Modules, Objets et Calcul Formel

Sylvain Boulmé, Thérèse Hardin & Renaud Rioboo

Laboratoire d'Informatique de Paris 6 - Case 168,
4 place Jussieu 75252 Paris CEDEX 05, France
{Sylvain.Boulme, Therese.Hardin, Renaud.Rioboo}@lip6.fr

Résumé

Le langage OCAML propose des classes et des modules très élaborés. Ces deux paradigmes apportent tous deux des facilités de décomposition, d'abstraction, etc. relativement proches. Cette richesse peut laisser le programmeur perplexe, lorsqu'il s'agit de choisir la manière d'implanter une spécification un peu complexe. Nous avons entrepris le développement (en OCAML) d'un environnement de programmation certifiée pour le calcul formel. Nous présentons dans cet article les étapes d'élaboration de notre propre méthodologie, qui marie objets et modules dans un style peu conventionnel. Nous expliquons en quoi la solution retenue nous permet de rendre compte au mieux de tous les aspects de notre spécification. Nous espérons ainsi apporter quelques éléments au débat modules/objets et montrer l'intérêt de certaines extensions des langages à base de modules ou de classes.

1. Introduction

L'utilisateur du langage OCAML dispose à la fois d'un système de modules très évolué et de traits objet très puissants et surtout bien compris. Cela facilite son activité de programmation car il peut exprimer toute la complexité de son implantation avec une grande palette de moyens. Cela peut le plonger parfois dans la perplexité car modules et classes offrent des formes d'abstraction très voisines et il n'est pas toujours évident de choisir entre l'une et l'autre [10].

Au cours des développements effectués dans le cadre du projet FOC, décrit ci-après, nous avons été confrontés à ce choix de style. Cet article présente notre cahier des charges, les différentes tentatives d'implantation effectuées et les raisons de leur abandon, enfin la justification de notre choix définitif d'implantation.

Le projet FOC a pour objectif de créer un environnement de programmation certifiée pour le calcul formel. Il ne s'agit pas de valider une librairie dédiée, mais d'offrir un environnement ouvert qui permette un développement modulaire du code et des preuves de ses propriétés. Cet outil doit aussi fournir des facilités de maintenance, de réutilisation. En effet, des algorithmes fondamentaux comme le calcul du pgcd de polynômes sont constamment améliorés en vue d'une meilleure efficacité. Il faut donc pouvoir suivre leur évolution.

Ces questions ont conduit la communauté du calcul formel à développer des langages de programmation originaux et très puissants, comme AXIOM (voir [3]). Celui-ci introduit les concepts de *catégories* et *domaines*, rappelant les classes de la programmation objet. Mais, même avec un langage de programmation bien adapté, les algorithmes considérés restent difficiles à programmer. Vu l'ampleur et la complexité des calculs, il est difficile de les tester. On est donc conduit à certifier les programmes.

Une première expérience [1] de couplage entre AXIOM et l'assistant à la démonstration COQ, nous a convaincus que la programmation de code à certifier devait se faire dans un langage dont la sémantique soit complètement étudiée. Ce projet utilise donc OCAML et COQ. En effet, créer notre propre langage nous aurait confrontés à des choix de conception analogues à ceux posés pour OCAML et à refaire tout un travail d'implantation et de sémantique déjà largement effectué.

Profitant de l'expérience acquise dans la programmation en AXIOM, nous avons mené une étude à la fois théorique et pratique sur la méthodologie de développement à adopter, qui est le sujet de cet article. On décrit d'abord le cahier des charges et la conception du projet, le codage avec uniquement des modules est présenté dans la section 3, où sont aussi discutés les liens entre héritage et modularité. Les premières solutions à base d'objets sont décrites dans la section 4. La section 5 expose la solution retenue.

2. Description du projet

Le projet FOC souhaiterait ultimement offrir aux manipulateurs de calcul formel un environnement complet de développement permettant d'exprimer spécifications, codes et preuves de propriétés à l'aide de syntaxes concrètes bien choisies. Il devrait fournir aussi une panoplie d'outils facilitant leur tâche (aide à la preuve, outils de mise au point...) construits essentiellemt à l'aide de OCAML et COQ.

