences de la variable liée, on retrouve le terme initial.*

Théorème 5.1.3 (Confluence) *$\mathcal{STR}^{Coq}_{Imp}$ est confluent.*

Preuve *$\mathcal{R}^{Coq}_{\beta\iota Rd}$ étant contextuelle et ne possédant aucune paire critique avec une autre règle de $\mathcal{R}^{Coq}$, le théorème de Church-Rosser s'applique. Pour les autres règles, on suit l'ordre partiel en utilisant par couche la confluence de $\mathcal{R}^{Coq}_{\Sigma Cur}$, $\mathcal{R}^{Coq}_{*Un*}$ et $\mathcal{R}^{Coq}_{\Pi Dis}$.*

$\square$

Théorème 5.1.4 (Terminaison et formes normales) *$\mathcal{STR}^{Coq}_{Imp}$ termine et les formes normales sont de la forme:*

$$\Sigma \overrightarrow{x} : \overrightarrow{X}.T$$

Où T ne contient ni β-redex ni Σ ni unit et où $\overrightarrow{X}$ est de la forme:

$$\Pi \overrightarrow{y} : \overrightarrow{Y}.\Sigma \overrightarrow{z} : \overrightarrow{Z}.U$$

Où $\overrightarrow{Y}$, $\overrightarrow{Z}$ et U ne contiennent pas de β-redex et avec la condition que si $\overrightarrow{y} = \overrightarrow{0}$ alors $\overrightarrow{z} = \overrightarrow{0}$.

Preuve *Utiliser la preuve du théorème 4.2.11 en remarquant que la présence d'éventuels Σ et unit dans $\overrightarrow{X}$ est due à l'interdiction de réécrire à gauche d'un lieur.*

$\square$

Définition 5.1.5 (Procédure de décision) *Dec^{Coq}_{Imp} est la procédure de décision basée sur Dec^{Coq} en utilisant $\mathcal{STR}^{Coq}_{Imp}$ pour normaliser et en n'appliquant pas $Th^{Coq}_{\Sigma Com}$ et $Th^{Coq}_{\Sigma Ass}$ dans les composantes.*

Remarque 5.1.6 *Pour faire correctement la comparaison, il faudrait aussi pouvoir permuter les Σ dans les composantes. Mais la décision de supprimer cette possibilité traduit la volonté d'économiser le temps en évitant un empilement des factorielles de permutation (celle des Π et celle des Σ).*

Les propriétés de Dec^{Coq}_{Imp} sont telles qu'on peut s'y attendre:

Théorème 5.1.7 (Correction) *Dec^{Coq}_{Imp} est correcte vis-à-vis de Th^{Coq}.*

Preuve *Utiliser la preuve du théorème 4.2.13.*

□

Théorème 5.1.8 (Incomplétude) *Dec^{Coq}_{Imp} est incomplète vis-à-vis de Th^{Coq}.*

Preuve *Dans Th^{Coq}, on a l'égalité suivante:*

$$\Sigma x : (\Pi y : A.unit).B = B[x \leftarrow ((\lambda u : unit.\lambda y : A.tt)\ x)][x \leftarrow tt]$$

Cette égalité peut être prouvée par la dérivation qui suit:

$$\dfrac{\dfrac{}{\begin{array}{l}\Pi y : A.unit = \\ \begin{pmatrix} \lambda f : (\Pi y : A.unit).tt \\ \lambda u : unit.\lambda y : A.tt \end{pmatrix} \\ unit\end{array}}\ Th^{Coq}_{\Pi U nr}}{\begin{array}{l}\Sigma x : (\Pi y : A.unit).B = \\ \begin{pmatrix} \lambda c : (\Sigma x : (\Pi x : A.unit).B).((\lambda f : (\Pi y : A.unit).tt)\ (\pi_1\ c),\ \pi_2\ c) \\ \lambda c : (\Sigma x : unit.B[x \leftarrow ((\lambda u : unit.\lambda y : A.tt)\ x)]).((\lambda u : unit.\lambda y : A.tt)\ (\pi_1\ c),\ \pi_2\ c) \end{pmatrix} \\ \Sigma x : unit.B[x \leftarrow ((\lambda u : unit.\lambda y : A.tt)\ x)]\ (\pi^1)\end{array}}\ Th^{Coq}_{\Sigma C nl}$$

$$\dfrac{}{\begin{array}{l}\Sigma x : unit.B[x \leftarrow ((\lambda u : unit.\lambda y : A.tt)\ x)] = \\ \begin{pmatrix} \lambda c : (\Sigma x : unit.B[x \leftarrow ((\lambda u : unit.\lambda y : A.tt)\ x)]).(\pi_2\ c) \\ \lambda a : B[x \leftarrow ((\lambda u : unit.\lambda y : A.tt)\ x)][x \leftarrow tt].(tt, a) \end{pmatrix} \\ B[x \leftarrow ((\lambda u : unit.\lambda y : A.tt)\ x)][x \leftarrow tt]\ (\pi^2)\end{array}}\ Th^{Coq}_{\Sigma U nl}$$

$$\dfrac{(\pi^1)\ (\pi^2)}{\begin{array}{l}\Sigma x : (\Pi y : A.unit).B = \\ \begin{pmatrix} (\lambda c : (\Sigma x : (\Pi x : A.unit).B).((\lambda f : (\Pi y : A.unit).tt)\ (\pi_1\ c),\ \pi_2\ c)) \circ \\ (\lambda c : (\Sigma x : unit.B[x \leftarrow ((\lambda u : unit.\lambda y : A.tt)\ x)]).(\pi_2\ c)) \\ (\lambda c : (\Sigma x : unit.B[x \leftarrow ((\lambda u : unit.\lambda y : A.tt)\ x)]).((\lambda u : unit.\lambda y : A.tt)\ (\pi_1\ c),\ \pi_2\ c)) \circ \\ (\lambda a : B[x \leftarrow ((\lambda u : unit.\lambda y : A.tt)\ x)][x \leftarrow tt].(tt, a) \end{pmatrix} \\ B[x \leftarrow ((\lambda u : unit.\lambda y : A.tt)\ x)][x \leftarrow tt]\end{array}}\ Th^{Coq}_{Trs}$$

Pour Dec^{Coq}_{Imp}, cette égalité n'est pas valide. En effet, après $\mathcal{STR}^{Coq}_{Imp}$-normalisation, on obtient les deux termes suivants:

$$\Sigma x : (\Pi y : A.unit).nf_{\mathcal{STR}^{Coq}_{Imp}}(B)\ (1)$$

$$nf_{\mathcal{STR}^{Coq}_{Imp}}(B[x \leftarrow ((\lambda u : unit.\lambda y : A.tt)\ x)][x \leftarrow tt])\ (2)$$

Comme x est un terme, on peut dire que $nf_{\mathcal{STR}^{Coq}_{Imp}}(B)$ et $nf_{\mathcal{STR}^{Coq}_{Imp}}(B[x \leftarrow ((\lambda u : unit.\lambda y : A.tt)\ x)][x \leftarrow tt])$ ont le même nombre de composantes. Donc, (1) a une composante de plus que (2), ce qui fera échouer la comparaison de Dec^{Coq}_{Imp}.

□

Théorème 5.1.9 (Terminaison) *Dec^{Coq}_{Imp} termine.*

Preuve *Utiliser le théorème 5.1.4 et la preuve du théorème 4.2.15.*

□

5.1.2 L'algorithme

On peut programmer Dec^{Coq}_{Imp} de la manière suivante:

dec_coq_imp*(typa,typb)*$\widehat{=}$
    *soit nfa=***normalisation***(typa)*
    *et nfb=***normalisation***(typb)*
    dans
        *retourner(***comparaison***(nfa,nfb)).*

normalisation*(typ)*$\widehat{=}$
    *retourner(***str_coq_imp***(***beta_red***(***inductif_red** *typ))).*

inductive_red a pour fonction de réécrire tous les types inductifs à un constructeur sans arité vers des Σ. Cette réécriture est susceptible de faire apparaître des β-redex, d'où l'application de **beta_red** qui β-normalise le terme ainsi réécrit. **str_coq_imp** réalise la $\mathcal{STR}^{Coq}_{Imp}$-normalisation et fonctionne de la manière suivante:

str_coq_imp*(typ)*$\widehat{=}$
    *soit (args,corps)=***curry** *typ*
        si corps=unit alors
            retourner(unit)
        sinon si args=[] alors
            *retourner(***elim_unit_sigma** *corps)*
        sinon
            *soit (args',corps')=***elim_unit_prod** *(args,corps)*
            dans
                *soit corps"=***elim_unit_sigma** *corps'*
                dans
                    *si (***sigma** *corps") alors*
                        *retourner(***distrib** *(args',corps"))*
                    sinon
                        *retourner(***cons_prod** *(args',corps")).*

curry currifie au maximum son argument (à la racine ou à la droite d'un Π) et rend un couple comprenant la liste des arguments (sous la forme (nom,type)) et le corps. **elim_unit_sigma** teste si le terme est sous la forme $\Sigma x : A.B$ et élimine les unit dans les composantes. **elim_unit_prod** élimine les unit dans les arguments. **sigma** teste si son argument est de la forme $\Sigma x : A.B$. **distrib** applique la distribution des arguments sur les Σ. Enfin **cons_prod** reconstruit un terme à partir de ses arguments et de son corps.

La comparaison s'effectue comme suit:

comparaison_(nfa,nfb)_$\hat{=}$
  _si (_**nbr_comp** _nfa)=(_**nbr_comp** _nfb) alors_
    reponse← faux;
    tant que (reponse=faux) faire
      _si (_**permut_possible** _nfa) alors_
        reponse:=(**terme_a_terme** _((_**permut** _nfa),nfb))_
      sinon
        retourner(faux)
    fintq;
    retourner(reponse)
  sinon
    retourner(faux).

nbr_comp calcule le nombre de composantes d'une $\mathcal{STR}^{Coq}_{Imp}$-forme normale. **permut_possible** vérifie si il y a encore des permutations possibles (compte-tenu des dépendances au niveau des composantes et des permutations déjà réalisées) de nfa. **permut** effectue une permutation possible et non encore essayée de nfa. **terme_a_terme** s'écrit ainsi:

terme_a_terme_(nfa,nfb)_$\hat{=}$
  pour chaque composante Ca_i _et_ Cb_i _faire_
    _si (_**anonyme** Ca_i_) et (_**anonyme** Cb_i_) alors_
      (si non(**eg_swap**_(_Ca_i,Cb_i_)) alors_
        retourner(faux))
    sinon si non(**anonyme** Ca_i_) et non(_**anonyme** Cb_i_) alors_
      (si non(**eg_swap**_(_Ca_i,Cb_i_)) alors_
        retourner(faux));
      mem_contrainte_(_Ca_i,Cb_i_)_
    sinon
      retourner(faux)
  finp;
  retourner(vrai)

anonyme vérifie si la composante n'a pas de nom de variable associé. **eg_swap** compare les types des composantes modulo permutation des arguments et une liste de contraintes du type (vara$_i$,varb$_i$) complétée lorsque les deux composantes ne sont pas anonymes. **mem_contrainte** ajoute une contrainte de variables.

Le code Objective Caml correspondant suit, en grande partie, le schéma de l'algorithme que nous venons de développer et le lecteur intéressé pourra trouver le module principal en annexe. Dans ce module, la procédure de décision **dec_coq_imp** est représentée par `iso_dep`.

5.2 La recherche

La première idée qui vient à l'esprit lorsque l'on veut faire de la recherche en bibliothèque, c'est d'utiliser les fichiers sources. C'est d'ailleurs ce que fait CamlSearch en explorant les fichiers d'interface `.mli`[1]. Cette solution n'a pas été envisagée en Coq dans la mesure où sa syntaxe est extensible (grâce à la commande `Grammar`[2]). Le parsing spécifique (c'est-à-dire en ne gardant que l'identificateur et son type) des `.v` n'est donc pas très aisé et avoir recours au parsing même de Coq n'est pas réellement souhaitable.

