Master 2 MPRI
Université Paris VII – Denis Diderot

Rapport de Stage

Reconnaissance automatique de structures mathématiques dans l'assistant de preuve Coq

Matthias Puech

Soutenu le 5 septembre 2008

dir. Hugo Herbelin, Chargé de recherche INRIA,
Laboratoire d'Informatique de l'Ecole Polytechnique, Equipe TypiCal

Fiche de synthèse

Le contexte général

Au fur et à mesure de son développement et de son adaptation à la formalisation des mathématiques, l'assistant de preuve Coq devient de plus en plus lié à des notions mathématiques standard, par le biais notamment de tactiques spécialisées. Par exemple, la tactique ring de simplification d'expressions polynomiales repose sur la notion mathématique d'anneau. Plus simplement, les tactiques reflexivity, symmetry reposent sur la notion de relation et sur leurs propriétés très courantes.

Malgré l'ubiquité de ces notions mathématiques, peu d'outils existent dans Coq pour en faciliter l'utilisation, notamment pour identifier des structures (algébriques) standard, et plusieurs mécanismes ad-hoc ont été développés pour chaque tactique (déclaration de morphismes pour setoid_rewrite, d'anneaux pour ring...).

Dans tous les cas, la démarche de reconnaissance de structure est toujours une opération manuelle. Par exemple, après avoir prouvé qu'une relation est une équivalence, il faut *déclarer* la relation pour pouvoir utiliser les tactiques sur les équivalences, opération technique qui nous éloigne de la preuve papier traditionnelle.

Le problème étudié

Automatiser cette déclaration permettrait un confort d'utilisation accru, et une abstraction plus grande vis-à-vis des détails techniques de l'implémentation : une fois une structure mathématique définie, elle serait immédiatement reconnue par le système et on pourrait directement l'utiliser comme telle (sans avoir à la déclarer).

C'est sur cette reconnaissance automatique de structures mathématiques que s'est porté notre travail. A notre connaissance, de nombreux assistants à la preuve (dont Coq) embarquent des méthodes d'automatisation mais aucune n'est destinée à ce genre de reconnaissance de structures. Il semble pourtant intéressant de permettre l'omission d'un maximum de détails techniques propres aux mathématiques formelles, et rendre compte (dans ce cadre restreint) de la notion de "raisonnement trivial" pour un mathématicien.

La contribution proposée

Récemment, un cadre unifié a été proposé par Matthieu Sozeau pour tenter de répondre entre autres au besoin de classification plus nette des structures mathématiques : les *classes de types*. Elles permettent de définir des entités mathématiques (les classes), et d'organiser le contenu des scripts de preuve en instances de ces classes, grâce à un concept d'héritage proche de la surcharge inhérente au langage mathématique usuel. On *définit* donc des classes correspondant aux concepts mathématiques, puis on *déclare* des instances de ces objets.

Une application directe de ce mécanisme consiste à l'utiliser avec les tactiques qui dépendent de structures mathématiques, et unifier de ce fait toutes les solutions ad-hoc de déclaration de structures. Cela a été implémenté précédemment par M. Sozeau pour un certain nombre de tactiques courantes.

Notre proposition pour ce stage a donc été de se reposer sur le mécanisme de *classes de types* pour reconnaître automatiquement les structures mathématiques, en concevant et en implémentant une *procédure de recherche automatique d'instances de classes*. En effet, on pourra représenter de nombreuses définitions mathématiques (qu'est-ce qu'un groupe, un

morphisme ?) par une classe, et les structures particulières par des instances de cette classe. Rechercher parmi les définitions, lemmes prouvés, ... de possibles structures mathématiques revient alors à chercher automatiquement les instances de classes.

Cette procédure de recherche d'instance se doit d'être la plus intuitive possible pour respecter la simplicité du système, assez générale pour procurer un environnement extensible à l'ajout de nouvelles procédures de décision ou tactiques spécifiques, et efficace pour être aussi transparente que possible.

Les arguments en faveur de sa validité

Grâce à ce mécanisme, un certain nombre de tâches administratives sont évitées à l'utilisateur. Déclarer des relations et des morphismes (`Add Relation`/`Add Morphism`), et bientôt des anneaux, des arithmétiques (`ring`, `omega`) devient inutile si le système les a reconnu comme tel à leur définition.

De plus, le fait que cette solution ne soit pas l'extension d'une ou de plusieurs tactiques mais un mécanisme général reposant sur un cadre bien compris (les classes de types), elle est une extension pratique et pérenne du système, qui ne demande aucune modification lors de l'ajout de nouvelles tactiques, pourvu qu'elles reposent sur les classes de types.

Le bilan et les perspectives

Par son indépendance vis-à-vis des structures mathématiques en elles-même, par sa nature voulue la plus générale possible, cette solution semble flexible et pérenne, et pourra évoluer sans modification au gré des ajouts ultérieurs de tactiques spécifiques. Par son fonctionnement simple, elle reste toutefois dans les limites de ce que l'utilisateur peut en attendre, tout en éliminant le besoin de certaines déclarations administratives lourdes.

Malgré cela, certains aspects comme la souplesse de la capacité de reconnaissance reste à améliorer. Certains détails spécifiques à la logique et non au cadre mathématique pourraient être passés outre dans cette reconnaissance, comme l'ordre attendu des hypothèses d'un lemme, la décomposition des conjonctions etc. Un cadre théorique pour ces problèmes est celui des isomorphismes de types, qui a été envisagé pendant le stage.

Plus généralement, cette problématique s'inscrit dans une volonté de conception d'une intelligence mathématique dans les assistants à la preuve, complémentaire de la recherche de preuve et permettant de s'abstraire de la logique sous-jacente trop «bas niveau» pour manipuler plus aisément des concepts mathématiques évolués et plus proche du raisonnement mathématique usuel.

Table des matières

Objectif initial du stage

Partant du constat que certaines tactiques de l'assistants de preuve Coq reposent sur des concepts mathématiques (`ring`, `symmetry`...), et qu'elles apparaissent comme de simples extensions du système alors que les outils mathématiques qu'elles manipulent sont cruciaux, l'objectif initial de ce stage était de concevoir et d'implémenter au sein de Coq un mécanisme permettant de câbler ce genre de notions mathématiques plus profondément dans le système, afin que Coq sache détecter automatiquement par exemple que telle ou telle combinaison de constantes et d'opérations associatives, commutatives, distributives, ... forme un monoïde, un groupe, un anneau, etc. et que les simplifications propres à ce type de structure soient implicitement prises en compte dans les tactiques d'automatisation.

Présentation du travail effectué

Nous avons conçu, implémenté et testé une méthode générale, automatique et paramétrable de reconnaissance de structures mathématiques, reposant sur les *classes de types*, adaptée à n'importe quel énoncé ou définition, et donc extensible à de potentielles nouvelles tactiques d'automatisation.

Par le biais de cette reconnaissance, on a été amené à développer des outils de recherche dans le contexte plus efficaces grâce à une librairie de types. On a donc réimplémenté les fonctions de recherche (`Search`, `SearchPattern`, `SearchRewrite`...) de façon plus efficace, mais aussi plus complète et plus exacte.

Nous avons pu observer les avantages, mais aussi les manques à gagner de cette méthode, et nous avons envisagé les options et voies théoriques vers lesquelles poursuivre l'implémentation.

Estimation du temps passé sur les principales tâches

- Analyse du problème, lecture de documentation connexe *(10%)* ;
- Détermination du cadre, familiarisation avec les classes de types *(5%)* ;
- Familiarisation avec le code source de Coq, compréhension de son architecture globale *(20%)* ;
- Développement de la procédure de recherche *(30%)*
- Développement de bibliothèque de types *(25%)*
- Intégration au sein du système Coq *(10%)*

1 Cadre général

Commençons par quelques rappels sur les assistants à la preuve, et en particulier Coq, rappels qui pourront naturellement être sautés par le lecteur ayant déjà manipulé l'objet.