2.1. Le cahier des charges

La première phase de développement est la mise en place du cœur du système : la librairie des structures algébriques. Le cahier des charges de cette librairie a servi de fil conducteur à la réalisation des différents prototypes présentés ici. Il demande de répondre à plusieurs impératifs :

1. L'architecture de la librairie doit refléter celle des structures algébriques, par exemple les groupes doivent être construits à partir des monoides.
2. Pour une même notion, plusieurs degrés d'abstraction sont proposées si nécessaire. Ainsi, on doit pouvoir utiliser les propriétés de l'opération d'un groupe (son type par exemple) sans avoir à considérer ses implantations.
3. Certaines notions doivent pouvoir être définies “par défaut” pour toute une famille de structures et redéfinies, si besoin est, pour une structure particulière. Ainsi, la définition de la fonction `est_different_de` comme négation de la fonction `est_egal_a` doit pouvoir être partagée par toutes les structures construites sur les ensembles. Mais celles-ci peuvent la redéfinir.
4. Cette exigence va de pair avec la possibilité de préciser pas à pas une structure algébrique pour passer par exemple d'une implantation de $\mathbb{Z}/2\mathbb{Z}$ qui utilise des entiers à une autre qui utilise des booléens, tout en partageant un certain nombre d'outils entre les deux implantations.
5. La librairie doit bien sûr contenir des implantations pour les structures de données que l'on utilise habituellement en calcul formel. On y trouvera donc une représentation des grands entiers, des entiers modulaires ainsi que des représentations des polynômes (au moins celles dites distribuées et récursives).

Dans la mise en œuvre de ces cinq points, on essaiera de faciliter au maximum la correspondance entre la description OCAML des structures algébriques et leur spécification en COQ afin d'aider la réalisation des preuves de correction. De plus, on cherchera à rendre le typage le plus fort possible tout en gérant au mieux les points précédents.

2.2. Conception

Les exigences énoncées ci-dessus ne sont pas spécifiques au calcul formel et correspondent à des traits bien étudiés en programmation : typage fort, polymorphisme, généricité, définitions par défaut, héritage, partage et réutilisation, etc. Ces traits ne peuvent être commodément décrits avec seulement les types classiques (union et enregistrement) des langages de programmation. Ils motivent d'ailleurs les travaux sur les modules et les objets.

Même si OCAML offre modules et objets pour implanter ces différents traits, la conception de la librairie n'est pas si aisée car certaines exigences sont contradictoires. Par exemple, il n'est pas très difficile de définir par un type différent (trait modulaire) $\mathbb{Z}/2\mathbb{Z}$ et $\mathbb{Z}$ mais il est alors plus compliqué de faire partager (trait objet) à ces deux implantations des outils appropriés pour les entiers ou pour les entiers modulaires. Il faut donc déterminer précisément quelle combinaison de traits objet et modulaires nous permet de définir au mieux l'architecture de la librairie. Certes, comme souligné dans [10], les formalismes à base de modules ou d'objets avec classes permettent d'utiliser les mêmes styles de programmation. On peut adopter un style "modulaire", appelé ici style ADT pour Type Abstrait de Données, en programmant à l'aide de classes OCAML. On peut aussi adopter un style, dit "par encapsulation des données dans les objets", avec une utilisation plus traditionnelle des objets. Ces deux styles seront précisés et commentés dans la suite.

Afin de mieux cerner les difficultés et les solutions possibles, nous avons réalisé différentes déclinaisons d'une librairie minimale. Celle-ci contient les outils nécessaires à la programmation des algorithmes de pgcd de polynômes sur des anneaux factoriels (i.e. qui possèdent un pgcd). Nous avons étudié ces réalisations selon trois points de vue : possibilité d'expression de la spécification mathématique, facilité du développement à la fois pour le concepteur et l'utilisateur, efficacité en temps de calcul.

2.3. Vocabulaire du projet

Tout au long de cette étude, nous avons été aussi amenés à dégager notre propre vocabulaire pour distinguer les points de vue abstrait (proche des mathématiques) et concret (proche de la programmation). Nous le décrivons brièvement, car il sera utilisé dans la suite.

Les "choses" manipulées en calcul formel sont des *individus* abstraits, comme $X + 2$, ou des *entités* concrètes, comme [(1,1);(2,0)] qui peut représenter ce polynôme. Les *sociétés* expriment un lien entre des individus, décrit par des *fonctionnalités*, par exemple, celui donné par une loi de composition interne vérifiant certaines propriétés. Les entités sont, elles, regroupées en *collections*, qui expriment un lien (entre les manipulations sur ses entités) donnée par des *opérations*.

Les *prototypes* décrivent des fonctionnalités qui permettent de grouper des sociétés en un *genre* spécifiant une structure mathématique. De même les *signatures*, qui décrivent les opérations, servent à regrouper les collections en *espèces*. Les propriétés d'une collection sont de nature sémantique, elles dépendent souvent de l'espèce et de la représentation choisie pour la collection. On doit disposer par exemple de différentes collections des polynômes (représentation distribuée, récursive, etc.).