Une alternative immédiate est de prendre les fichiers objets (`.vo` en Coq). Toutefois, on ne peut les scanner sous cette forme et il faut les "dégeler" pour pouvoir les étudier. Cependant, ce n'est pas encore suffisant, car le type, que l'utilisateur aura saisi, sera parsé dans un contexte où les identificateurs des modules ne sont pas visibles. Il faut donc charger le module, parser le type utilisateur et scanner le contexte. Un problème apparaît: le module est chargé et l'utilisateur ne veut pas forcément qu'il le reste après sa recherche. Il faut donc le décharger après avoir fini de le scanner. Malheureusement, cette opération n'est pas intégralement réalisable en Coq à cause des fichiers ML qui sont chargés une fois pour toute dans le toplevel d'Objective Caml. Ainsi, on sent bien que deux outils de recherche sont nécessaires. Le premier est intégré au toplevel de Coq et se contente de faire la recherche dans le contexte courant. Le deuxième, totalement indépendant, contient toute la bibliothèque et est exclusivement dédié à la recherche.

C'est cette option qui a été effectivement choisie. Néanmoins, pour l'outil externe, des précautions sont à prendre. En effet, on ne peut sciemment pas charger tous les `.vo` de la bibliothèque (tous ceux qui sont accessibles à partir du `LoadPath`[3]) dans un contexte de Coq sous peine d'occuper une place mémoire très importante. L'économie a donc consisté à charger uniquement les fichiers d'interface (les `.vi`[4]) en appliquant un filtre supplémentaire qui ne considère que les variables, les constantes et les types mutuellement inductifs.

Pour en savoir plus sur les détails d'implémentation, on pourra se reporter à l'annexe. Pour connaître le mode d'utilisation et les performances des outils de recherche, on pourra lire le chapitre 6.

---

[1] Les fichiers `.mli` permettent de gérer la visibilité des identificateurs des fichiers `.ml` correspondants. Voir [XL94]

[2] Voir [BB+97]

[3] Le `LoadPath` est une liste de chemins que Coq consulte pour trouver les dépendances impliquées par le chargement ou la compilation d'un module. Il contient, entre autres, le chemin courant, les chemins de theories et les chemins de tactics/contrib. On le modifie grâce à la commande `AddPath` soit directement au toplevel soit de façon batch avec un fichier `.coqrc`. Voir [BB+97]

[4] Les fichiers `.vi` sont générés par la commande `coqc -i` et ne contiennent pas les termes preuves.

Chapitre 6

Documentation et jeux d'essais

Nous nous plaçons désormais du point de vue de l'utilisateur. Ainsi, dans un premier temps, nous allons voir comment utiliser les outils de recherche suivant les besoins qui se présentent et l'environnement travail. Puis, dans un deuxième temps , on s'intéressera aux résulats auxquels on doit s'attendre, compte-tenu de la théorie élaborée au chapitre 4.

6.1 Documentation

Comme on l'a déjà dit au chapitre 5, Search2 est en fait constitué de deux outils de recherche. Le premier fonctionne dans le toplevel de Coq (`coqtop`) et ne scanne que l'environnement courant. Il est invoqué de la manière suivante:

`Search2` <*type*>.

Le deuxième est un programme à part appelé `coq_search2_top` et compilé en même temps que Coq. A son lancement, `coq_search2_top` recherche tous les fichiers `.vi` présents dans le `LoadPath`[1] et les charge en les filtrant (il ne garde que les variables, constantes et types mutuellement inductifs) dans son environnement. L'utilisateur se trouve alors dans un toplevel similaire à `coqtop` où il peut bien-entendu déclencher la recherche comme précédemment mais aussi manipuler des informations relatives au `LoadPath` et imprimer des identificateurs visibles dans le contexte.

Intégré aux deux outils, la commande `Time` permet d'imprimer le temps de recherche (utilisateur et système) à chaque invocation de `Search2`. Cette option peut être déselectionnée par la commande `Untime`.

[1]La compilation s'occupe de génèrer en plus des `.vo` les `.vi` correspondants aux répertoires dans `theories` et `tactics/contrib`. Ces répertoires étant dans le `LoadPath`, il sont pris en compte dans le contexte de `coq_search2_top`. Pour les fichiers personnels, l'utilisateur doit non seulement songer à ce que ses répertoires soient dans le `LoadPath` mais aussi à générer les `.vi` de ses fichiers source.

6.2 Jeux d'essais

Les jeux d'essais se feront en deux temps. Une première partie va consister à tester, dans `coqtop`, les différents axiomes de Th^{Coq} dans un contexte et si possible indépendamment les uns des autres. La deuxième partie aura pour objectif de traiter, dans `coq_search2_top`, des exemples plus complexes et plus réalistes que l'utilisateur est susceptible de demander. La machine utilisée pour les tests est un Pentium Pro 200MHz et les exécutables sont en bytecode (un gain d'un facteur 5 est à attendre pour du natif).

6.2.1 Validité de la théorie

Le fichier suivant va nous permettre de tester chacune des règles de Th^{Coq}:

```
(*hypothesis_test_file.v*)

(**** for testing ****)
Variable A,B,C,D: Prop.
Variable E,F,G: Set.

(**** for beta-conversion ****)
Hypothesis H_conv0: A->([X:Set](X) nat)->B.
Hypothesis H_conv1: A->nat->B.

(**** for alpha-conversion with pi ****)
Hypothesis H_alpi0: (a:nat)(b:bool)(a=a/\b=b).
Hypothesis H_alpi1: (s:nat)(t:bool)(s=s/\t=t).

(**** for alpha-conversion with sigma ****)
Hypothesis H_alps0: (Ex [a:nat](Ex [b:nat](a=b))).
Hypothesis H_alps1: (Ex [s:nat](Ex [t:nat](s=t))).

(**** for sigma-commutativity ****)
Hypothesis H_swap_prod0: A->(B/\C)->D.
Hypothesis H_swap_prod1: A->(C/\B)->D.

(**** for sigma-associativity ****)
Hypothesis H_ass_prod0: A->(E*(F*G))->B.
Hypothesis H_ass_prod1: A->((E*F)*G)->B.

(**** for curry ****)
Hypothesis H_curry0: A->((a:{b:nat|b=b})(a=a)).
Hypothesis H_curry1:
  A->(a:nat)(b:a=a)
  ((exist nat ([a:nat](a=a)) a b)=(exist nat ([a:nat](a=a)) a b)).

(**** for distribution ****)
```

```
Hypothesis H_distrib0: A->(a:nat)(a=a/\a=a).
Hypothesis H_distrib1: A->((a:nat)(a=a))/\((a:nat)(a=a)).

(**** for sigma-unit-right ****)
Hypothesis H_sunitr0: A->E*unit.
Hypothesis H_sunitr1: A->E.

(**** for sigma-unit-left ****)
Hypothesis H_sunitl0: A->{a:unit|a=a}.
Hypothesis H_sunitl1: A->tt=tt.

(**** for pi-unit-right ****)
Hypothesis H_punitr0: A->B->unit.
Hypothesis H_punitr1: unit.

(**** for pi-unit-left ****)
Hypothesis H_punitl0: A->(a:unit)(a=a).
Hypothesis H_punitl1: A->tt=tt.
```

Nous pouvons maintenant lancer `coqtop`, sélectionner l'option `Time`, charger le fichier `hypothesis_test_file.v` et réaliser les tests:

```
Welcome to Coq V6.2 (May 15 1997)

Coq < Time.

Coq < Load "hypothesis_test_file".
A is assumed
B is assumed
C is assumed
D is assumed
E is assumed
F is assumed
G is assumed
H_conv0 is assumed
H_conv1 is assumed
H_alpi0 is assumed
H_alpi1 is assumed
H_alps0 is assumed
H_alps1 is assumed
H_swap_prod0 is assumed
H_swap_prod1 is assumed
H_ass_prod0 is assumed
H_ass_prod1 is assumed
H_curry0 is assumed
H_curry1 is assumed
```

```
H_distrib0 is assumed
H_distrib1 is assumed
H_sunitr0 is assumed
H_sunitr1 is assumed
H_sunitl0 is assumed
H_sunitl1 is assumed
H_punitr0 is assumed
H_punitr1 is assumed
H_punitl0 is assumed
H_punitl1 is assumed

Coq < Search2 A->([X:Set](X) nat)->B.
#--* [H_conv0 : A->([X:Set]X nat)->B]
#--* [H_conv1 : A->nat->B]
User Time: 0.99
System Time: 0

Coq < Search2 (a:nat)(b:bool)(a=a/\b=b).
#--* [H_alpi0 : (a:nat)(b:bool)a=a/\b=b]
#--* [H_alpi1 : (s:nat)(t:bool)s=s/\t=t]
User Time: 0.08
System Time: 0

Coq < Search2 (Ex [a:nat](Ex [b:nat](a=b))).
#--* [H_alps0 : (Ex [a:nat](Ex [b:nat]a=b))]
#--* [H_alps1 : (Ex [s:nat](Ex [t:nat]s=t))]
User Time: 0.09
System Time: 0

Coq < Search2 A->(B/\C)->D.
#--* [H_swap_prod0 : A->B/\C->D]
#--* [H_swap_prod1 : A->C/\B->D]
User Time: 0.1
System Time: 0

Coq < Search2 A->(E*(F*G))->B.
#--* [H_ass_prod0 : A->E*F*G->B]
#--* [H_ass_prod1 : A->(E*F)*G->B]
User Time: 0.1
System Time: 0

Coq < Search2 A->((a:{b:nat|b=b})(a=a)).
#--* [H_curry0 : A->(a:{b:nat | b=b})a=a]
#--* [H_curry1 :
A
 ->(a:nat)
    (b:a=a)(exist nat [a0:nat]a0=a0 a b)=(exist nat [a0:nat]a0=a0 a b)]
```

```
User Time: 0.13
System Time: 0

Coq < Search2 A->(a:nat)(a=a/\a=a).
#--* [H_distrib0 : A->(a:nat)a=a/\a=a]
#--* [H_distrib1 : A->((a:nat)a=a)/\((a:nat)a=a)]
User Time: 0.08
System Time: 0

Coq < Search2 A->E*unit.
#--* [H_sunitr0 : A->E*unit]
#--* [H_sunitr1 : A->E]
User Time: 0.08
System Time: 0

Coq < Search2 A->{a:unit|a=a}.
#--* [H_sunitl0 : A->{a:unit | a=a}]
#--* [H_sunitl1 : A->tt=tt]
#--* [H_punitl0 : A->(a:unit)a=a]
#--* [H_punitl1 : A->tt=tt]
User Time: 0.09
System Time: 0

Coq < Search2 A->B->unit.
#Datatypes#<unit>#tt:unit
#Logic#<True>#I:True
#Logic_Type#<UnitT>#IT:UnitT
#--* [H_punitr0 : A->B->unit]
#--* [H_punitr1 : unit]
User Time: 0.08
System Time: 0

Coq < Search2 A->(a:unit)(a=a).
#--* [H_sunitl0 : A->{a:unit | a=a}]
#--* [H_sunitl1 : A->tt=tt]
#--* [H_punitl0 : A->(a:unit)a=a]
#--* [H_punitl1 : A->tt=tt]
User Time: 0.09
System Time: 0

Coq <
```

6.2.2 Exemples concrets

On va lancer `coq_search2_top` et imaginer les requêtes 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

Coq_Search2 < Search2 (A:Set)(z,x,y:A)y=z->x=y->x=z.
#Logic#trans_equal:(A:Set)(x,y,z:A)x=y->y=z->x=z
User Time: 2.76
System Time: 0

Coq_Search2 < Search2 (P:Prop)P\/~P.
#Classical_Prop#--* [classic : (P:Prop)P\/~P]
User Time: 0.57
System Time: 0

Coq_Search2 < Search2 (B,A:Set)(y,x:A)x=y->(g:A->B)(g x)=(g y).
#Logic#f_equal:(A,B:Set)(f:A->B)(x,y:A)x=y->(f x)=(f y)
User Time: 1.27
System Time: 0

Coq_Search2 < Search2 (a:bool)a=false->a=true->False.
#Bool#--* [eq_true_false_abs : (b:bool)b=true->b=false->False]
User Time: 0.67
System Time: 0

Coq_Search2 < Search2 (X,Y:Prop)~(Y->X)->(decidable Y)->Y/\~X.
#auxiliary#--* [not_imp : (A,B:Prop)(decidable A)->~(A->B)->A/\~B]
User Time: 0.4
```

```
System Time: 0

Coq_Search2 < Search2 (A:Set)(list A)->A->(list A).
#basic_stuff#<list>#cons:(A:Set)A->(list A)->(list A)
User Time: 0.61
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.