1.1 Contexte

1.1.1 Assistants à la preuve

Les assistants à la preuve sont des logiciels capables de vérifier un raisonnement logique formulé dans un langage propre. Ce langage est une *logique*. La propriété principale que l'on peut attendre d'une logique pour un assistant à la preuve est qu'elle puisse exprimer (coder) un maximum de concepts mathématiques, des plus triviaux aux plus abstraits.

Coq ([BBC+08]) est un assistant à la preuve basé sur une logique dérivée du λ-calcul : le calcul des constructions inductives, introduit dans [CP90]. La particularité de cette logique est qu'elle exprime dans le même langage les énoncés et les preuves de ces énoncés.

Cependant, la simplicité et la compacité de ce langage en font un outil de très bas niveau : il serait inconfortable (bien que possible) pour l'utilisateur d'écrire ses preuves dans ce langage. A donc été conçu un deuxième langage, plus proche de celui utilisé dans les preuves classiques sur papier, appelé *langage de tactique*. Il s'agit d'une palette d'outils, du plus simple au plus «intelligent», mettant en oeuvre des types de raisonnements connus des mathématiciens. Voici par exemple quelques tactiques courantes et leur signification informelle :

Tactique	**Signification**
`intro x.`	Soit x l'objet en question...
`assumption.`	... Ce qui est dans nos hypothèses
`apply H.`	D'après H, on a ...
`induction x.`	Par induction sur x, ...

A l'heure actuelle, on compte à peu près une centaine de tactiques et leurs variantes, accompagnées d'un langage pour les combiner ($\mathcal{L}_{tac}$).

1.1.2 Tactiques mathématiques

Ces tactiques, dans leur majorité, parlent d'objets appartenant au calcul des constructions inductives : induction, raisonnement par contradiction, par cas... s'énoncent comme des méthodes générale de formation de termes (de preuve), indépendemment de tout objet défini dans la logique.

D'autres tactiques, cependant, reposent sur des concepts définis dans le langage, souvent des concepts mathématiques. C'est le cas de la tactique `reflexivity` par exemple : cette tactique résout un but de la forme $x = x$. Or l'égalité n'est pas primitive en Coq, c'est une relation *définie* (dans la bibliothèque standard).

A l'autre bout du spectre, on peut aussi prendre l'exemple de `ring` : c'est une tactique de simplification d'expressions polynômiales exprimées sur des anneaux. Il est possible de l'utiliser directement sur des polynômes construits avec $+$ et $\times$ (ceux de $\mathbb{Z}$) car la bibliothèque standard déclare cet anneau.

Une constatation s'impose : si l'égalité de Leibnitz est sans doutes la relation réflexive la plus utilisée, ce n'est pas la seule, et l'anneau $(\mathbb{Z}, +, *)$ non plus. C'est pour cela qu'il existe un mécanisme, propre à chacune de ces tactiques, de déclaration de structure mathématique.

Voici les commandes de déclaration pour quelques tactiques mathématiques :

Tactique	Déclaration de structure
`field`	`Add Field`
`ring`	`Add Ring`
`reflexivity`	`Add Relation`
`rewrite`	`Add Morphism`

1.2 Enjeux

Cette déclaration est une opération manuelle, de l'ordre de l'administratif : il s'agit de faire connaître au système une structure que l'on a déjà définie pour qu'il l'ajoute à une base de données interne. Elle peut de ce fait être déroutante pour le mathématicien car elle ne correspond à rien dans les preuves classiques qu'il effectue.

Exemple 1 *On aura dans une preuve :*

> ***Lemme 1 :*** *la relation R est une équivalence*
> ***Preuve :*** *[...]*

Puis plus tard :

> ***Lemme 2 :*** *[...]*
> ***Preuve :*** *par transitivité de R*

La preuve du deuxième lemme fait intervenir l'énoncé du premier, mais l'on préfèrera énoncer le concept en jeu (la transitivité) plutôt qu'appliquer le premier lemme. De même dans Coq, on voudra pouvoir écrire :

```
Lemma R_equiv : equivalence R.
Proof. [...]. Qed.

Lemma foo : [...].
Proof. transitivity y; trivial. Qed.
```

Sans avoir à interposer entre les deux lignes une déclaration de la forme :

```
Add Parametric Relation x1 x2 : (A x1 x2) R
     reflexivity proved by R_refl
     symmetry proved by R_sym
     transitivity proved by R_trans
   as R_rel.
```

On se propose donc pour ce stage de mettre en place les mécanismes permettant la reconnaissance automatique de ces structures mathématique. Elle se base sur une reconnaissance textuelle de type filtrage, et sa généralité repose sur le nouveau concept de *classes de types* de Coq.

On présentera dans un premier temps le concept et l'utilité des classes de types, en programmation et dans le contexte des assistants de preuve. La deuxième partie concernera la procédure de recherche d'instance en elle-même ; on l'abordera d'un point de vue abstrait pour en montrer la puissance (et les limitations). On entrera alors dans un des aspects techniques soulevés par l'implémentation : comment rechercher efficacement un terme (une définition ou l'énoncé d'un lemme) dans une bibliothèque mathématiques. Cela nous amènera à présenter la nouvelle implémentation des outils de recherche, et leurs différences avec l'ancienne.

2 Les classes de types

Les classes de types (*type classes* en anglais) sont un mécanisme de surcharge introduit notamment dans le langage de programmation fonctionnel Haskell. Adapté récemment à Coq, il fournit un cadre pratique pour la formalisation mathématique. Notre reconnaissance de structures se reposant sur ce concept, on commencera par l'introduire, puis on montrera comment il peut être utile pour représenter les mathématiques.

2.1 Polymorphisme ad-hoc

Les classes de types ont été introduites en 1989 dans [WB89] pour tenter de donner une solution au problème du polymorphisme ad-hoc, autrement appelé surcharge. Elles ont été implémentées dans le langage Haskell en 1996 ([HHJW96]), et y ont rencontré un vif succès.

La notion de polymorphisme occupe une place centrale dans la théorie des langages de programmations. C'est une caractéristique des langages fonctionnels permettant d'utiliser différents types de données en utilisant une interface uniforme, le but étant de rendre possible l'écriture de code générique et réutilisable. On distingue deux types de polymorphismes :

le polymorphisme paramétrique permet de définir des fonctions dont le type n'est pas entièrement connu. Suivant leur instanciation en contexte, elles se spécifieront à un type particulier : en ML, la fonction *id* a le type polymorphe $\alpha \to \alpha$. La fonction effectuera dans tous les cas le même calcul.

le polymorphisme ad-hoc sert à exprimer le comportement d'une même fonction ou d'un même opérateur dans différents environnements. Par exemple, l'opérateur + a un sens différent en présence d'entiers et de nombres flottants : le calcul effectué dans un cas et dans l'autre n'est pas le même. En Caml, langage sans polymorphisme ad-hoc, on a au contraire deux opérateurs distincts `+` et `+.`. En Haskell, il est possible d'utiliser le même opérateur. Sa dénotation sera calculée en fonction du type des données auxquels on l'applique. Son nom vient du fait que l'on définit plusieurs fois la fonction avec le même nom, dans chaque situation où elle sera utile (pour les entiers, les flottants, les complexes...).

Les classes de types donnent un cadre puissant et flexible d'implémentation du polymorphisme ad-hoc. Le principe est de définir une relation «est un» entre des conteneurs (les classes) représentant les fonctions à surcharger et les objets effectifs du système (les instances). Cette relation est résolue à la compilation, et la bonne fonction est directement appelée dans le code final, sans perte de performance.

2.2 Présentation par l'exemple

En Haskell, on définit une *classe* comme un ensemble de méthodes qui s'appliquent à un objet d'un certain type. L'exemple le plus simple est celui de la classe des types affichables. On souhaite définir pour tous les types de données une fonction permettant son affichage (conversion en chaîne de caractère). Cette fonction se comportera différemment suivant le type rencontré : elle affichera les entiers sous forme décimale, les valeurs booléennes comme `true` et `false`, ...Voici la définition de la classe :

```
class Show a where
  show :: a -> string
```

Ici, le `a` est un lieur (il peut y en avoir plus), il correspond à un type de donnée quelconque. Tous les types qui feront partie de cette classe – qui seront *instances* de cette classe – devront fournir une méthode `show` permettant d'en afficher les éléments. Voici un exemple de déclaration d'instance :

```
instance Show bool where
  show b = if b then "true" else "false"
```

A partir de cette déclaration, on pourra invoquer sur tout objet de type `bool` la méthode `show`. C'est celle définie ci-dessus qui sera executée. Si l'on déclare par la suite d'autres instances pour d'autres types, on pourra les appliquer à `show`, et la bonne fonction sera choisie par le compilateur en fonction du type de son argument.