Chapitre 7

Conclusion

7.1 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 Dec^{Coq}_{Imp} 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.

7.2 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. Avec cette option, de telles séquences de recherche seraient possibles:

```
Variable A,B,C,D: Prop.

Hypothesis H_fil0: A->B/\C->D.
Hypothesis H_fil1: A->B->C->D.
Hypothesis H_fil2: A->C->B->D.

Search2 A->('a)->D.
#--* [H_fil0 : A->B/\C->D]
```

```
#--* [H_fil0 : A->B->C->D]
#--* [H_fil0 : A->C->B->D]
```

Tout comme pour Th^{ML}, il faudra certainement opérer une réduction du problème d'unification qui, si il se fait directement modulo le sous-ensemble de Th^{Coq} pour lequel Dec^{Coq}_{Imp} est correct et complet, risque d'être indécidable (voir à ce propos et à titre d'exemple [PN97]).

- réécriture à droite lorsqu'il n'y a pas de dépendance: on a vu, au chapitre 5, que les réécritures à gauche d'un lieur non anonyme entraînaient des modifications à droite, ce qui obligeait à garder les termes de conversion tout au long de la $\mathcal{STR}^{Coq}$-normalisation. Souhaitant éliminer le calcul systématique des termes intervertibles, on a tout simplement empêcher les réécritures à gauche. C'est un peu trop restrictif, car lorsque le lieur est anonyme, aucune modification n'est à effectuer à droite. On peut donc imaginer un $\mathcal{STR}^{Coq}_{Imp}$ où les seules réécritures à droite d'un lieur se font pour $\mathcal{R}^{Coq}_{\beta Red}$ en toutes circonstances et pour toutes les autres règles si le lieur est anonyme. A ce moment-là, la théorie correspondante à cette implémentation sera un sur-ensemble de Th^{ML}.

- identifier plus de types inductifs: on aimerait bien parvenir à identifier les types inductifs à plusieurs constructeurs ayant la même structure (comme dans la preuve du théorème 4.2.7) ou possédant une forme différente mais construisant sémantiquement le même ensemble.

- suppression des `.vi`: à l'heure actuelle, `coq_search2_top` fonctionne exclusivement en recherchant tous les `.vi` accessibles depuis le `LoadPath`. L'utilisateur qui veut que ses fichiers soient pris en compte doit générer, en plus des `.vo`, les `.vi` (par `coqc -i`) correspondants. C'est un peu contraignant. C'est pourquoi, une autre technique consisterait à utiliser plutôt les `.vo` et à les filtrer au chargement[1].

[1]Pour ce faire, on utilise les fonctions de spécification associées à chaque type d'objets