A ce stade déjà, la promesse est tenue : on peut définir la classe des types additionnables (et multipliables), puis déclarer deux instances pour `int` et `float`, chacuns avec une fonction + particulière :

```
class Addable a where
  (+) :: a -> a -> a

instance Addable Int where
  a + b = plus_int a b

instance Addable Float where
  a + b = plus_float a b
```

A l'appel de l'opérateur +, le compilateur servira de dispatcheur et choisira la bonne instance de + en fonction du type des arguments auquel il est appliqué.

Notons qu'il est bien sûr possible qu'une classe contienne plus d'une seule fonction. Il est aussi possible de définir génériquement une fonction sur une classe, comme dans l'exemple :

```
class  Eq a  where
  (==), (/=)  ::  a -> a -> Bool
  x /= y  = not (x == y)
```

Ici, l'opérateur `/=` est défini en fonction de `==`, il n'est donc pas nécessaire de le fournir lors de la déclaration d'instance.

Un degré plus avant, il est possible d'engendrer des définitions paramétrées par une instance. Elles ne s'appliqueront qu'aux instances d'une classe, et étendront donc la définition de

classes. Supposons que l'on ait besoin de multiplier par deux des nombres, flottants ou entiers. Inutile de définir deux fois l'opération, on se servira de l'addition surchargée :

```
two_times   :: (Addable a) => a -> a
two_times x = x + x
```

Le premier paramètre de la fonction précédant la double flèche `=>` indique que cette fonction `two_times` est définie si le type de son argument est instance de la classe `Addable`.

2.3 Héritage et hiérarchie de classes

2.3.1 Présentation

Mais la puissance des classes de types ne s'arrête pas là. Il est possible de tirer parti d'un mécanisme d'héritage et de sous-structures correspondant à la relation «est-un» entre deux classes, à la manière des langages orientés objets.

De nombreux concepts de programmation peuvent être pensés de cette façon modulaire. Si l'on dispose d'un ordre total sur un type, on est en mesure de trier un ensemble d'objets de ce type :

```
class Ord t where
  (<=) :: a -> a -> Bool

class (Ord t) => Sortable t where
  sort :: [t] -> [t]
```

On lit ce code de la façon suivante : t est ordonné si on dispose d'une fonction de comparaison sur ces éléments. t est triable si il est ordonné, et si l'on dispose d'une fonction de tri sur les listes de t. Autrement dit, la classe `Sortable` *hérite* des méthodes de la classe `Ord`. On dit que `Ord` est une *super-classe* de `Sortable`

Comment déclarer des instances de ces classes héritées ? Deux solutions :

- En déclarant deux instances indépendemment :
  ```
  instance Ord Int where
    [...]
  instance Sortable Int where
    sort l = [...]
  ```
  On devra déclarer pour chaque type ordonné une instance de tri. La contrepartie est que l'on pourra "personnaliser" la fonction de tri.
- En créant une instance paramétrée : Pour tous les types ordonnés, on donne une méthode de tri générique n'utilisant que les opérateurs de comparaison fournis par la super-classe :
  ```
  instance (Ord t) => Sortable t where
    sort l = quick_sort (<) l
  ```
  Ainsi, toute instance de `Ord` sera instance de `Sortable` grâce à la méthode générique `quick_sort`.

2.3.2 Résolution de la surcharge

Arrêtons-nous quelques instants sur ce dernier aspect, et voyons comment il a été possible de le mettre en oeuvre par l'exemple.

Sans héritage, la résolution des surcharges était simplement l'affaire de la consultation d'une table des instances à la compilation. Tapons par exemple dans la boucle interactive Haskell :

```
ghci> two_times "hello"
```

Le typage relèvera que `two_times` impose que le type son argument soit instance de `Addable`. Il ira donc chercher dans la table des instances une occurence de `Addable String`, qu'il ne trouvera pas, et échouera. Par contre, si l'on tape :

```
ghci> two_times 3.14
```

l'occurence `Addable Float` sera trouvé (car on en a défini l'instance précédemment) , et `two_times` sera remplacée par la méthode réelle pointée.

Dans le cas où l'on utilise l'héritage de classe, la résolution devient moins triviale ; il s'agit de faire une recherche dans la hiérarchie des instances. Demandons par exemple :

```
ghci> sort [3;2;1]
```

On cherche une instance de `Sortable Int`, qui n'est pas trouvée. Par contre, on sait que `Ord` $\alpha \Rightarrow$ `Sortable` α (par l'instance paramétrée). Donc on cherche une instance de `Ord Int`, que l'on trouve dans le prélude Haskell. On a enfin résolu la méthode `sort` : on utilisera `quick_sort lt_int`.

On voit ici que la résolution de surcharge nécessite une recherche parmis les instances déclarées. Cette recherche peut être vue comme, et a la puissance d'une *recherche de preuve* en PROLOG. Simulons le comportement précédent par une base de donnée et une requête à PROLOG :

```
sortable(int).
sortable(A) :- ord(A).

?- ord(int).
```

Cette recherche donne bien sûr une solution, qui est celle attendue.

On ne prétendra pas donner ici une description exhaustive de ce mécanisme de recherche, le but est simplement d'en faire prendre conscience au lecteur, et d'en donner l'intuition. On remarquera une dernière chose : ce mécanisme de recherche, et même l'ensemble du mécanisme de classes de type sont des objets à part dans la taxonomie Haskell. Il est impossible d'exprimer ce mécanisme directement en Haskell pour Haskell puisqu'il parle d'un objet (les classes) dont le type contient lui même un type. Ceci n'est pas exprimable dans Haskell car son système de type n'est pas assez puissant.

2.4 Portage des *type classes* à Coq

Dans le cadre de sa thèse, Matthieu Sozeau a récemment porté les classes de types dans Coq. Une description précise de l'implémentation ainsi qu'un aperçu des développements possibles est disponible ([SO08]). On n'en donnera ici les que grandes lignes.

2.4.1 Motivations

Pourquoi adapter une caractéristique d'un langage de programmation permettant l'écriture de code générique à un outil de preuve comme Coq, servant entre autre à formaliser des mathématiques ? Les avantages sont nombreux, car il existe des similitudes entre ces deux activités :

- Dans le langage mathématique courant, de nombreuse notation sont volontairement ambiguës, et résolues intuitivement par le lecteur. Par exemple, le symbole $\supset$ aura une signification différente en déduction naturelle et en théorie des ensemble. Il serait pratique de pouvoir utiliser dans Coq la même notation, et le laisser inférer de quelle notion il s'agit en fonction du contexte. Cela ressemble beaucoup au concept de polymorphisme ad-hoc...
- La notion de hiérarchie de concepts est très courante en mathématique : toute la hiérarchie des structures algébriques en est un exemple édifiant. L'héritage des *type classes* définit une hiérarchie par la relation *super-classe*.
- Enfin, les structures algébriques sont souvent construites comme des agrégats d'objets : un groupe est un ensemble, une opération binaire sur cet ensemble, et la preuve de quatre propriétés fondamentales. De la même façon, une classe est un agrégat de méthodes.

2.4.2 Implémentation

A la différence de Haskell, Coq est un langage de programmation fonctionnel possédant un système de type dépendant d'ordre supérieur. Grâce à cela, il a été possible de réaliser un portage *de première classe* des classes de types, c'est à dire un portage où les classes ne seront pas des objets "méta" mais feront bien partie du système au même titre que les définitions ou les lemmes. On se repose pour cela sur plusieurs mécanismes existants dans Coq :

- Les enregistrements dépendants, qui représenteront les classes ;
- Les arguments implicites, pour faire une partie de la résolution ;
- L'automatisation de type PROLOG (tactique `eauto`) pour la résolution de la surcharge en elle-même.

Sans rentrer dans les détails, on se contentera de donner ici une traduction d'un programme avec classes en un programme sans classes :

Exemple 2 ***Avec classes :***

```
Class Reflexive (A:Type) (R:A->A->Prop) :=
  reflexive : forall x, R x x.

Instance eq_nat_refl : Reflexive nat (@eq nat) :=
  reflexive := @refl_equal nat.

Definition foo : 2 + 2 = 4 := reflexive 4.
```

Sans classe :

```
Record Reflexive (A:Type) (R:A->A->Prop) :=
  { reflexive : forall x, R x x }.
```

```
Definition eq_nat_refl : Reflexive nat (@eq nat) :=
  Build_Reflexive nat (@eq nat) (@refl_equal nat).
Hint Resolve eq_nat_refl.

Definition foo : 2 + 2 = 4.
  assert (instance:Reflexive nat (@eq nat)). eauto.
  pose (reflexive_method:=instance.(reflexive nat (@eq nat))).
  exact (reflexive_method 4).
```

On notera que la recherche se fait grâce à la tactique `eauto` : c'est elle qui simule le mécanisme de résolution de surcharge en choisissant la bonne instance.

2.4.3 Tactiques mathématiques et classes

Ce nouveau mécanisme de classe donne un cadre assez général pour unifier les différentes méthodes manuelles de déclaration de structures exposées en introduction (`Add Morphism` etc.). Comme il est possible de représenter ces structures mathématiques par des classes, et déclarer des instances particulières de ces structures, on pourrait définir une/des classes pour chaque tactique, et considérer les instances de cette classe comme les objets sur lesquels la tactique peut opérer.

Considérons par exemple `reflexivity`. On peut théoriquement l'appliquer à n'importe quelle instance de la classe `Reflexive` présentée dans l'exemple plus haut. C'est le cas pratiquement depuis peu dans Coq, grâce à une nouvelle déclaration dans la bibliothèque standard et un changement de comportement dans le code de la tactique : Il suffit de déclarer notre relation comme instance de `Reflexive`.

Le même principe s'applique aux autres tactiques mathématiques, dans divers états de finitude de l'implémentation. L'idée est qu'il est simple de lier une tactique existante à une classe de types donnée, et qu'elles fournissent un cadre unifié pour ce type de problèmes.

3 Recherche d'instances et reconnaissance de motif

Exposons maintenant le coeur de ce projet de reconnaissance de structures mathématiques : la recherche automatique d'instances.

Dans le contexte où les tactiques mathématique opèrent sur les instances d'une classe donnée, reconnaître automatiquement des structures mathématiques revient à *reconnaître automatiquement les instances* de cette classe.

3.1 Notations

Pour présenter la recherche de façon théorique, on va omettre certains détails.

On introduit le concept de *signature* d'une classe. La signature d'une classe C une liste de jugements de typage $\langle x_1 : T_1 \dots x_n : T_n \rangle$, qui sont les arguments et les champs de C. La signature d'une sous-classe – une classe paramétrée par d'autres classes – est une liste ayant pour premiers éléments le type des super-classes (rappelons que les classes sont des enregistrements de première classe). Les méthodes surchargées dans la sous-classe sont désignées par leur projection (puisqu'on peut nommer l'instance explicitement).

Rappelons que les champs d'une classe sont dépendants : le type du i^e champ peut contenir le nom de tous les champs $j < i$. On peut voir ces noms comme des métavariables pour le type du champ. On dénotera par $\star$ l'univers `Type`.

Exemple 3 *La signature de la classe :*

```
Class Monoid (A:Type) :=
  prod : A -> A -> A;
  assoc : forall a b c, prod (prod a b) c = prod a (prod b c);
  e : A;
  ident_l : forall a, prod a e = a;
  ident_r : forall a, prod e a = a.
```

s'écrit :

$$\left\langle \begin{array}{rcl} A & : & \star \ ; \\ (*) & : & A \to A \to A \\ assoc & : & \forall abc, a * (b * c) = (a * b) * c \ ; \\ e & : & A \ ; \\ identl & : & \forall a, a * e = a \ ; \\ identr & : & \forall a, e * a = a \end{array} \right\rangle$$

Exemple 4 *La signature de la classe :*

```
Class [Monoid A] => Group :=
  inverse : forall x:A, exists y:A,
      x * y = y * x = e.
```

s'écrit :

$$\left\langle \begin{array}{rcl} mon & : & Monoid\ A \ ; \\ inv & : & (\forall x : A, \exists y : A,\ mon.prod\ x\ y = mon.prod\ y\ x = mon.e) \end{array} \right\rangle$$

Une instance i de C est une liste de termes $t_1 : T_1 \ \ldots \ t_n : T_n$, où $\langle x_1 : T_1; \ldots; x_n : T_n \rangle$ est la signature de C. On écrira $i : C$ ou (en version longue) inst $t_1 : T_1 \ \ldots \ t_n : T_n$. Notons que cette interprétation est très proche de ce qui est effectivement mis en place dans Coq, puisque l'implémentation des classes de type est telle qu'une classe est un enregistrement, et une instance un objet de ce type.

3.2 Principe algorithmique

Le parti-pris de cette méthode est de rechercher et de générer aussi tôt que possible des instances de classes, pour paver la route devant l'algorithme de résolution de surcharge. On se propose donc, à la définition d'un objet $t : T$ dans Coq (lemme, axiome, inductif, constructeur) de rechercher parmi les classes définies s'il est possible de faire intervenir t dans la déclaration d'une instance.

C'est un problème d'unification du premier ordre (un `evar_defs` pour les coquistes) : les signatures de classes sont des listes de couples variables d'unification - types ouverts.

Exemple 5 *La classe* `Monoid` *est perçue dans Coq par notre algorithme comme le problème d'unification suivant :*

```
?1==[ |- Type]
?2==[ |- ?1 -> ?1 -> ?1]
?3==[ |- forall a b c : ?1, ?2 (?2 a b) c = ?2 a (?2 b c)]
?4==[ |- ?1]
?5==[ |- forall a : ?1, ?2 a ?4 = a]
?6==[ |- forall a : ?1, ?2 ?4 a = a]
```

On lira `?i==[ |- T]` *comme : la* i^e *métavariable d'unification représente un objet de type* T*.*

On cherche donc un champ d'une classe filtrant le type T de l'objet que l'on vient de définir. Si l'on trouve un (ou des) candidats, on essayera de trouver les objets manquant dans le contexte Γ. Si l'on arrive à instancier toutes les métavariables dans la signature d'une classe, on a construit une instance.

L'algorithme a un comportement combinatoire exponentiel : On cherche *toutes* les instances possibles pour *toutes* les classes définies.

Cette première proposition purement syntaxique (filtrage du premier ordre) serait trop naïve pour être réellement utile en pratique. On va la raffiner successivement, mais il faut d'abord changer de point de vue quant à cette recherche.

3.3 Recherche d'instance comme recherche de preuve

Dans le paradigme de la théorie des types de Coq, la preuve d'un énoncé est un terme ayant pour type cet énoncé. Rechercher une instance $i : C$ revient donc à chercher une preuve de l'énoncé C dans un contexte donné. C'est évidemment un problème indécidable dans sa forme générale. On va donc choisir un sous-ensemble de résultats décidables, en fait une sous-relation de la relation de typage, que l'on exposera comme elle sous la forme d'un système déductif. Dans un contexte Γ donné, il faut voir une dérivation d'un séquent $\Gamma \vdash \mathsf{inst}\ t_1 : T_1\ \ldots\ t_n : T_n$ comme une instance de la classe $\langle x_1 : T_1 \ldots x_n : T_n \rangle$, détéctée automatiquement par notre mécanisme.

On procèdera par itération successive, en exposant les systèmes dans l'ordre de complexité croissante, et en donnant des indications quant à leur décidabilité.