Annexe A

Code du module principal

```
(***********************************************************************)
(*                The Calculus of Inductive Constructions              *)
(*                                                                     *)
(*                               Projet Coq                            *)
(*                                                                     *)
(*                     INRIA                       ENS-CNRS            *)
(*              Rocquencourt                       Lyon                *)
(*                                                                     *)
(*                                Coq V6.2                             *)
(*                              Jul 10th 1997                          *)
(*                                                                     *)
(***********************************************************************)
(*                               search2.ml                            *)
(***********************************************************************)

open Constrtypes;;
open Environ;;
open Generic;;
open Himsg;;
open Libobject;;
open Library;;
open More_util;;
open Names;;
open Pp;;
open Pretty;;
open Proof;;
open Reduction;;
open Std;;
open Summary;;
open System;;
open Term;;
open Vectops;;
```

```
(******** Updating of the inductive types table ********)
(* The list of the inductive types with one constructor *)
let tbl_ind_one=Hashtbl.create 53;;

(* Substitutes a rel by a term *)
let rec subs_rel rel term=function
   DOP1(a,b) -> DOP1(a,subs_rel rel term b)
  |DOP2(a,b,c) -> DOP2(a,subs_rel rel term b,subs_rel rel term c)
  |DOPN(a,b) -> DOPN(a,Array.map (subs_rel rel term) b)
  |DOPL(a,b) -> DOPL(a,List.map (subs_rel rel term) b)
  |DLAM(a,b) -> DLAM(a,subs_rel (rel+1) term b)
  |DLAMV(a,b) -> DLAMV(a,Array.map (subs_rel (rel+1) term) b)
  |(Rel a) as b -> if a=rel then term else b
  |_ as a -> a;;

(* Modify the unpractical types of constructors to correct types *)
let modify_constructors name pk mindlc=
  let (lna,lC)=decomp_all_DLAMV_name mindlc
  in
    let lst=ref (Array.to_list lC)
    and lgra=List.length lna
    in
      for cmr=0 to (lgra-1) do
        lst:=List.map (subs_rel (lgra-cmr) (DOPN(MutInd(path_of_string
             (name^"."^(string_of_kind pk)),cmr),Array.of_list [])))
             (!lst)
      done;
      !lst;;

(* Puts off all the Cast *)
let rec put_off_casts=function
   DOP1(a,b) -> DOP1(a,put_off_casts b)
  |DOP2(Cast,a,b) -> put_off_casts a
  |DOP2(a,b,c) -> DOP2(a,put_off_casts b,put_off_casts c)
  |DOPN(a,b) -> DOPN(a,Array.map put_off_casts b)
  |DOPL(a,b) -> DOPL(a,List.map put_off_casts b)
  |DLAM(a,b) -> DLAM(a,put_off_casts b)
  |DLAMV(a,b) -> DLAMV(a,Array.map put_off_casts b)
  |a -> a;;

(* Transforms a string to a list of char *)
let list_of_string str=
  let lst=ref []
  in
    for cmr=0 to ((String.length str)-1) do
      lst:=(!lst)@[String.get str cmr]
    done;
```

```
    !lst;;

(* Transforms a list of char in a string *)
let string_of_list lst=
  let str=String.create (List.length lst)
  in
    let rec fill cmr=function
        [] -> str
       |a::b ->
         String.set str cmr a;
         fill (cmr+1) b
    in
      fill 0 lst;;

(* Extract the name of the section_path *)
let extract sp=
  let lsts=list_of_string(string_of_path sp)
  and lstc=ref []
  in
    let rec inspect cmr=function
        [] -> failwith "extract"
       |a::b ->
         if cmr<4 then
           inspect (cmr+1) b
         else
           if not(a='#') then
             (lstc:=a::(!lstc);
              inspect cmr b)
    in
      inspect 0 (List.rev lsts);
      string_of_list(!lstc);;

(* Tests if the ind_one is really inductive *)
let rec occ name=function
   DOP1(_,a) -> occ name a
  |DOP2(_,a,b) -> (occ name a) or (occ name b)
  |DOPN(a,b) ->
    (match a with
        MutInd(c,_) ->
          (extract c)=name or (List.exists (occ name) (Array.to_list b))
       |MutConstruct((c,_),_) ->
          (extract c)=name or (List.exists (occ name) (Array.to_list b))
       |_ -> List.exists (occ name) (Array.to_list b))
  |DOPL(_,a) -> List.exists (occ name) a
  |DLAM(_,a) -> occ name a
  |DLAMV(_,a) -> List.exists (occ name) (Array.to_list a)
  |_ -> false
```

```
and is_rec name=function
   DOP2(Prod,a,DLAM(b,c)) ->
     (occ name a) or (is_rec name c)
  |_ -> false;;

(* For mutind_one and mutconstruct_one *)
exception Mutind_one;;
exception Mutconstruct_one;;

(* Tests if muti is a MutInd with one constructor and gives the new type
   corresponding *)
let mutind_one muti=
  let name=
    match muti with
       MutInd(a,_) -> extract a
      |_ -> failwith "mutind_one"
  in
    try (fst(Hashtbl.find tbl_ind_one name))
    with
       Not_found -> raise Mutind_one;;

(* Tests if mutc is the only MutConstruct of a MutInd and gives the new
   term corresponding *)
let mutconstruct_one mutc=
  let name=
    match mutc with
       MutConstruct((a,_),_) -> extract a
      |_ -> failwith "mutconstruct_one"
  in
    try (snd(Hashtbl.find tbl_ind_one name))
    with
       Not_found -> raise Mutconstruct_one;;

(* Rewrites all inductives types with one constructor to a generic pair
   Sigma with the constructor Csigma *)
let rec ind_rew=function
   DOP1(a,b) -> DOP1(a,ind_rew b)
  |DOP2(a,b,c) -> DOP2(a,ind_rew b,ind_rew c)
  |DOPN(a,b) ->
    (match a with
        MutInd(truc,_) ->
          (try
             (mutind_one a)
           with Mutind_one ->
              DOPN(a,Array.map ind_rew b))
       |MutConstruct((_,_),_) ->
         (try
```

```
            (mutconstruct_one a)
          with Mutconstruct_one ->
             DOPN(a,Array.map ind_rew b))
       |_ -> DOPN(a,Array.map ind_rew b))
  |DOPL(a,b) -> DOPL(a,List.map ind_rew b)
  |DLAM(a,b) -> DLAM(a,ind_rew b)
  |DLAMV(a,b) -> DLAMV(a,Array.map ind_rew b)
  |a -> a;;

(* Use ind_rew till the normal form *)
let total_ind_rew typ=
  let mem=ref typ
  and cal=ref typ
  in
    cal:=ind_rew(!mem);
    while not((!mem)=(!cal)) do
      mem:=(!cal);
      cal:=ind_rew(!mem)
    done;
    !mem;;

(* Transforms prod to (array,dest) *)
let decomp_prod_array term=
  let nbr=nb_prod term
  in
    let tab=Array.create nbr (Anonymous,DOP0(Implicit))
    and cmr=ref 0
    in
      let rec inspect=function
         DOP2(Prod,a,DLAM(b,c)) ->
           Array.set tab (!cmr) (b,a);
           incr cmr;
           inspect c
        |_ as a -> (tab,a)
      in
        inspect term;;

(* Tool for upd_lst_one in object MUTUALINDUCTIVE *)
let upd_mutual name mib=
 let {mINDKIND=pk ; mINDPACKETS=mipv ; mINDNPARAMS=nbparex}=mib
 in
   let chg_mutind nbpar typ_cons=
     let nbtpar=nb_prod typ_cons
     in
       let rec fill cmr=function
          DOP2(Prod,a,DLAM(b,c)) as d ->
            if cmr<nbpar then
```

```
          DOP2(Lambda,a,DLAM(b,fill (cmr+1) c))
         else
          if cmr<nbtpar-1 then
            DOP2(XTRA("Sigma",[]),a,DLAM(b,fill (cmr+1) c))
          else
            a
       |a->
         DOPN(MutInd(path_of_string "#Datatypes#unit.cci",0),
         Array.of_list [])
     in
       fill 0 typ_cons
and chg_mutconstruct nbpar typ_cons=
  let nbtpar=nb_prod typ_cons
  in
    let body nbr=
      let rec construct cmr=
        if cmr>1 then
          DOP2(XTRA("Csigma",[]),Rel cmr,construct (cmr-1))
        else
          Rel cmr
      in
        construct nbr
    in
      let rec fill cmr=function
         DOP2(Prod,a,DLAM(b,c)) ->
          if cmr<nbpar-1 then
            DOP2(Lambda,a,DLAM(b,fill (cmr+1) c))
          else if cmr=nbpar-1 then
            if (nbtpar-nbpar)=0 then
              DOP2(Lambda,a,DLAM(b,DOPN(MutConstruct((path_of_string
              "#Datatypes#unit.cci",0),1),Array.of_list [])))
            else
              DOP2(Lambda,a,DLAM(b,fill (cmr+1) c))
          else
            if cmr<nbtpar-1 then
              DOP2(Lambda,a,DLAM(b,fill (cmr+1) c))
            else
              DOP2(Lambda,a,DLAM(b,body (nbtpar-nbpar)))
        |a ->
          DOPN(MutConstruct((path_of_string "#Datatypes#unit.cci",0),
          1),Array.of_list [])
      in
        fill 0 typ_cons
in
  let explore_mip mip=
    let {mINDCONSNAMES=mcs ; mINDTYPENAME=typ ; mINDLC=mlc ;
         mINDARITY=ari}=mip
```

```
       in
         let nbpar=nb_prod ari
         and (par_tab,_)=decomp_prod_array (put_off_casts ari)
         in
           if ((Array.length mcs)=1)&((nbpar-nbparex)=0) then
             (let tpc=put_off_casts(List.hd (modify_constructors
                      (name^(string_of_id typ)) pk mlc))
              and name=string_of_id typ
              in
                if not(is_rec name tpc) then
                  Hashtbl.add tbl_ind_one name (total_ind_rew
                  (chg_mutind nbparex tpc),chg_mutconstruct nbparex
                  tpc))
     in
       if Array.length mipv = 1 then
         explore_mip mipv.(0)
       else
         Array.iter explore_mip mipv;;

(* Tool for upd_lst_one in Lib.leaf *)
let upd_leaf_entry name (spopt,lobj) =
  let tag = object_tag lobj in
  match (spopt,tag) with
    (_,"MUTUALINDUCTIVE") ->
    let (cmap,_) = outMutualInductive lobj in
    if Listmap.in_dom cmap CCI then
      upd_mutual name (Listmap.map cmap CCI)
  |_ -> ()
;;

(* Tools for upd_lst_one in Lib.node and Lib.library_segment *)
let rec upd_library_entry depth name=function
   (sp,Lib.LEAF lobj) -> upd_leaf_entry name (sp,lobj)
  |(_,Lib.ClosedDir(a,_,_,ctxt)) ->
    if depth=0 then
      upd_context 1 (name^a^"#") ctxt
  |_ -> ()
and upd_context depth name=
 let rec prec = function
    [] -> ()
   |h::rest ->
     upd_library_entry depth name h;
     prec rest
 in prec
;;

(* Update tbl_ind_one *)
```

```
let upd_tbl_ind_one ()=
  Hashtbl.clear tbl_ind_one;
  upd_context 0 "#" (List.rev(Lib.contents_after None));;

(******** Types' reduction ********)
(* Decrements all rel>nbr *)
let rec decr_rel nbr=function
   DOP1(a,b) -> DOP1(a,decr_rel nbr b)
  |DOP2(a,b,c) -> DOP2(a,decr_rel nbr b,decr_rel nbr c)
  |DOPN(a,b) -> DOPN(a,Array.map (decr_rel nbr) b)
  |DOPL(a,b) -> DOPL(a,List.map (decr_rel nbr) b)
  |DLAM(a,b) -> DLAM(a,decr_rel (nbr+1) b)
  |DLAMV(a,b) -> DLAMV(a,Array.map (decr_rel (nbr+1)) b)
  |(Rel a) as b ->
    if a>nbr then
      Rel(a-1)
    else
      b
  |a -> a;;

(* Elims the unit in the list *)
let elim_unit lst=
  let subs_tt lst=
    let rec inspect rel=function
       [] -> []
      |a::b ->
        (fst a,(subs_rel rel (DOPN(MutConstruct((path_of_string
        "#Datatypes#unit.cci",0),1),Array.of_list [])) (snd a)))::
        (inspect (rel+1) b)
    in
      inspect 1 lst
  in
    let rec iter=function
       [] -> []
      |a::b ->
        if (snd a)=(DOPN(MutInd(path_of_string "#Datatypes#unit.cci",0),
        Array.of_list [])) then
          if (fst a)=Anonymous then
            iter b
          else
            iter (subs_tt b)
        else
          a::(iter b)
    in
      iter lst;;

(* Transforms the normal form to list of components *)
```

```
let rec list_of_sigma=function
   DOP2(XTRA("Sigma",[]),a,DLAM(b,c)) ->
     [(b,a)]@(list_of_sigma c)
  |a -> [(Anonymous,a)];;

(* Apply associativity on a sigma term *)
let sigma_test term=
  let rec apply_assoc (name,body)=function
     DOP2(XTRA("Sigma",[]),a,DLAM(b,c)) ->
       DOP2(XTRA("Sigma",[]),a,DLAM(name,apply_assoc (b,(subs_rel 1
       (DOP2(XTRA("Csigma",[]),Rel 2,Rel 1)) body)) c))
    |a -> DOP2(XTRA("Sigma",[]),a,DLAM(name,snd(sigma_test_rec body)))
  and sigma_test_rec=function
     DOP2(XTRA("Sigma",[]),a,DLAM(b,c)) ->
       let resa=sigma_test_rec a
       in
         if (fst resa) then
           (true,apply_assoc (b,c) (snd resa))
         else
           (true,DOP2(XTRA("Sigma",[]),a,DLAM(b,(snd (sigma_test_rec
           c)))))
    |a -> (false,a)
  and elim_unit=function
     DOP2(XTRA("Sigma",[]),a,DLAM(b,c)) ->
       if c=(DOPN(MutInd(path_of_string "#Datatypes#unit.cci",0),