3.3.1 Première proposition : recheche directe

Dans un premier temps, on a donc mis en place une reconnaissance assez naïve (celle exposée plus haut), s'apparentant plus à un simple travail combinatoire plutôt qu'à de la recherche de preuve : Il s'agit simplement de trouver n objet dans Γ ayant des types filtrant la signature d'une classe. Deux règles triviales suffisent :

$$\text{(var)}\ \frac{}{\Gamma, t : T \ \vdash\ t : T}$$

$$\text{(inst)}\ \frac{\Gamma \ \vdash\ t_1 : T_1 \qquad \cdots \qquad \Gamma \ \vdash\ t_n : T_n}{\Gamma \ \vdash\ \mathsf{inst}\ t_1 : T_1\ \ldots\ t_n : T_n}$$

L'«intelligence» mis en oeuvre ici est quasi-nulle car aucune recherche n'est faite. Considérons par exemple la classe `Reflexive` à un champ de type $\forall x, R\ x\ x$. Avec ce système, il est impossible de reconnaître les énoncés plus généraux comme `refl_equal` : $\forall A\, x,\ eq\ A\ x\ x$

3.3.2 Deuxième proposition : recherche généralisante bornée

Pour reconnaître ce type d'énoncés plus généraux, on se propose de rajouter la règle du produit dans le système.

$$\text{(prod)}\ \frac{\Gamma\ \vdash\ t : \forall x : A \cdot B \qquad \Gamma\ \vdash\ u : A}{\Gamma\ \vdash\ t\ u : B[x := u]}$$

On capture ainsi la notion d'énoncé plus général en se donnant la possibilité d'*instancier* leur paramétricité (appliquer les $\forall$). La recherche des dérivations dans ce nouveau système reste décidable car le contexte est une suite finie : quand on cherche un objet de type B il suffit de le parcourir à la recherche d'objets de type $\forall x_1 : A_1 \ldots x_n : A_n\ \cdot\ B$.

Le problème de cette recherche est qu'elle peut ne pas terminer dans certains cas : avec les objets $0 : nat$ et $S : nat \to nat$ dans le contexte, la classe $\langle \_ : nat \rangle$ engendrerait un infinité d'instances : 0, $S\ 0$, $S\ S\ 0 \ldots$ Il est donc nécessaire de fixer un nombre arbitraire n qui *bornera* la profondeur de la recherche.

Un problème de l'ordre de la complexité algorithmique est celui de l'explosion combinatoire que cela peut engendrer. Sur l'exemple précédent, rajoutons la définition $pred : nat \to nat$. On générera n instances, qui sont les n combinaisons bien typées de 0 S et $pred$. On n'ose même pas imaginer le comportement d'un tel algorithme dans un contexte «réel» comme en théorie des groupes, où ce genre de cyclicité intervient souvent...

3.3.3 Proposition finale : instances paramétrées

Pour pallier à ce problème algorithmique, réfléchissons à la nature de cette recherche : elle s'effectue au moment de la définition, pour «faciliter le travail» en quelque sorte de la recherche lors de la résolution de surcharge. En effet, on déclare à la définition de chaque nouveaux objets les instances auxquels ils peuvent appartenir. A l'utilisation d'une méthode surchargée, l'algorithme de résolution n'aura plus qu'à aller chercher dans ces instances.

On peut essayer d'utiliser cette résolution (de type PROLOG) plus profondément, et décharger du même geste le travail de la recherche d'instance. Pour cela, introduisons l'idée d'*instances paramétrées*. Elles généralisent la notion d'instance, en introduisant un ou des paramètres supplémentaires (entre $\{\}$) :

$$\Gamma\ \vdash\ \mathsf{inst}\ \{n : nat\}\ (S\ n : nat)$$

Cela signifie : pour tout objet de type nat, on sait créer une instance de la classe $[\_ : nat]$.

La règle de déduction à rajouter dans sa forme générale est :

$$\text{(parinst)}\ \frac{\Gamma\ \vdash\ t_i : \forall x : T \cdot T_i \qquad \Gamma, x : T\ \vdash\ \mathsf{inst}\ t_1 : T_1 \ldots (t_i\ x) : T_i \ldots t_n : T_n}{\Gamma\ \vdash\ \mathsf{inst}\ \{x : T\}\ t_1 : T_1 \ldots (t_i\ x) : T_i \ldots t_n : T_n}$$

On pourra la lire comme : si l'on possède pour créer une instance un objet d'un type plus général que celui attendu (t_i), on crée une instance que l'on paramètre par un argument (de type T).

Cette règle, par rapport à celle du produit, a l'avantage de ne pas créer un nombre irraisonnable d'instances. Grâce à elle, la classe $\langle \_ : nat \rangle$ ne génère que deux instances générales : $\mathsf{inst}\ 0 : nat$ et $\mathsf{inst}\ \{n : nat\}\ (S\ n) : nat$. On a ainsi «reporté» la difficulté d'un calcul (recherche d'instance) vers un autre (résolution de la surcharge). Ainsi, pour savoir que 5 est instance de

$\langle \_ : nat \rangle$, c'est à la résolution que s'effectueront les cinq étapes de raisonnement induite par ces deux règles.

La notion d'instance paramétrée, bien que décidable et valide, n'est pour l'instant pas implémentée dans Coq. Cette dernière proposition est donc pour le moment en attente, et sera – nous l'espérons – réalisée dans un futur proche.

4 Bibliothèque de types pour la recherche efficace

Pragmatiquement, une opération principale de l'algorithme de recherche d'instance est la recherche d'objets dans le contexte Γ (définitions, lemmes, inductifs, constructeurs) ayant un type donné. Cette opération, répétée à chaque itération de l'agorithme de recherche, se doit donc d'être très rapide. Dans le cadre de son implémentation, il a donc fallu trouver un moyen efficace pour chercher un motif dans le contexte.

4.1 Bibliothèques d'énoncés mathématiques

Dans le cadre plus général de la gestion de connaissances mathématiques, la recherche d'énoncés (d'objets d'un type donné) est une problématique récurrente pour la consultation de grandes bases de données. Elle intervient dans les projets de constitution de base de données unifiées de preuves formelles, comme Whelp [AGC+04] ou Alcor [CG07], mais aussi dans le cadre des bibliothèques standard de langages fonctionnels [Rit93].

La question générale est : *comment concevoir et implémenter une base de données de connaissances mathématiques de sorte que son interrogation soit rapide, et que les résultats soient pertinents ?* Comment définir cette notion de pertinence des résultats ? Quelle structure adopter pour rendre l'interrogation efficace ?

C'est certainement une très vaste question, mais on peut dès lors distinguer plusieurs sous-problèmes et énoncer un mini-cahier des charges :

- La forme des demandes doit être le plus proche possible du langage mathématique usuel dans toutes son ambiguité, par l'utilisation de notations surchargées, de coercions, de sous-structures... Ceci est non seulement une question technique de vernaculaire mathématique utilisé, mais pose aussi des questions importantes sur la nature des énoncés : imaginons que l'on cherche un énoncé sur les groupes, qui est disponible dans la base de données mais dans une forme plus générale (sur les monoïdes) ; la recherche doit pouvoir renvoyer ce résultat.
- Les détails propres à la logique sous-jacente au système (nature des lieurs, sous-typage), ainsi que les spécificités techniques du système (représentation interne des termes) doivent être le plus possible cachés à l'utilisateur, de sorte que la recherche s'effectue *modulo* ces détails.
- A plus haut niveau et pour plus de confort, certains énoncés doivent pouvoir être assimilés : on peut imaginer que l'on assimile lors de la recherche l'ordre des quantificateurs, l'association des différents produits $(\times, \wedge)$, la commutation des opérateurs commutatifs, la curryfication, même toutes les équivalences tautologiques... L'étude de ces automatisations entre dans le cadre théorique des *isomorphismes de types* et la littérature foisonne de tentatives de conception de méthodes de recherche, dans le cadre des assistants de preuve comme dans celui des langages fonctionnels.
- Pour être utilisable sur un large corpus, la complexité de la recherche ne doit pas dépendre du nombre d'éléments que contient la bibliothèque. La structure de donnée uti-

lisée dépend aussi du type de recherche à effectuer : recherche exacte, par filtrage, par unification...