            Array.of_list [])) then
         a
       else
         if a=(DOPN(MutInd(path_of_string "#Datatypes#unit.cci",0),
         Array.of_list [])) then
           elim_unit (decr_rel 1 (subs_rel 1 (DOPN(MutConstruct
           ((path_of_string "#Datatypes#unit.cci",0),1),Array.of_list
           [])) c))
         else
           DOP2(XTRA("Sigma",[]),a,DLAM(b,elim_unit c))
    |a -> a
  and final_test=function
     DOP2(XTRA("Sigma",[]),_,_) -> true
    |_ -> false
  in
    let resterm=sigma_test_rec term
    in
      if (fst resterm) then
        (let newterm=elim_unit (snd resterm)
         in
           (final_test newterm,newterm))
      else
```

```
        resterm;;

(* Transforms (list,dest) to prod *)
let make_prod (lst,dest)=
  let rec inspect=function
     [] -> dest
    |a::b -> DOP2(Prod,snd a,DLAM(fst a,inspect b))
  in
    inspect lst;;

(* Applies curry *)
let rec apply_curry (name,body) lstres=function
   [] -> failwith "apply_curry"
  |a::b ->
    if b=[] then
      (lstres@[(name,snd a)],body)
    else
      apply_curry (fst a,subs_rel 1 (DOP2(XTRA("Csigma",[]),Rel 2,Rel
      1)) body) (lstres@[(name,snd a)]) b;;

(* Makes a list of rel n...1 *)
let make_rel nbr=
  let lst=ref []
  in
    for cmr=nbr downto 1 do
      lst:=(!lst)@[(Rel cmr)]
    done;
    !lst;;

(* Increments a list of subs *)
let incr_lst lstsubs=
  let rec incr_lst_rel=function
     [] -> []
    |a::b ->
      (match a with
          (Rel c) -> (Rel (c+1))::(incr_lst_rel b)
         |_ -> failwith "incr_lst")
  and incr_tot=function
     [] -> []
    |a::b ->
      ((fst a)+1,incr_lst_rel (snd a))::(incr_tot b)
  in
    incr_tot lstsubs;;

(* Makes the total substitution for distrib *)
let rec subs_tot lstsubs nbr=function
   DOP1(a,b) -> DOP1(a,subs_tot lstsubs nbr b)
```

```
  |DOP2(a,b,c) -> DOP2(a,subs_tot lstsubs nbr b,subs_tot lstsubs nbr c)
  |DOPN(a,b) ->
    if (a=AppL) then
      (match b.(0) with
          Rel c ->
            (try (DOPN(a,Array.of_list((List.assoc c lstsubs)@(List.map
                  (subs_tot lstsubs nbr) (List.tl (Array.to_list b))))))
             with
               Not_found -> DOPN(a,Array.map (subs_tot lstsubs nbr) b))
         |_ -> DOPN(a,Array.map (subs_tot lstsubs nbr) b))
    else
      DOPN(a,Array.map (subs_tot lstsubs nbr) b)
  |DOPL(a,b) -> DOPL(a,List.map (subs_tot lstsubs nbr) b)
  |DLAM(a,b) -> DLAM(a,subs_tot (incr_lst lstsubs) (nbr+1) b)
  |DLAMV(a,b) ->
    DLAMV(a,Array.map (subs_tot (incr_lst lstsubs) (nbr+1)) b)
  |(Rel a) as b ->
    (try (DOPN(AppL,Array.of_list(List.assoc a lstsubs)))
     with
        Not_found ->
          if a>nbr then
            Rel(a-nbr)
          else
            b)
  |a -> a;;

(* Puts name to anonymous prod *)
let put_name lstc=
  let rec modify cmr=function
     [] -> []
    |a::b ->
      if (fst a)=Anonymous then
        (Name (id_of_string ("#"^(string_of_int cmr))),snd a)::(modify
        (cmr+1) b)
      else
        a::(modify cmr b)
  in
    modify 0 lstc;;

(* Increments the first rel of lstsubs *)
let rec incr_fst_rel=function
   [] -> []
  |a::b -> ((fst a)+1,snd a)::(incr_fst_rel b);;

(* Applies distrib *)
let apply_distrib lstc termsig=
  let lgr=List.length lstc
```

```
  and newlstc=put_name lstc
  in
    let rec apply_distrib_rec lstsubs cmr=function
       DOP2(XTRA("Sigma",[]),a,DLAM(b,c)) ->
         DOP2(XTRA("Sigma",[]),make_prod (newlstc,subs_tot lstsubs cmr
         a),DLAM(b,apply_distrib_rec ((incr_fst_rel lstsubs)@[(1,
         make_rel (lgr+1))]) (cmr+1) c))
      |a -> make_prod (newlstc,subs_tot lstsubs cmr a)
    in
      apply_distrib_rec [] 0 termsig;;

(* Transforms a list of sigma to a sigma term *)
let rec sigma_of_list=function
   [] -> failwith "sigma_of_list"
  |a::b ->
    if b=[] then
      (snd a)
    else
      DOP2(XTRA("Sigma",[]),(snd a),DLAM(fst a,sigma_of_list b));;

(* New reduction for dependences, more performant I hope *)
let rec cur_dep lst=function
   DOP2(Prod,a,DLAM(b,c)) ->
     let (res,tms)=sigma_test a
     in
       if res then
         (let (lstres,bd)=apply_curry (b,c) [] (list_of_sigma tms)
          in
            cur_dep (lst@lstres) bd)
       else
         cur_dep (lst@[(b,a)]) c
  |a -> (lst,a)
and nf_dep term=
  let (lst,tc)=cur_dep [] term
  in
    if tc=(DOPN(MutInd(path_of_string "#Datatypes#unit.cci",0),
          Array.of_list [])) then
      DOPN(MutInd(path_of_string "#Datatypes#unit.cci",0),Array.of_list
      [])
    else if lst=[] then
      snd(sigma_test tc)
    else
      let lstt=elim_unit (lst@[(Anonymous,tc)])
      in
        let lstc=List.rev(List.tl (List.rev lstt))
        and lstb=(snd (List.hd (List.rev lstt)))
        in
```

```
          let (res,tms)=sigma_test lstb
          in
            if res then
              apply_distrib lstc tms
            else
              make_prod (lstc,tms);;

(******** Swap ********)
(* For permut *)
exception Permut_finished;;
exception Permut_impossible;;
exception Intern;;

(* Generates all permutations of 0..n-1 *)
let permut n=
  let cour=ref(Array.create n 0)
  and indc=ref 0
  and lstt=ref []
  and lstc=ref []
  in
    let init()=
      for cmr=0 to (n-1) do
        Array.set (!cour) cmr cmr
      done
    and swap()=
      let aux=Array.copy (!cour)
      in
        Array.set aux 0 (!cour).(!indc);
        Array.set aux (!indc) (!cour).(0);
        aux
    in
      let rec serve()=
        let tab=swap()
        in
          if (List.mem tab (!lstt)) then
            (incr indc;
             (if (!indc)=n then
                (cour:=
                   (try
                      (List.hd (!lstc))
                    with
                       Failure "hd" -> raise Permut_finished);
                 (try
                    (lstc:=(List.tl (!lstc)))
                  with
                     Failure "tl" -> lstc:=[]);
                 indc:=0));
```

```
            serve())
          else
            (lstt:=(!lstt)@[tab];
            lstc:=(!lstc)@[tab];
            incr indc;
            (if (!indc)=n then
               (cour:=
                  (try
                     (List.hd (!lstc))
                   with
                     Failure "hd" -> raise Permut_finished);
                (try
                   (lstc:=(List.tl (!lstc)))
                 with
                   Failure "tl" -> lstc:=[]);
                indc:=0));
            tab)
      in
        init();
        serve;;

(* Transforms array(permutation) to list of couples(always
   permutation) *)
let permut_array_list tab=
  let lst=ref []
  in
    for cmr=0 to ((Array.length tab)-1) do
      lst:=(!lst)@[(cmr,tab.(cmr))]
    done;
    !lst;;

(* Transforms (array,dest) to prod *)
let recompose (tab,dest)=
  let rec inspect=function
     [] -> dest
    |a::b -> DOP2(Prod,snd a,DLAM(fst a,inspect b))
  in
    inspect (Array.to_list tab);;

(* Returns the list of (Name, Negative Rel) *)
let ident_rel_list bgn term=
  let (tab,_)=decomp_prod_array term
  and lst=ref []
  and cmr=ref bgn
  in
    for aux=0 to ((Array.length tab)-1) do
      if not((fst (tab.(aux)))=Anonymous) then
```

```
        (lst:=(!lst)@[(fst (tab.(aux)),!cmr)];
         decr cmr)
    done;
    !lst;;

(* Puts the negative rel *)
let put_neg_rel lst_rel term=
  let rec apply lst_rel=function
     DOP2(Prod,a,DLAM(b,c)) ->
        if b=Anonymous then
          DOP2(Prod,a,DLAM(b,apply lst_rel c))
        else
          DOP2(Prod,a,DLAM(b,apply lst_rel (subs_rel 1 (Rel (List.assoc
          b lst_rel)) c)))
    |a -> a
  in
    apply lst_rel term;;

(* Puts off one negative rel *)
let rec subs_rel2 relneg relpos=function
   DOP1(a,b) -> DOP1(a,subs_rel2 relneg relpos b)
  |DOP2(a,b,c) ->
    DOP2(a,subs_rel2 relneg relpos b,subs_rel2 relneg relpos c)
  |DOPN(a,b) -> DOPN(a,Array.map (subs_rel2 relneg relpos ) b)
  |DOPL(a,b) -> DOPL(a,List.map (subs_rel2 relneg relpos) b)
  |DLAM(a,b) -> DLAM(a,subs_rel2 relneg (relpos+1) b)
  |DLAMV(a,b) -> DLAMV(a,Array.map (subs_rel2 relneg relpos) b)
  |(Rel a) as b -> if a=relneg then Rel(relpos) else b
  |_ as a -> a;;

(* Puts off the negative rel *)
let put_off_neg_rel lst term=
  let (tab,dest)=decomp_prod_array term
  in
    let taba=ref tab
    and desta=ref dest
    and lgr=Array.length tab
    in
      for cmr=0 to (lgr-1) do
        (if not((fst ((!taba).(cmr)))=Anonymous) then
           let (tabx,destx)=
             decomp_prod_array(subs_rel2 (List.assoc (fst
             ((!taba).(cmr))) lst) 1 (recompose (Array.sub (!taba)
             (cmr+1) (lgr-cmr-1),!desta)))
           in
             taba:=
               Array.append (Array.sub (!taba) 0 (cmr+1)) tabx;
```

```
            desta:=destx)
      done;
      recompose (!taba,!desta);;

(* Tells if relneg occurs in the term *)
let rec rel_neg_occur relneg=function
   DOP1(_,a) -> rel_neg_occur relneg a
  |DOP2(_,a,b) ->
    (rel_neg_occur relneg a) or (rel_neg_occur relneg b)
  |DOPN(_,a) -> List.exists (rel_neg_occur relneg) (Array.to_list a)
  |DOPL(_,a) -> List.exists (rel_neg_occur relneg) a
  |DLAM(_,a) -> rel_neg_occur relneg a
  |DLAMV(_,a) -> List.exists (rel_neg_occur relneg) (Array.to_list a)
  |(Rel a) -> a=relneg
  |_ -> false;;

(* Permuts a term *)
let permut_term lst termneg permcoup=
  let (tabneg,destneg)=decomp_prod_array termneg
  and tabcoup=Array.of_list permcoup
  and occur_in_array relneg tab=
    try
      (for cmr=0 to ((Array.length tab)-1) do
         if rel_neg_occur relneg (snd (tab.(cmr))) then
           raise Intern
       done);
      false
    with
       Intern -> true
  in
    let tabperm=ref (Array.copy tabneg)
    and lgr=Array.length tabneg
    in
      for cmr=0 to (lgr-1) do
        Array.set (!tabperm) cmr (tabneg.(snd (tabcoup.(cmr))))
      done;
      for cmr=0 to (lgr-1) do
        (if not((fst ((!tabperm).(cmr)))=Anonymous) then
          if (occur_in_array (List.assoc (fst ((!tabperm).(cmr))) lst)
             (Array.sub (!tabperm) 0 cmr)) then
            raise Permut_impossible)
      done;
      put_off_neg_rel lst (recompose (!tabperm,destneg));;

(* Equality between 2 terms modulo alpha-conversion and universes *)
let rec eq_constr_rec2 (m,n) = if eq(m,n) then true else
  match (strip_head_cast m,strip_head_cast n) with
```

```
     (DOP2(Cast,c1,_),c2)        -> eq_constr_rec2(c1,c2)
   | (c1,DOP2(Cast,c2,_))              -> eq_constr_rec2(c1,c2)
   | (Rel p1,Rel p2)                   -> p1=p2
   | (DOPN(oper1,cl1),DOPN(oper2,cl2)) ->
     oper1=oper2 & for_all2eq_vect (curry eq_constr_rec2) cl1 cl2
   | (DOP0 oper1,DOP0 oper2)           ->
     (match (oper1,oper2) with
         (Sort(Type a),Sort(Type b)) -> true
        |(_,_) -> oper1=oper2)
   | (DOP1(i,c1),DOP1(j,c2))           -> (i=j) & eq_constr_rec2(c1,c2)
   | (DOP2(i,c1,c1'),DOP2(j,c2,c2'))   ->
                (i=j) & eq_constr_rec2(c1,c2) & eq_constr_rec2(c1',c2')
   | (DLAM(_,c1),DLAM(_,c2))         -> eq_constr_rec2(c1,c2)
   | (DLAMV(_,cl1),DLAMV(_,cl2))       ->
                for_all2eq_vect (curry eq_constr_rec2) cl1 cl2
   | _         -> false
;;

(* Equality between 2 constrs modulo alpha-conversion and universes *)
let eq_constr2 ((m:constr),(n:constr)) = eq_constr_rec2(m,n);;

(* Small equality modulo universes *)
let eq_univ (n,m)=
  if n=m then
    true
  else
    match (n,m) with
       (DOP0 oper1,DOP0 oper2) ->
         (match (oper1,oper2) with
             (Sort(Type a),Sort(Type b)) -> true
            |(_,_) -> false)
      |(_,_) -> false;;

(* idem as List.assoc but modulo eq_univ *)
let rec assoc_univ elem=function
   [] -> raise Not_found
  |a::b ->
    if eq_univ(elem,fst a) then
      (snd a)
    else
      (assoc_univ elem b);;

(* Test if the type is dependant or not *)
let rec is_dependant=function
   DOP0 _ -> false
  |DOP1(_,a) -> is_dependant a
  |DOP2(_,a,b) -> (is_dependant a) or (is_dependant b)
```

```
  |DOPN(_,a) -> List.exists is_dependant (Array.to_list a)
  |DOPL(_,a) -> List.exists is_dependant a