Dans le cadre de ce stage, on s'est focalisé sur ce dernier point car il est essentiel pour une implémentation utilisable dans un contexte réel, où le nombre d'objets définis peut atteindre dix mille.

4.2 Réseaux de discrimination ou filtrage en lot

Ici est présentée une structure de données efficace pour stocker un ensemble de termes, utilisable dans le contexte d'une bibliothèque d'énoncés mathématique. C'est cette structure qui a été implémenté dans Coq, mais une grande attention a été porté à la généricité du code pour qu'il puisse être réutilisable dans d'autres contextes. Nous avons d'abord pensé qu'il s'agissait d'une structure originale, avant de prendre connaissance de l'existence des réseaux de discrimination.

Le concept de réseau de discrimination vient du domaine de l'intelligence artificielle, présenté par exemple dans [Cha87]. Il s'agit de trier un ensemble de connaissance selon des critères ordonnés par importance décroissante et de les stocker comme un arbre dont chaque noeud est un critère et son sous-arbre l'ensemble des connaissances répondant à ce critère.

On présentera d'abord l'interêt d'une telle structure pour notre problème, puis le concept de base, et enfin quelques détails de l'implémentation.

4.2.1 Recherche par filtrage

On cherche en général à poser une requête sous forme d'un motif et à récupérer tous les objets filtrant ce motif (recherche par filtrage). Le problème général de la recherche par filtrage est le suivant : Soit Λ une grammaire de termes. Soit $S \subset \Lambda$. Etant donné un motif p sur Λ, on cherche tous les $t \in S$ tels que $p \cong t$ (p filtre t).

On représentera ces grammaires (abstraites) en Caml par des types algébrique, mutuellement récursifs si besoin. Par exemple :

```
type term =
  | Var of int
  | Lam of term
  | App of term * term
```

Un premier algorithme naïf en $O(|S|)$ consiste à itérer sur tous les $t \in S$ un algorithme de filtrage :

```
Set.find_all (fun t -> filters p t) S
```

Ce n'est évidemment pas satisfaisant si S est grand. Un réseau de discrimination permet d'obtenir le même résultat avec une complexité plus faible (de l'ordre de la taille du motif de recherche).

Remarquons une dernière chose : Il est souvent nécessaire pour qu'il soit pertinent que le filtrage s'effectue *modulo* certaines équivalences entre les termes. En particulier, dans une théorie des définitions (comme Coq) où les termes peuvent contenir des constantes faisant référence à des objets prédéfinis, deux termes sont δ-équivalents s'il existe une façon de « dérouler » les définitions pour rendre les termes égaux. Un algorithme trivial pour identifier

les termes *modulo* δ-équivalence consiste à dérouler les termes jusqu'à leur forme δ-normale et les comparer ensuite. C'est loin d'être optimal, mais c'est la méthode qu'il a été choisi d'implémenter car elle est simple.

Une autre équivalence que l'on veut pouvoir prendre en compte est la β-équivalence. Malheureusement, le problème de l'identité de deux termes *modulo* β-équivalence est indécidable : c'est le problème de l'unification d'ordre supérieur ([Dow01]). Bien qu'il existe des méthode d'unification prenant en compte β dans certains cas, il a été choisi de ne pas s'attaquer à ce problème difficile lors de l'implémentation.

4.2.2 Les réseaux de discrimination

Pour se faire une idée de ce que sont ces réseaux, ou pourra d'abord les voir comme des généralisation de deux idées communes pour la représentation d'ensembles de données.

Indexation par le constructeur de tête Pour optimiser la recherche décrite ci-dessus, la première idée que l'on peut avoir est de séparer l'ensemble S en n ensembles, triés par le constructeur de tête de chaque terme. Sachant le constructeur de tête du motif, la recherche s'en trouve accélérée d'un facteur n. Si cela n'est pas assez efficace, on voudra réitérer ce principe pour le deuxième constructeur, et obtenir donc $n * n$ ensembles distincts.

Les réseaux de discriminations sont justement l'itération de ce principe jusqu'aux feuilles du terme : à chaque «étage» de la structure, on donne l'ensemble des termes *filtrant* le chemin parcouru depuis la racine.

Généralisation des Tries On peut aussi voir les réseaux de discrimination comme une généralisation de la structure de dictionnaire, ou *trie* (prononcer [tri]). Les tries sont une structure arborescente de recherche par préfixe de chaîne de caractères, partageant les préfixes communs à plusieurs mots[1].

On stocke à la dernière lettre de chaque mot un identifiant unique pour le mot, indiquant que l'on se situe effectivement à la fin d'un mot (et éventuellement sa définition par exemple).

Les listes (chaînes de caractères) utilisés comme structure de base dans les tries sont une grammaire de termes comme une autre ($list ::= \mathsf{nil} \mid char\ list$). Peut-on adapter ce principe à des grammaires arbitraires ? C'est l'idée derrière les réseaux de discrimination : La différence majeure est qu'une liste n'a qu'une queue, tandis qu'un terme quelconque peut avoir plusieurs sous-termes directs, en fonction du constructeur. Les préfixes deviennent donc des termes à trous (des motifs), et la recherche se fait par filtrage et plus par préfixe commun. Attention, il faut prendre quelques précautions lors de la recherche du fait de cette complexification.

Structure et propriétés Reprenons notre grammaire de lambda-termes ci-dessus et considérons les termes :

1.	$\lambda x \cdot x$	`Lam(Var 1)`
2.	$\lambda xy \cdot x$	`Lam(Lam(Var 1))`
3.	$(\lambda x \cdot x)(\lambda x \cdot x)$	`App(Lam(Var 1)Lam(Var 1))`

Le réseau de discrimination contenant ces trois termes est représenté Fig. 1

[1] `http ://en.wikipedia.org/wiki/Trie`

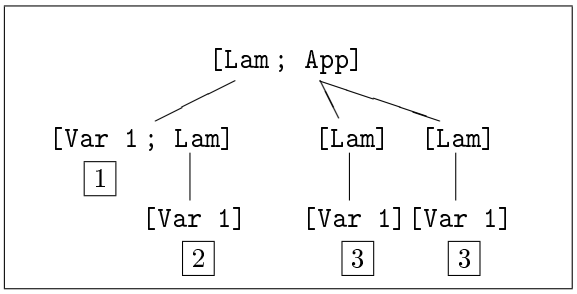

FIG. 1 – Exemple de réseau de discrimination

Il s'agit donc de conserver la représentation arborescente des termes, mais pour chaque sous-terme, proposer un *ensemble* de constructeurs possibles : l'ensemble des sous-termes qui correspondent à des termes présents. Interroger cette structure, c'est récupérer tous les termes qu'elle contient, ou (plus intéressant) récupérer tous les termes dont l'arbre syntaxique commence par une certaine série de constructeurs, *i.e* récupérer tous les termes qui filtrent un motif.

Exemple 6 *Le motif* `App(X,Y)` *appliqué au réseau de la* FIG. *1 retourne le résultat :* $\{3\}$ *(terme n°3). Le motif* `Lam(X)` *retourne* $\{1; 2\}$

Appelons «contexte» un motif ayant exactement une métavariable. La propriété intéressante de cette structure est la conservation des sous-termes : Soit R un réseau ; Soit S un sous-réseau de R, identifié par un contexte $C[]$. On a :

$$\forall t \in S,\ C[t] \in R$$

Autrement dit, si l'on cherche tous les éléments sous un contexte, on obtient les éléments qui filtrent ce contexte.