  |DLAM(_,a) -> is_dependant a
  |DLAMV(_,a) -> List.exists is_dependant (Array.to_list a)
  |VAR _ -> false
  |Rel _ -> true;;

(* Give, for a prod type, the list of not-dependant types with their
   number of occurences *)
let list_not_dep typ=
  let (tab,_)=decomp_prod_array typ
  and lst=ref []
  and elem=ref (DOP0(Implicit))
  in
    let rec update_lst elem=function
        [] -> failwith "list_not_dep"
       |a::b ->
         if eq_univ((fst a),elem) then
           (fst a,(snd a)+1)::b
         else
           a::(update_lst elem b)
    in
      for cmr=0 to ((Array.length tab)-1) do
        elem:=snd(tab.(cmr));
        if not(is_dependant(!elem)) then
          if List.exists (function a -> eq_univ((!elem),fst a)) (!lst)
          then
            lst:=update_lst (!elem) (!lst)
          else
            lst:=(!lst)@[(!elem,1)]
      done;
      !lst;;

(* Optimization before compare modulo swap *)
let opt_not_dep typ0 typ1=
  let lst0=ref (list_not_dep typ0)
  and lst1=ref (list_not_dep typ1)
  in
    let rec put_off elem=function
        [] -> failwith "opt_not_dep"
       |a::b ->
         if eq_univ((fst a),elem) then
           b
         else
           a::(put_off elem b)
    and comp()=
      if (!lst0=[]) then
```

```
        true
      else
        (let elem=List.hd (!lst0)
         in
           try
             (if (assoc_univ (fst elem) (!lst1))=(snd elem) then
                (lst0:=List.tl (!lst0);
                 lst1:=put_off (fst elem) (!lst1);
                 comp())
              else
                false)
           with
              Not_found -> false)
    in
      if (List.length (!lst0))=(List.length (!lst1)) then
        comp()
      else
        false;;

(* Another optimization which compares the goals *)
let goals_comp typ0 typ1=
  let (_,dest0)=decomp_prod_array typ0
  and (_,dest1)=decomp_prod_array typ1
  and coh_lst=ref []
  in
    let rec is_coherent (rel0,rel1)=
      try
        (rel1=(List.assoc rel0 (!coh_lst)))
      with
        Not_found ->
          coh_lst:=(!coh_lst)@[(rel0,rel1)];
          true
    and comp_rec (n,m)=
      if eq(m,n) then
        true
      else
        match (strip_head_cast m,strip_head_cast n) with
          (DOP2(Cast,c1,_),c2)       -> comp_rec(c1,c2)
         | (c1,DOP2(Cast,c2,_))              -> comp_rec(c1,c2)
         | (Rel p1,Rel p2)                   ->
             (is_coherent (p1,p2))
         | (DOPN(oper1,cl1),DOPN(oper2,cl2)) ->
                oper1=oper2 & for_all2eq_vect (curry comp_rec) cl1 cl2
         | (DOP0 oper1,DOP0 oper2)           ->
           (match (oper1,oper2) with
               (Sort(Type a),Sort(Type b)) -> true
              |(_,_) -> oper1=oper2)
```

```
        | (DOP1(i,c1),DOP1(j,c2))           -> (i=j) & comp_rec(c1,c2)
        | (DOP2(i,c1,c1'),DOP2(j,c2,c2'))   ->
             (i=j) & comp_rec(c1,c2) & comp_rec(c1',c2')
        | (DLAM(_,c1),DLAM(_,c2))       -> comp_rec(c1,c2)
        | (DLAMV(_,cl1),DLAMV(_,cl2))       ->
             for_all2eq_vect (curry comp_rec) cl1 cl2
        | _       -> false
    in
      let comp_rec_constr ((m:constr),(n:constr))=comp_rec (m,n)
      in
        comp_rec_constr (dest0,dest1);;

(* Compare 2 types modulo swap *)
let swap_iso cmr equal (a,b)=
  let nba=nb_prod a
  and nbb=nb_prod b
  and aut=ref false
  and lst=ident_rel_list cmr a
  and permc=ref [||]
  in
    let aneg=put_neg_rel lst a
    in
      let perm=permut nba
      in
        if (nba=nbb) then
          if (nba=0) or (nba=1) then
            (equal(a,b),[||])
          else if (opt_not_dep a b)&(goals_comp a b) then
            (try
               (while not(!aut) do
                  try
                    (permc:=perm();
                     aut:=equal((permut_term lst aneg (permut_array_list
                          (!permc))),b))
                  with
                    Permut_impossible -> ()
                done;
                (!aut,!permc))
             with
               Permut_finished -> (false,[||]))
          else
            (false,[||])
        else
          (false,[||]);;

(******** Normal forms' comparison ********)
let nf_bse=ref (DOP0(Implicit));;
```

```
(* Gives the number of bounded variables with an array of occurences for
   a term *)
let bnd_var term=
  let lst=ref []
  in
    let rec update_lst rel_neg=function
        [] -> [(rel_neg,1)]
      |a::b ->
        if (fst a)=rel_neg then
          ((fst a),(snd a)+1)::b
        else
          a::(update_lst rel_neg b)
    and fill=function
        DOP0 _ -> ()
      |DOP1(_,a) -> fill a
      |DOP2(_,a,b) ->
        fill a;
        fill b
      |DOPN(_,a) -> List.iter fill (Array.to_list a)
      |DOPL(_,a) -> List.iter fill a
      |DLAM(_,a) -> fill a
      |DLAMV(_,a) -> List.iter fill (Array.to_list a)
      |VAR _ -> ()
      |Rel a -> lst:=update_lst a (!lst)
    and sort tab=
      let modf=ref true
      and cmr=ref 0
      and lgr=Array.length tab
      and aux=ref 0
      in
        while (!modf) do
          modf:=false;
          while (!cmr)<(lgr-1) do
            if tab.(!cmr)>tab.((!cmr)+1) then
              (aux:=tab.(!cmr);
               Array.set tab (!cmr) (tab.((!cmr)+1));
               Array.set tab ((!cmr)+1) (!aux);
               modf:=true);
            incr cmr
          done;
          cmr:=0
        done
    and fill_tab tab cmr=function
        [] -> ()
      |a::b ->
        Array.set tab cmr (snd a);
```

```
        fill_tab tab (cmr+1) b
    in
      fill term;
      (let tab=Array.create (List.length (!lst)) 0
       in
         fill_tab tab 0 (!lst);
         sort tab;
         (List.length (!lst),tab));;

(* Order on the arrays of occurences *)
let strict_gt_occ (tab0,tab1)=
  let ans=ref false
  in
    for cmr=0 to ((Array.length tab0)-1) do
      if tab0.(cmr)>tab1.(cmr) then
        ans:=true
    done;
    !ans;;

(* Returns a list of (Name,Negative Rel) *)
let rel_neg_lst lst_sigma=
  let lst=ref []
  and cmr=ref (-1)
  and tab_sigma=Array.of_list lst_sigma
  in
    for aux=0 to ((Array.length tab_sigma)-1) do
      if not((fst (tab_sigma.(aux)))=Anonymous) then
        (lst:=(!lst)@[(fst (tab_sigma.(aux)),!cmr)];
         decr cmr)
    done;
    (!lst,!cmr);;

(* Puts the negative Rels *)
let chg_neg_rel lst_rel lst_sigma=
  let rec one_rel rel_neg rel=function
     [] -> []
    |a::b ->
      if (fst a)=Anonymous then
        [(fst a,subs_rel rel (Rel rel_neg) (snd a))]@(one_rel rel_neg
        rel b)
      else
        [(fst a,subs_rel rel (Rel rel_neg) (snd a))]@(one_rel rel_neg
        (rel+1) b)
  and apply lst_rel=function
     [] -> []
    |a::b ->
      if (fst a)=Anonymous then
```

```
        a::(apply lst_rel b)
      else
        a::(apply lst_rel (one_rel (List.assoc (fst a) lst_rel) 1 b))
  in
    apply lst_rel lst_sigma;;

(* Forms the list of classes from list_sigma *)
let lst_classes neg_lst_sigma=
  let lst_cls=ref []
  in
    let rec update_lst_cls (cls,cpl)=function
       [] -> [(cls,[cpl])]
      |a::b ->
        if (fst(fst a))=(fst cls) then
          if (snd(fst a))=(snd cls) then
            (fst a,(snd a)@[cpl])::b
          else if strict_gt_occ (snd(fst a),snd cls) then
            (cls,[cpl])::a::b
          else
            a::(update_lst_cls (cls,cpl) b)
        else if (fst(fst a))>(fst cls) then
          (cls,[cpl])::a::b
        else
          a::(update_lst_cls (cls,cpl) b)
    and apply=function
       [] -> ()
      |a::b -> lst_cls:=update_lst_cls (bnd_var (snd a),a) (!lst_cls)
    in
      apply neg_lst_sigma;
      !lst_cls;;

(* Equality which constructs a list of coherence for Negative Rel *)
let equal_self (typ0,typ1)=
  let coh_lst=ref []
  in
    let rec is_coherent (rel0,rel1)=
      try
        (rel1=(List.assoc rel0 (!coh_lst)))
      with
        Not_found ->
          coh_lst:=(!coh_lst)@[(rel0,rel1)];
          true
    and comp_rec (n,m)=
      if eq(m,n) then
        true
      else
        match (strip_head_cast m,strip_head_cast n) with
```

```
          (DOP2(Cast,c1,_),c2)       -> comp_rec(c1,c2)
        | (c1,DOP2(Cast,c2,_))              -> comp_rec(c1,c2)
        | (Rel p1,Rel p2)                   ->
          if p1>0 then
            p1=p2
          else
            (is_coherent (p1,p2))
        | (DOPN(oper1,cl1),DOPN(oper2,cl2)) ->
               oper1=oper2 & for_all2eq_vect (curry comp_rec) cl1 cl2
        | (DOP0 oper1,DOP0 oper2)           ->
          (match (oper1,oper2) with
              (Sort(Type a),Sort(Type b)) -> true
             |(_,_) -> oper1=oper2)
        | (DOP1(i,c1),DOP1(j,c2))           -> (i=j) & comp_rec(c1,c2)
        | (DOP2(i,c1,c1'),DOP2(j,c2,c2'))   ->
               (i=j) & comp_rec(c1,c2) & comp_rec(c1',c2')
        | (DLAM(_,c1),DLAM(_,c2))       -> comp_rec(c1,c2)
        | (DLAMV(_,cl1),DLAMV(_,cl2))       ->
               for_all2eq_vect (curry comp_rec) cl1 cl2
        | _       -> false
    in
      let comp_rec_constr ((m:constr),(n:constr))=comp_rec (m,n)
      in
        comp_rec_constr (typ0,typ1);;

(* Optimization based on the idea of classes *)
(*let opt_cls (nlst0,cmr0) nlst1=
  let lst_cls0=lst_classes nlst0
  and lst_cls1=lst_classes nlst1
  in
    let rec comp_lists=function
        ((a::b),(c::d)) ->
          if (fst a)=(fst c) then
            if (List.length (snd a))=(List.length (snd c)) then
              comp_lists (b,d)
            else
              false
          else
            false
       |([],[]) -> true
       |(_,_) -> failwith "opt_cls"
    and comp_classes cls0 cls1=
      let rec one_list elem=function
          [] -> []
         |a::b ->
           if swap_iso cmr0 equal_self (snd a,snd elem) then
             b
```

```
          else
            a::(one_list elem b)
      and two_two lst0 lst1=
        let lgr0=List.length lst0
        and aux1=ref lst1
        and mem=ref lst1
        in
          try
            (for cmr0=0 to (lgr0-1) do
               aux1:=one_list (List.nth lst0 cmr0) !aux1;
               if (!mem)=(!aux1) then
                 failwith "fail_opt_cls"
               else
                 mem:=(!aux1)
             done;
             true)
          with
             Failure "fail_opt_cls" -> false
      in
        two_two (snd cls0) (snd cls1)
    in
      if (List.length lst_cls0)=(List.length lst_cls1) then
        if comp_lists (lst_cls0,lst_cls1) then
          List.for_all2 comp_classes lst_cls0 lst_cls1
        else
          false
      else
        false;;*)

(* Permuts a term of components *)
let permut_sigma_term lst_neg lst_sigma_neg permcoup=
  let tab_term=Array.of_list lst_sigma_neg
  and tabcoup=Array.of_list permcoup
  and occur_in_array relneg tab=
    try
      (for cmr=0 to ((Array.length tab)-1) do
         if rel_neg_occur relneg (snd (tab.(cmr))) then
           raise Intern
       done);
      false
    with
       Intern -> true
  in
    let tabperm=Array.copy tab_term
    and lgr=Array.length tab_term
    in
      for cmr=0 to (lgr-1) do
```

```
        Array.set tabperm cmr (tab_term.(snd (tabcoup.(cmr))))
      done;
      for cmr=0 to (lgr-1) do
        (if not((fst (tabperm.(cmr)))=Anonymous) then
          if (occur_in_array (List.assoc (fst (tabperm.(cmr))) lst_neg)
             (Array.sub tabperm 0 cmr)) then
            raise Permut_impossible)
      done;
      Array.to_list tabperm;;

(* Idem as List.assoc but with ((a,b),_) *)
let rec assoc2 a=function
   [] -> raise Not_found
  |((b,c),_)::d ->
    if a=b then
      c
    else
      assoc2 a d;;

(* Equality modulo a fixed list of coherence *)
let equal_comp coh_lst (typ0,typ1)=
  let rec is_coherent (rel0,rel1)=
    try
      (rel1=(assoc2 rel0 coh_lst))
    with
       Not_found ->
         failwith "equal_comp"
  and comp_rec (n,m)=
      if eq(m,n) then
        true
      else
        match (strip_head_cast m,strip_head_cast n) with
           (DOP2(Cast,c1,_),c2)       -> comp_rec(c1,c2)
         | (c1,DOP2(Cast,c2,_))              -> comp_rec(c1,c2)
         | (Rel p1,Rel p2)                   ->
           if p1>0 then
             p1=p2
           else
             (is_coherent (p1,p2))
         | (DOPN(oper1,cl1),DOPN(oper2,cl2)) ->
                oper1=oper2 & for_all2eq_vect (curry comp_rec) cl1 cl2
         | (DOP0 oper1,DOP0 oper2)           ->
           (match (oper1,oper2) with
               (Sort(Type a),Sort(Type b)) -> true
              |(_,_) -> oper1=oper2)
         | (DOP1(i,c1),DOP1(j,c2))           -> (i=j) & comp_rec(c1,c2)
         | (DOP2(i,c1,c1'),DOP2(j,c2,c2'))   ->
```

```
                (i=j) & comp_rec(c1,c2) & comp_rec(c1',c2')
         | (DLAM(_,c1),DLAM(_,c2))       -> comp_rec(c1,c2)
         | (DLAMV(_,cl1),DLAMV(_,cl2))       ->
                for_all2eq_vect (curry comp_rec) cl1 cl2
         | _       -> false
    in
      let comp_rec_constr ((m:constr),(n:constr))=comp_rec (m,n)
      in
        comp_rec_constr (typ0,typ1);;

(* Idem as List.assoc but with ((a,_),b) *)
let rec assoc3 a=function
   [] -> raise Not_found
  |((b,_),c)::d ->
    if a=b then
      c
    else
      assoc3 a d;;

(* Applies a permutation perm to tab *)
let apply_perm perm tab=
  let tabperm=Array.copy tab
  and lgr=Array.length tab
  in
    for cmr=0 to (lgr-1) do
      tabperm.(cmr) <- (tab.(perm.(cmr)))
    done;
    tabperm;;

(* Applies the coherence list to a term *)
let rec apply_coh_lst coh_lst=function
   DOP1(a,b) -> DOP1(a,apply_coh_lst coh_lst b)
  |DOP2(a,b,c) ->
    DOP2(a,apply_coh_lst coh_lst b,apply_coh_lst coh_lst c)
  |DOPN(AppL,a) ->
    (match (a.(0)) with
        Rel b ->
          (try
             (let lgr_par=(Array.length a)-1
              and perm=assoc3 b coh_lst
              in
                if (lgr_par=(Array.length perm)) then
                  DOPN(AppL,Array.map (apply_coh_lst coh_lst)
                  (Array.append [|Rel b|] (apply_perm perm (Array.sub a
                  1 lgr_par))))
                else
                  DOPN(AppL,Array.map (apply_coh_lst coh_lst) a))
```

```
          with
             Not_found ->
               DOPN(AppL,Array.map (apply_coh_lst coh_lst) a))
       |_ -> DOPN(AppL,Array.map (apply_coh_lst coh_lst) a))
  |DOPN(a,b) -> DOPN(a,Array.map (apply_coh_lst coh_lst) b)
  |DOPL(a,b) -> DOPL(a,List.map (apply_coh_lst coh_lst) b)
  |DLAM(a,b) -> DLAM(a,apply_coh_lst coh_lst b)
  |DLAMV(a,b) -> DLAMV(a,Array.map (apply_coh_lst coh_lst) b)
  |a -> a;;

(* Compares sigma expressions *)
let sigma_iso (nlst0,cmr0) nlst1 ns_lst0 ns_lst1=
  let coh_lst=ref []
  and mem=ref []
  in
    let term_to_term cpl0 cpl1=
      if (fst cpl0)=Anonymous then
        if (fst cpl1)=Anonymous then
          fst (swap_iso cmr0 (equal_comp (!coh_lst)) (apply_coh_lst
          (!coh_lst) (snd cpl0),(snd cpl1)))
        else
          false
      else
        if (fst cpl1)=Anonymous then
          false
        else
          (let res=swap_iso cmr0 (equal_comp (!mem)) (apply_coh_lst
                   (!coh_lst) (snd cpl0),(snd cpl1))
           in
             coh_lst:=(!coh_lst)@[((List.assoc (fst cpl0) nlst0,
                      List.assoc (fst cpl1) nlst1),snd res)];
             fst res)
    in
      (*if (opt_cls (ns_lst0,cmr0) ns_lst1) then*)
      if (List.length ns_lst0)=(List.length ns_lst1) then
        (let perm=permut (List.length ns_lst0)
         and ans=ref false
         in
           (try
              (while not(!ans) do
                 try
                   (ans:=List.for_all2 term_to_term (permut_sigma_term
                         nlst0 ns_lst0 (permut_array_list (perm())))
                         ns_lst1)
                 with
                    Permut_impossible -> ()
               done)
```

```
          with
             Permut_finished -> ());
          !ans)
    else
      false;;

(******** Decision procedure ********)
(* Renames if a variable occurs more than one time *)
let rename term=
  let rec rename_rec lstv=function
     DOP1(a,b) ->
       let resb=rename_rec lstv b
       in
         (fst resb,DOP1(a,snd resb))
    |DOP2(a,b,c) ->
      let resb=rename_rec lstv b
      in
        let resc=rename_rec (fst resb) c
        in
          (fst resc,DOP2(a,snd resb,snd resc))
    |DOPN(a,b) ->
      let lgr=Array.length b
      in
        if lgr=0 then
          (lstv,DOPN(a,b))
        else
          let aux=ref lstv
          and tab=Array.create lgr ([],DOP0(Implicit))
          in
            for cmr=0 to (lgr-1) do
              tab.(cmr) <- (rename_rec (!aux) (b.(cmr)));
              aux:=(fst (tab.(cmr)))
            done;
            (!aux,DOPN(a,Array.map snd tab))
    |DOPL(a,b) ->
      let c=Array.of_list b
      and lgr=List.length b
      in
        if lgr=0 then
          (lstv,DOPL(a,b))
        else
          let aux=ref lstv
          and tab=Array.create lgr ([],DOP0(Implicit))
          in
            for cmr=0 to (lgr-1) do
              tab.(cmr) <- (rename_rec (!aux) (c.(cmr)));
              aux:=(fst (tab.(cmr)))
```

```
            done;
            (!aux,DOPL(a,Array.to_list(Array.map snd tab)))
    |DLAM(a,b) ->
      (match a with
          Anonymous ->
            let resb=rename_rec lstv b
            in
              (fst resb,DLAM(a,snd resb))
         |Name c ->
           let aux=ref (string_of_id c)
           in
             while (List.mem (!aux) lstv) do
               aux:=("#"^(!aux))
             done;
             (let resb=rename_rec (lstv@[!aux]) b
              in
                (fst resb,DLAM(Name (id_of_string (!aux)),snd resb))))
    |DLAMV(a,b) ->
      let lgr=Array.length b
      in
        if lgr=0 then
          (lstv,DLAMV(a,b))
        else
          let aux=ref lstv
          and tab=Array.create lgr ([],DOPO(Implicit))
          in
            for cmr=0 to (lgr-1) do
              tab.(cmr) <- (rename_rec (!aux) (b.(cmr)));
              aux:=(fst (tab.(cmr)))
            done;
            (!aux,DLAMV(a,Array.map snd tab))
    |a -> (lstv,a)
  in
    snd (rename_rec [] term);;

(* Tests if user_type and nfb are isomorphic *)
let iso_dep nfb=
  let lsta=list_of_sigma(!nf_bse)
  and lstb=list_of_sigma(nfb)
  in
    let (nlsta,cmra)=rel_neg_lst lsta
    and (nlstb,_)=rel_neg_lst lstb
    in
      let ns_lsta=chg_neg_rel nlsta lsta
      and ns_lstb=chg_neg_rel nlstb lstb
      in
        sigma_iso (nlsta,cmra) nlstb ns_lsta ns_lstb;;
```

```
(******** Types' display ********)
(* The table of the normal forms *)
let rew_tab=Hashtbl.create 51;;

(* To know the search time *)
let seetime=ref false;;
let utimebe=ref 0.0;;
let stimebe=ref 0.0;;

(* Prints a variable and its type *)
let print_var2 name typ =
    [< 'sTR "*** [" ; 'sTR name ; 'sTR " : ";
     pTERM typ;
     'sTR "]" >];;

(* For print_mutual2 *)
let rec stream_map=function
   [] -> [<>]
  |a::b -> [< a ; stream_map b >];;

(* Applies iso_dep to mutual inductive types *)
let print_mutual2 name mib=
let pterm,pterminenv=(pTERM,pTERMINENV) in
 let {mINDKIND=pk; mINDPACKETS=mipv; mINDNPARAMS=nparams} = mib in
 let (lpars,_) = decomp_n_prod mt_evc nparams mipv.(0).mINDARITY in
 let lparsname = List.map fst lpars in
 let lparsprint = assumptions_for_print lparsname in
 let inspect_constructors mip=
   let lst=
     modify_constructors (name^(string_of_id mipv.(0).mINDTYPENAME)) pk
     mip.mINDLC
   in
     let lidC=
       List.map2 (fun id c -> (id,c)) (Array.to_list mip.mINDCONSNAMES)
       lst
     in
       let rec inspect_rec=function
           [] -> [<>]
          |a::b ->
            let nf=
              try
                (Hashtbl.find rew_tab (name^"<"^
                 (string_of_id mip.mINDTYPENAME)^">#"^(string_of_id (fst
                 a))))
              with
                 Not_found ->
```

```
                let nfcal=rename(nf_dep(Reduction.nf_betaiota(ind_rew
                          (Reduction.nf_betaiota (snd a)))))
                in
                  Hashtbl.add rew_tab (name^"<"^
                  (string_of_id mip.mINDTYPENAME)^">#"^(string_of_id
                  (fst a))) nfcal;
                  nfcal
        in
          if (iso_dep nf) then
            [< 'sTR (name^"<"^
               (string_of_id mip.mINDTYPENAME)^">#"^(string_of_id
               (fst a))^":") ; pterm (snd a) ; 'fNL ;
               inspect_rec b >]
          else
            inspect_rec b
    in
      inspect_rec lidC
in
let implicit_args_id id l =
     if l = [] then [<>]
     else [< 'sTR"For "; pID id; 'sTR": "; print_impl_args l ; 'fNL >]
in
let implicit_args_msg mipv =
 [< prlist
    (fun mip ->
 [< (implicit_args_id mip.mINDTYPENAME (list_of_implicits
    mip.mINDIMPLICITS));
    prlist (fun (idc,lc) ->
      (implicit_args_id idc (list_of_implicits lc)))
           (List.combine (Array.to_list mip.mINDCONSNAMES)
                         (Array.to_list mip.mINDCONSIMPLICITS)) >])
     (Array.to_list mipv) >] in
let print_oneind name namedis mip =
    let arity0=mip.mINDARITY
    and (_,arity1)=decomp_n_prod mt_evc nparams mip.mINDARITY in
      let nf=
        try
          (Hashtbl.find rew_tab (name^(string_of_id mip.mINDTYPENAME)))
        with
           Not_found ->
             let nfcal=rename(nf_dep(Reduction.nf_betaiota(ind_rew
                       (Reduction.nf_betaiota arity0))))
             in
               Hashtbl.add rew_tab (name^(string_of_id
               mip.mINDTYPENAME)) nfcal;
               nfcal
      in
```

```
    (if (iso_dep nf)  then
      (hOV 0 [< 'sTR namedis; pID mip.mINDTYPENAME ; if nparams = 0
       then [<>] else [< 'sTR" ["; print_env CCI (List.rev lpars);
                'sTR "]">]; 'bRK(1,5);  'sTR " : "; pterminenv
                (lparsprint,arity1) ; 'fNL ;
                inspect_constructors mip >] )
     else
       [< inspect_constructors mip >]) in
 let mip = mipv.(0) in
(* Case one [co]inductive *)
 if Array.length mipv = 1 then
   (let arity0=mip.mINDARITY
    and (_,arity1)=decomp_n_prod mt_evc nparams mip.mINDARITY in
      let nf=
         try
           (Hashtbl.find rew_tab (name^(string_of_id mip.mINDTYPENAME)))
         with
            Not_found ->
              let nfcal=rename(nf_dep(Reduction.nf_betaiota(ind_rew
                          (Reduction.nf_betaiota arity0))))
              in
                Hashtbl.add rew_tab (name^(string_of_id
                mip.mINDTYPENAME)) nfcal;
                nfcal
      in
        (if (iso_dep nf) then
      (let sfinite = if mip.mINDFINITE then "Inductive " else
                      "CoInductive " in
       (hOV 0 [< 'sTR (name^sfinite) ; pID mip.mINDTYPENAME ;
       if nparams = 0 then [<>]
                      else [< 'sTR" ["; print_env CCI (List.rev lpars);
                 'sTR "]">]; 'bRK(1,5); 'sTR ": ";
                 pterminenv (lparsprint,arity1) ; 'fNL ;
                 inspect_constructors mip >]))
     else
       [<inspect_constructors mip>]))
(* Mutual [co]inductive definitions *)
 else
  let (mipli,miplc) = List.fold_left
      (fun (li,lc) mi ->
         if mi.mINDFINITE then (li@[mi],lc) else (li,lc@[mi]))
      ([],[]) (Array.to_list mipv) in
        let strind =
          if mipli = [] then [<>]
          else  [< stream_map (List.map (print_oneind name
                (name^"Mutual Inductive ")) mipli) >]
      and strcoind =
```

```
        if miplc = [] then [<>]
        else  [< stream_map (List.map (print_oneind name
                   (name^"Mutual CoInductive ")) miplc) >]
  in
 (hOV 0 [< if mip.mINDFINITE then [< strind; strcoind >] else
        [<strcoind; strind>] >])
;;

(* Applies iso_dep to objects that are variable, constants or mutual
   inductive types *)
let print_leaf_entry2 name2 (spopt,lobj)=
  let tag = object_tag lobj in
  match (spopt,tag) with
    (_,"VARIABLE") ->
    let (name,(typ,_),_,l,_,_) = outVariable lobj
    in
      let nf=
        try
          (Hashtbl.find rew_tab (name2^(string_of_id name)))
        with