Opérations sur ces réseaux Les opérations primitives sur cette structure s'expriment sous forme ensembliste de façon élégante. On donnera ici l'expression du parcours car les autres opérations sont facilement retrouvable : ajout, suppression, parcours sous un motif,... Appelons :

- C l'ensemble des constructeurs c à un noeud donné
- $S(c)$ l'ensemble des fils s d'un constructeur donné
- $V(c)$ l'ensemble des valeurs v stockées aux noeuds terminaux

Le parcours (énumération de tous les termes présents dans un réseau) s'énonce comme suit :

$$\begin{array}{rcl} \texttt{all}(C) & = & \bigcup_{c \in C} \left(\bigcap_{s \in S(c)} \texttt{all}(s) \right) \quad \text{et} \\ \texttt{all}(C) & = & \bigcup_{v \in V(C)} V \end{array}$$

4.2.3 Implémentation fonctorielle générique

Une implémentation de cette structure de données a été réalisée dans Coq pour stocker l'ensemble des énoncés et types des objets et pouvoir y rechercher efficacement. On présentera ici l'interface et quelques choix d'implémentation.

Pour commencer, émettons quelques remarques : L'ensemble des types intéressants ici sont les types récursifs parce qu'ils capturent l'idée de grammaire de termes. Deux exemples :

```
type peano = O | S of peano
type btree = L | N of btree * btree
```

En paramétrant la définition par le type récursif, on obtient le même type, mais «découpé» :

```
type 'peano peano0 = O | S of 'peano
type 'btree btree0 = L | N of 'btree * 'btree
```

Les types de départs sont récupérables en prenant le point fixe de ces types (attention, on a besoin de l'option `-rectypes` de Caml pour cela : elle désactive l'*occur check* dans le typage et permet la définition de types réellements récursifs) ; en appliquant ces types à `unit` on obtient une structure de termes à un seul «étage» :

```
type peano = peano peano0
type peano_head = unit peano0
```

Un motif pour un type t maintenant, est un terme de type t ayant éventuellement des «trous» (la structure du terme est interrompue). On peut exprimer cela de façon générale en Caml grâce au type t_0 généré comme ci-dessus :

```
type 't t0
type pattern =
    Term of pattern t0
  | Meta
```

On «intercale» en quelque sorte un choix à chaque étage d'un terme : le sous-terme est soit un terme, soit un trou (une métavariable). De la même façon, on obtient une première approximation de la structure de réseau de discrimination en «intercalant» un choix à chaque sous-terme, c'est à dire un *ensemble*[2] de sous-termes :

```
type dn = dn list t0
```

Cependant, pour pouvoir distinguer des termes insérés dans cette structure, on doit leur associer à chacun un identifiant unique, et stocker cet identifiant aux feuilles du termes. La structure de réseau devient plus complexe :

```
type ident
type node =
  | Node of dn t0
  | Leaf of dn t0 * ident list
and dn = Nodes of node list
```

[2]Dans un souci de simplicité, on représente ici cet ensemble par une liste, mais ce choix est évidemment inefficace en pratique. Un ensemble indexé par le constructeur de tête `Map` est le choix adopté dans l'implémentation.

```
module type Datatype =
sig
  type 'a t
  val map : ('a -> 'b) -> 'a t -> 'b t
  val map2 : ('a -> 'b -> 'c) -> 'a t -> 'b t -> 'c t
  val fold : ('a -> 'b -> 'a) -> 'a -> 'b t -> 'a
  val fold2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b t -> 'c t -> 'a
  val compare : unit t -> unit t -> int
  val terminal : 'a t -> bool
end

module Make :
  functor (T:Datatype) ->
  functor (Ident:Set.OrderedType) ->
  functor (Meta:Set.OrderedType) ->
sig
  type t
  type ident = Ident.t
  type meta = Meta.t
  type 'a structure = 'a T.t
  module Idset : Set.S with type elt=ident
  type pattern =
    | Term of pattern structure
    | Meta of meta

  val empty : t
  val add : (t -> 'a -> t) -> t -> 'a structure -> ident -> t
  val find_all : t -> Idset.t
  val find_match : t -> pattern -> Idset.t
  val fold_pattern : (Idset.t -> Idset.t -> Idset.t) -> Idset.t ->
    (meta -> t -> Idset.t) -> t -> pattern -> Idset.t
  val inter : t -> t -> t
  val union : t -> t -> t
end
```

FIG. 2 – Foncteur de réseau de discrimination

On présente maintenant l'interface développée (FIG. 2). Il s'agit d'un foncteur prenant un type de donnée «paramétrisé» comme au dessus accompagné de ses fonctions de parcours, et renvoyant le type des réseaux de discrimination et les opérations d'ajout, recherche... appropriées. On y ajoute que : les identifieurs en question doivent être ordonnés, et l'on stocke un indice dans chaque métavariable de façon à pouvoir poser des contraintes d'égalité sur ces métavariables ; on a donc besoin aussi d'un type ordonné de métavariables.

Décrivons le comportement des fonctions fournies :

`add f dn t i` ajoute le couple (t, i) au réseau `dn`. Cette fonction est non-récursive : la fonction `f` est utilisée pour ajouter le sous-terme

`find_add dn` renvoie tous les identifiants des termes contenus dans le réseau `dn`

`find_match dn p` renvoie tous les identifiants des termes qui filtrent le motif `p` dans `dn`.

`fold_pattern merge n f dn p` itère la fonction `f` sur toutes les métavariables contenues dans

`p`, et tous les sous-réseaux correspondants dans `dn`. Elle utilise la fonction `merge` et l'élément de départ `n` pour agréger les résultats (typiquement $\cup$ ou $\cap$).

En appliquant ce foncteur à la représentation classique des termes en Coq (le type `constr`), on obtient une structure prête à l'emploi pour mettre en oeuvre une bibliothèque stockant l'intégralité du contexte.

Pour finir, montrons comme on peut grâce à cette interface exprimer des problèmes de recherche complexes dans un réseau, comme par exemple dans cet extrait (simplifié) du code, qui cherche tous les termes filtrant un motif, possiblement sous un ou plusieurs produits dépendants (`forall`).

```
let search_pattern_concl pat dn =
  let prod_pat = PATTERN [ forall _:_, ?1] in
  let rec aux dn =
    Idset.union
      (find_match t pat)
      (fold_pattern union Idset.empty aux dn prod_pat) in
  aux dn
```

Pour simplifier le code, on suppose ici la présence d'un préprocesseur qui remplace `PATTERN [...]` par un terme de type `pattern` ayant pour métavariable `?1`

4.2.4 Problèmes pratiques soulevés

Présentons brièvement quelques problèmes soulevés par l'implémentation, qui l'ont rendu plus complexe que celle présentée ci-dessus.

Décalage d'indices de de Bruijn La représentation des termes Coq se fait selon le principe du *locally nameless* : Les variables locales ainsi que les variable faisant référence à des lieurs dans les termes sont représentés par des indices de de Bruijn. Le reste (constantes,...) est nommé.

Il est important de prendre ces indices en considération dans notre représentation : les indices présents dans un sous-réseau sous un lieur doivent être décalés lors de la comparaison. Par exemple, considérons le réseau composé du seul terme :

$$\lambda T : \star \cdot \lambda x : T \cdot T$$

Sa représentation de de Bruijn est :

$$\lambda \star \cdot \lambda 1 \cdot 2$$

La recherche du motif $\lambda \_ : \mathbf{X} \cdot \mathbf{X}$ échouerait car les deux métavariables $\mathbf{X}$ ne seront pas égale syntactiquement : il faut prendre en compte que cette comparaison entre 1 et 2 se fait avec un décalage d'indice de 1 puisqu'on passe sous un λ.

Problèmes relatifs à la représentation des termes D'autres problèmes liés à cette représentation se sont posés. L'arité de l'application en est un : pour des raisons d'efficacité de la réduction, l'application est représenté de façon n-aire par une liste. Cela empêche le filtrage de *eq nat* 2 2 avec $\mathbf{X}$ 2 2 par exemple car l'application est dans un cas d'arité 3, dans l'autre d'arité 2. Pour remédier à cela, il faut exploser toutes les applications en applications binaires.

Un autre écueuil intervient dans le traitement des sortes : la comparaison de deux sortes doit faire intervenir les contraintes de sous-typage, ce qui rend cette comparaison non symétrique. Et enfin au niveau du nommage optionnel des variables dans les lieurs : il faut les ignorer lors de la comparaison.