           Not_found ->
             let nfcal=rename(nf_dep(Reduction.nf_betaiota(ind_rew
                        (Reduction.nf_betaiota typ))))
             in
               Hashtbl.add rew_tab (name2^(string_of_id name)) nfcal;
               nfcal
      in
        if (iso_dep nf) then
          [< 'sTR name2 ; print_var2 (string_of_id name) typ;
            print_impl_args (list_of_implicits l) ; 'fNL>]
        else
          [<>]
  | (sp,"CONSTANT") ->
    let (cmap,_,_) = outConstant lobj in
    if Listmap.in_dom cmap CCI then
      let {cONSTBODY=val_0;cONSTTYPE=typ;cONSTIMPARGS=l} =
        Listmap.map cmap CCI
      in
        let nf=
          try
            (Hashtbl.find rew_tab (name2^(string_of_id (basename sp))))
          with
             Not_found ->
               let nfcal=rename(nf_dep(Reduction.nf_betaiota(ind_rew
                          (Reduction.nf_betaiota typ))))
               in
                 Hashtbl.add rew_tab (name2^(string_of_id
```

```
                  (basename sp))) nfcal;
                  nfcal
         in
           if (iso_dep nf) then
             hOV 0 [< (match val_0 with
               None -> [< 'sTR (name2^"--* [") ;
                          'sTR (string_of_id (basename sp)) ;
                          'sTR " : "; 'cUT ; pTERM typ ; 'sTR"]" >]
             | _    -> [< 'sTR(name2^(string_of_id (basename sp))) ;
                          'sTR ":" ; 'cUT ;  pTERM typ  >]);
                          print_impl_args (list_of_implicits l) ; 'fNL >]
           else
             [<>]
     else
       hOV 0 [< 'sTR(name2^"Fw constant ") ;
                'sTR (Printer.print_name (fwsp_of sp)) ; 'fNL>]

   | (sp,"MUTUALINDUCTIVE") ->
     let (cmap,_) = outMutualInductive lobj in
     if Listmap.in_dom cmap CCI then
       [< print_mutual2 name2 (Listmap.map cmap CCI) >]
     else
       hOV 0 [< 'sTR(name2^"Fw inductive definition ") ;
                'sTR (Printer.print_name (fwsp_of sp)) ; 'fNL >]
   |_ -> [<>]
;;

(* Searchs in context *)
let rec print_library_entry2 depth name=function
   (sp,Lib.LEAF lobj) ->
     (*print_endline (string_of_path sp);
     flush stdout;*)
     [< print_leaf_entry2 name (sp,lobj) >]
  |(_,Lib.ClosedDir(a,_,_,ctxt)) ->
    if depth=0 then
      if not(a="Cpo") then
        [< print_context2 1 (name^a^"#") ctxt >]
      else
        [<>]
    else
      [<>]
  |_ -> [<>]
and print_context2 depth name=
 let rec prec = function
    [] -> [<>]
   |h::rest -> [< print_library_entry2 depth name h ; prec rest >]
 in prec
```

```
;;

(* Search2 for coqtop *)
let type_search typ_bse=
  if (!seetime) then
    (let {Unix.tms_utime=tu;Unix.tms_stime=ts}=Unix.times()
     in
       utimebe:=tu;
       stimebe:=ts);
  upd_tbl_ind_one();
  nf_bse:=rename(nf_dep(Reduction.nf_betaiota(ind_rew
          (Reduction.nf_betaiota typ_bse))));
  mSG(print_context2 0 "#" (List.rev(Lib.contents_after None)));
  if (!seetime) then
    (let {Unix.tms_utime=tu;Unix.tms_stime=ts}=Unix.times()
     in
       print_string "User Time: ";
       print_float (tu-.(!utimebe));
       print_string "\nSystem Time: ";
       print_float (ts-.(!stimebe));
       print_newline();
       flush stdout)
;;

(* Search2 for coq_search2_top *)
let type_search2 typ_bse=
  if (!seetime) then
    (let {Unix.tms_utime=tu;Unix.tms_stime=ts}=Unix.times()
     in
       utimebe:=tu;
       stimebe:=ts);
  nf_bse:=rename(nf_dep(Reduction.nf_betaiota(ind_rew
          (Reduction.nf_betaiota typ_bse))));
  mSG(print_context2 0 "#" (List.rev(Lib.contents_after None)));
  if (!seetime) then
    (let {Unix.tms_utime=tu;Unix.tms_stime=ts}=Unix.times()
     in
       print_string "User Time: ";
       print_float (tu-.(!utimebe));
       print_string "\nSystem Time: ";
       print_float (ts-.(!stimebe));
       print_newline();
       flush stdout)
;;

(******** Light Require ********)
(* Filter which keeps only constants, variables and mutual inductive
```

```
   types *)
let filter_s node=
  let rec filter_node node=
    (match node with
        Lib.ClosedDir(st,od,cd,ls) ->
          Lib.ClosedDir(st,od,cd,filter_ls ls)
       |_ -> failwith "filter")
  and filter_ls=function
     [] -> []
    |a::b ->
      (match a with
          (_,Lib.LEAF obj) ->
            let tag=object_tag obj
            in
              if tag="CONSTANT" or tag="MUTUALINDUCTIVE" or
                 tag="VARIABLE" then
                a::(filter_ls b)
              else
                filter_ls b
         |(_,Lib.Import(_,_,_)) -> a::(filter_ls b)
         |_ -> filter_ls b)
  in
    filter_node node;;

(* Light internings *)
let rec intern_module_implementation2 s=
  intern_module_implementation_from2 s s
and intern_module_implementation_from2 s filename =
    let (sp,d) = raw_intern_module filename in
    (match d with
       Lib.ClosedDir(_,odc,_,_) as d ->
         let impbefore = needed_modules d in
         List.iter (fun sp -> if (not((module_is_loaded sp))) then
                      intern_module_implementation2(string_of_id
                      (basename sp))) impbefore;
         Lib.add_module (sp,d);
         cache_decl (sp,d)
     | _ -> failwith "library__intern_module2")
and intern_module_specification2 s = intern_module_specification_from2 s
                                      s
and intern_module_specification_from2 s filename =
    let (sp,d) = raw_intern_module_specification filename in
    if not(module_is_loaded sp) then
    (match (filter_s d) with
       Lib.ClosedDir(_,odc,_,_) as d ->
         let impbefore = needed_modules d in
         List.iter (fun sp ->
```

```
                      if not(module_is_loaded sp) then
                        intern_module_specification2(string_of_id
                        (basename sp)))
                  impbefore;
         Lib.add_module (sp,d);
         cache_decl (sp,d)
     | _ -> failwith "library__intern_module_specification2")
and intern_module2 spec s = intern_module_from2 spec s s
and intern_module_from2 spec s filename = match spec with
    Some true  -> intern_module_specification_from2 s filename
  | Some false -> intern_module_implementation_from2 s filename
  | None       ->
      begin if is_in_path (make_suffix filename ".vo") then
              intern_module_implementation_from2 s filename
            else if is_in_path (make_suffix filename ".vi") then
              intern_module_specification_from2 s filename
            else
              errorlabstrm "Library.intern_module2"
              (hOV 0 [< 'sTR"Can't find module" ; 'sPC ; 'sTR filename ;
                        'sPC ; 'sTR"on loadpath" >])
      end
and require_module2 spec name filename locally =
  if not (module_is_known name) then
    (match filename with
       Some f -> intern_module_from2 spec name f
     | None   -> intern_module2 spec name );
  if locally then import_module2 name else export_module2 name
and import_export_module2 s import_p =
  let anon_id = Lib.next_anonymous_id() in
  let isp = try lookup_module s
            with Failure _ -> errorlabstrm "import_module2"
                              [< 'sTR s ; 'sTR " not loaded." >] in
  if (not((Listset.memb !iMPORTED isp))) then
    cache_decl (Lib.add((anon_id,OBJ),Lib.Import(s,isp,import_p)))
  else
    ()
and import_module2 s = import_export_module2 s true
and export_module2 s = import_export_module2 s false;;

(******** Load the rew_tab ********)
(* Rewrite the type of the mutual inductive type and the types of its
   constructors *)
let load_mutual name mib=
let pterm,pterminenv=(pTERM,pTERMINENV) in
 let {mINDKIND=pk; mINDPACKETS=mipv; mINDNPARAMS=nparams} = mib in
 let (lpars,_) = decomp_n_prod mt_evc nparams mipv.(0).mINDARITY in
 let lparsname = List.map fst lpars in
```

```
 let lparsprint = assumptions_for_print lparsname in
 let inspect_constructors mip=
   let lst=modify_constructors (name^(string_of_id
           mipv.(0).mINDTYPENAME)) pk mip.mINDLC
   in
     let lidC=
       List.map2 (fun id c -> (id,c)) (Array.to_list mip.mINDCONSNAMES)
       lst
     in
       let rec inspect_rec=function
          [] -> ()
         |a::b ->
           Hashtbl.add rew_tab (name^"<"^(string_of_id
           mip.mINDTYPENAME)^">#"^(string_of_id(fst a))) (rename(nf_dep
           (Reduction.nf_betaiota(ind_rew(Reduction.nf_betaiota (snd
           a))))));
           inspect_rec b
       in
         inspect_rec lidC
 in
 let implicit_args_id id l =
      if l = [] then [<>]
      else [< 'sTR"For "; pID id; 'sTR": "; print_impl_args l ; 'fNL >]
 in
 let implicit_args_msg mipv =
  [< prlist
     (fun mip ->
  [< (implicit_args_id mip.mINDTYPENAME (list_of_implicits
     mip.mINDIMPLICITS)); prlist (fun (idc,lc) ->
       (implicit_args_id idc (list_of_implicits lc)))
            (List.combine (Array.to_list mip.mINDCONSNAMES)
                          (Array.to_list mip.mINDCONSIMPLICITS)) >])
      (Array.to_list mipv) >] in
 let print_oneind name mip =
     let arity0=mip.mINDARITY
     and (_,arity1)=decomp_n_prod mt_evc nparams mip.mINDARITY in
       Hashtbl.add rew_tab (name^(string_of_id mip.mINDTYPENAME))
       (rename (nf_dep(Reduction.nf_betaiota(ind_rew
       (Reduction.nf_betaiota arity0)))));
       inspect_constructors mip in
  let mip = mipv.(0) in
(* Case one [co]inductive *)
 if Array.length mipv = 1 then
   (let arity0=mip.mINDARITY
    and (_,arity1)=decomp_n_prod mt_evc nparams mip.mINDARITY in
      Hashtbl.add rew_tab (name^(string_of_id mip.mINDTYPENAME)) (rename
      (nf_dep(Reduction.nf_betaiota(ind_rew(Reduction.nf_betaiota
```

```
      arity0)))));
      inspect_constructors mip)
(* Mutual [co]inductive definitions *)
 else
  let (mipli,miplc) = List.fold_left
      (fun (li,lc) mi -> if mi.mINDFINITE then (li@[mi],lc) else (li,
       lc@[mi]))
      ([],[]) (Array.to_list mipv) in
    if mip.mINDFINITE then
      List.iter (print_oneind name) (mipli@miplc)
    else
      List.iter (print_oneind name) (miplc@mipli)
;;

(* Rewrite the types of the variables, constants and mutual inductive
   types *)
let load_leaf_entry name2 (spopt,lobj)=
  let tag = object_tag lobj in
  match (spopt,tag) with
    (_,"VARIABLE") ->
    let (name,(typ,_),_,l,_,_) = outVariable lobj
    in
      Hashtbl.add rew_tab (name2^(string_of_id name)) (rename(nf_dep
      (Reduction.nf_betaiota(ind_rew(Reduction.nf_betaiota typ)))))
  | (sp,"CONSTANT") ->
    let (cmap,_,_) = outConstant lobj in
    if Listmap.in_dom cmap CCI then
      let {cONSTBODY=val_0;cONSTTYPE=typ;cONSTIMPARGS=l} = Listmap.map
        cmap CCI
      in
        Hashtbl.add rew_tab (name2^(string_of_id (basename sp))) (rename
        (nf_dep (Reduction.nf_betaiota(ind_rew(Reduction.nf_betaiota
        typ)))))
  | (sp,"MUTUALINDUCTIVE") ->
    let (cmap,_) = outMutualInductive lobj in
    if Listmap.in_dom cmap CCI then
      load_mutual name2 (Listmap.map cmap CCI)
  |_ -> ()
;;

(* $Id: search2.ml,v 2.1 1997/09/02 17:10:54 delahaye Exp $ *)
```

Annexe B

Autres modules

Des modifications de modules existants ainsi que de nouveaux modules ont été nécessaires à la réalisation des outils de Search2. Nous ne donnerons pas ici le code de ces modules servant essentiellement à gérer les communications avec le système Coq. Nous pouvons toutefois exposer ce qui a été fait:

`~/V6.2/src/env/cmd_search2_line.ml`[1]
`~/V6.2/src/env/coq_search2_toplevel.ml`[1]
`~/V6.2/src/env/pfedit_search2.ml`[1]
`~/V6.2/src/env/vernacentries.ml`[2]
`~/V6.2/src/env/version_search2.ml`[1]
`~/V6.2/src/launch/coq_search2_top.ml`[1]
`~/V6.2/src/link/coq_search2_main.ml`[1]
`~/V6.2/src/parsing/g_search2_vernac.ml4`[1]
`~/V6.2/src/parsing/g_vernac.ml4`[2]

[1] Nouveau
[2] Modifié

Bibliographie

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

[JG72] J.-Y.Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse, Université Paris 7, 1972.

[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.

[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.