4.3 Réimplémentation des procédures de recherche

Pour rechercher des instances automatiquement, il fallait donc pouvoir chercher efficacement dans le contexte des termes dont le type filtrait un motif donné. C'est pour cela que l'on a conçu cette base de données d'objets du contexte indexés par leur type.

Il existait une autre fonctionnalité qui pouvait bénéficier facilement de cette bibliothèque : les fonctions de recherches (naturellement). Il en existe à l'heure actuelle quatre dans Coq :

- `Search id` recherche dans tous les objets chargés ceux de la forme $\forall x_1 \dots x_n$, `id` $t_1 \dots t_m$, où `id` est un identificateur (une constante, un inductif...) ;
- `SearchAbout id` cherche tous les objets dont le type contient l'identificateur `id` ;
- `SearchPattern p` cherche tous les objets dont la conclusion correspond au motif `p` ;
- `SearchRewrite p` cherche tous les objets dont la conclusion est de la forme $p =$ `_` ou `_` $= p$ (pour être utilisé avec la tactique `rewrite`).

L'implémentation de ces commandes était incomplète et lente : on itérait sur chaque objet du contexte une opération de filtrage, et les motifs ne pouvaient être que des applications de constantes (pas de $\forall$ par exemple).

Nous avons réimplémentés ces procédures de recherche grâce à la librairie de types développée. Elles s'en trouvent accélérées grandement, et certains problèmes se trouvent résolues. Voici les différences majeures entre les deux implémentations :

- Toutes les expressions sont acceptées par la recherche ;
  Exemple 7 *Le motif* `SearchPattern (forall x, _ x x).` *était rejetté car il comprenant un* $\forall$. *Il est maintenant accepté.*
- Les motifs sont typés, ce qui n'était pas le cas : certains trous dans les motifs peuvent être résolus par le typage.
  Exemple 8

4.4 Pour aller plus loin : isomorphismes de types

Pour conclure, nous exposerons une direction possible dans laquelle ce travail pourrait continuer. Il s'agit en fait d'un champ de recherche dont les applications vont bien au delà de la recherche automatique ou de l'interrogation de bases de données.

Une des limitations très forte de l'algorithme de recherche d'instance ainsi que de la base de données mise au point est sa rigidité syntaxique : pour retourner un résultat, la requête doit très précisément filtrer un terme dans la base, exactement de la façon dont il a été entré. Or, comme nous l'avions énoncé dans notre cahier des charges au début de cette partie, certains détails sous-jacents à la logique gagneraient à être négligés par la recherche :

Exemple 9 *les termes* $\forall xy, P(x, y)$ *et* $\forall xy, P(y, x)$ *ont la même "signification", mais diffèrent par l'ordre des arguments et ne peuvent pas être assimilés par le système. De même pour les deux expressions de la transitivité :* $\forall xyz, P(x, y) \to P(y, z) \to P(x, z)$ *et* $\forall xyz, P(y, z) \to P(x, y) \to P(x, z)$

Exemple 10 *les termes $A \times B \to C$ et $A \to B \to C$ diffèrent par la curryfication, mais sont équivalents (même isomorphes). Ils devraient aussi pouvoir être assimilés.*

4.4.1 Présentation théorique

L'étude des isomorphismes de types est l'étude, dans une théorie des types donnée, des types non seulement équivalents, mais dont l'équivalence se fait «sans perte d'information logique». Formellement,

Définition 1 (isomorphisme) *A et B sont isomorphes s'il existe $f : A \to B$ et $g : B \to A$ tels que $f \circ g = id_A$ et $g \circ f = id_B$*

Un résultat majeur pour le domaine est dû à Soloviev ([Sol93]), qui a mis en évidence un système d'isomorphismes complet pour la théorie modélisée par les catégories cartésiennes fermées (le λ-calcul simplement typé avec produit et type unité). On peut donc prouver par un raisonnement équationnel que deux types sont isomorphes.

De plus, il a été prouvé qu'une sous-partie intéressante de cette théorie équationnelle est décidable (elle forme un système de réecriture normalisant). Il est donc (presque) possible de décider si deux types sont isomorphes, et donc de réaliser une bibliothèque où la recherche s'effectue *modulo isomorphismes*. On citera la tentative fructueuse de Rittri [Rit93] pour la bibliothèque standard des langages ML.

Il est à noter qu'il peut être important d'exhiber les fonctions de conversion f et g, car elles permettent de transformer les programmes (ou les preuves) en conséquence. Ceci est souvent inutile lorsqu'il s'agit simplement de proposer plus de résultats à l'utilisateur, mais peut être crucial dans le cadre d'automatisations des assistants de preuve.

Cette théorie a été adaptée aux λ-calcul dépendants par Delahaye ([DDCW97]), qui a implémenté une recherche pour Coq modulo isomorphismes. Malheureusement, son code ne construit pas les termes de conversion, et n'est donc pas exploitable pour l'automatisation et la recherche de preuve. De plus, la théorie sous-jacente s'éloigne de celle de Coq : elle considère un produit cartésien atomique, et considère tous les types inductifs comme des constantes, ce qui réduit de beaucoup le nombre de types assimilables.

4.4.2 Perspectives

La recherche d'instance bénéficierait sans doutes beaucoup de plus de souplesse pour sa partie "reconnaissance de motifs", en ne choisissant pas les énoncés par pur filtrage syntaxique, mais en assimilant plus de termes «trivialement équivalents», concept que la notion d'isomorphisme semble capturer.

Mais ce n'est sans doutes pas le seul outil de Coq qui pourrait en bénéficier : les tactiques de recherche de preuve comme `auto` gagneraient certainement en puissance à pouvoir raisonner *modulo* certains isomorphismes.

Un plan de travail pour cette réalisation future pourrait être :

- Conception d'une théorie axiomatique plus adaptée au calcul des constructions inductives, prenant en compte plus de schémas d'inductifs ;
- Mise en évidence d'un sous-ensemble décidable d'équations ;
- Mise en oeuvre d'un algorithme d'unification modulo isomorphismes générant les termes de conversion

– Intégration dans Coq, à un niveau assez bas pour être pris en compte par les tactiques `apply`, `auto`... mais aussi pour la recherche (`SearchPattern`) et la reconnaissance d'instances.

Références

[AGC+04] A. Asperti, F. Guidi, C.S. Coen, E. Tassi, and S. Zacchiroli. A content based mathematical search engine : Whelp. *Post-proceedings of the Types 2004 International Conference*, 3839 :17–32, 2004.

[BBC+08] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.C. Filliâtre, E. Giménez, H. Herbelin, et al. The Coq Proof Assistant Reference Manual Version 8.2. *INRIA*, 2008.

[CG07] P. Cairns and J. Gow. Integrating Searching and Authoring in Mizar. *Journal of Automated Reasoning*, 39(2) :141–160, 2007.

[Cha87] E. Charniak. *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, 1987.

[CP90] T. Coquand and C. Paulin. Inductively defined types. *Proceedings of the international conference on Computer logic table of contents*, pages 50–66, 1990.

[DDCW97] D. Delahaye, R. Di Cosmo, and B. Werner. Recherche dans une bibliotheque de preuves Coq en utilisant filetype et modulo isomorphismes. *PRC/GDR de programmation, Pole Preuves et Specifications Algebriques*, 1997.

[Dow01] G. Dowek. Higher-order unification and matching. *Handbook of automated reasoning*, pages 1009–1062, 2001.

[HHJW96] C.V. Hall, K. Hammond, S.L.P. Jones, and P.L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2) :109–138, 1996.

[Rit93] M. Rittri. Retrieving Library Functions by Unifying Types Modulo Linear Isomorphism. *ITA*, 27(6) :523–540, 1993.

[SO08] Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In *TPHOLS'08*, volume 5170 of *Lecture Notes in Computer Science*. Springer, August 2008. To appear.

[Sol93] S.V. Soloviev. A complete axiom system for isomorphisms of types in closed categories. *Lecture Notes in Artificial Intelligence*, 698 :380–392, 1993.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, 1989.