3. La déclinaison avec les modules

Il semble naturel de coder les *espèces* par des signatures de modules de OCAML et les *collections* par des implantations de modules. En effet, les *sociétés* (structures mathématiques) étant connues, les collections qui les implantent peuvent être codées statiquement. Les calculs ne portent en général que sur les entités et non sur les collections elles-mêmes.

3.1. Abstraction et typage

Le mécanisme types abstraits/types manifestes des modules permet de protéger la représentation des entités. Cela amène l'utilisateur à programmer de façon plus modulaire, tout en évitant confusions et erreurs.

Par exemple, l'espèce des anneaux est décrite par la signature suivante :

```
module type Anneau =
sig
  type t
  val egal: (t*t) -> bool
  val plus: (t*t) -> t
  val mult: (t*t) -> t
  val opp: t -> t
  val un: t
  val print: t -> string
  ...
end
```

L'anneau $\mathbb{Z}/2\mathbb{Z}$ est défini par :

```
module Z2z : Anneau =
struct
  type t=bool
  let egal (x,y) = (x=y)
  let plus (x,y) =
    (x || y) && not (x && y)
  let mult (x,y) = x && y
  let opp x = x
  let un=true
  let print=function
      false -> "0"
    | true -> "1"
  ...
end
```

Suit un calcul dans `Z2z`:

```
# let zero=Z2z.plus (Z2z.un, Z2z.opp Z2z.un);;
val zero : Z2z.t = <abstr>
# Z2z.print zero;;
- : string = "0"
```

Le type des entités de `Z2z` (`zero` par exemple) est abstrait, c'est-à-dire que le codage de $\mathbb{Z}/2\mathbb{Z}$ en tant qu'anneau de booléens est ici masqué, comme le montre l'essai suivant :

```
# Z2z.plus (Z2z.un,true);;
This expression has type Z2z.t * bool but is here used with type
  Z2z.t * Z2z.t
```

Ce mécanisme d'abstraction de type est encore renforcé avec les foncteurs, qui permettent de définir des collections paramétrées, avec exactement l'abstraction de type souhaitée. Expliquons cela sur un exemple.

L'espèce des polynômes en une variable peut être décrite par :

```
module type PolyFormel =
sig
  module Base: Anneau
  type t
  val egal: (t*t) -> bool
  val mult_extern:(Base.t*t)->t
  ...
end
```

La collection paramétrée des polynômes creux est définie par :

```
module PolyCreux (A:Anneau) :
  (PolyFormel with module Base=A) =
struct
  module Base=A
  type t=(A.t*int) list
  let egal = ...
  ...
end
```

La collection des polynômes creux à cœfficients dans $\mathbb{Z}/2\mathbb{Z}$ s'écrit :

```
module PolZ2z = PolyCreux (Z2z)
```

```
# let idPolZ2z y = PolZ2z.mult_extern (Z2z.un,y);;
val idPolZ2z : PolZ2z.t -> PolZ2z.t = <fun>
# idPolZ2z [(Z2z.un,1)];;
This expression has type (Z2z.t * int) list but is here used
  with type   PolZ2z.t = PolyCreux(Z2z).t
```

Le type `PolZ2z.t` est incompatible avec `(Z2z.t*int) list`. La représentation des polynômes creux peut donc être modifiée de façon transparente pour ses importateurs. En revanche, `PolZ2z.Base.t` est bien égal à `Z2z.t`, ce qui contrôle la correction de la spécifiaction. Ainsi, le typage offert par les modules permet de décrire finement la spécification et de contrôler la cohérence de la description.

3.2. Héritage

Beaucoup d'algorithmes de calcul formel s'appliquent à toute une famille de structures. On souhaite cependant pouvoir spécialiser certains de ces algorithmes pour une structure particulière, ayant plus de propriétés. Montrons comment ce mécanisme d'héritage peut être décrit avec les modules.

3.2.1. Enrichissement dans les modules

Pour montrer les différentes possibilités de simulation de l'héritage dans les modules, prenons l'exemple des anneaux intègres (caractérisés par l'existence d'une division exacte). On peut simplement poser :

```
module AnneauIntegre =
  sig
```

```
  module A: Anneau
  val exquo: A.t*A.t->A.t
end
```

Les modules de signature `AnneauIntegre` ont un champ `A` de signature `Anneau`. Dans une vraie librairie, comme un anneau est un groupe, ce champ `A` a un champ `G` de signature `Group`, qui a un champ `M` de signature `Monoid`, etc. Donc si on veut accéder à certaines fonctionnalités d'un anneau intègre provenant du monoïde sous-jacent, il faudra préfixer leur nom par "`A.G.M....`". Cela demande la connaissance explicite de toute l'architecture de la librairie (au moins, dans la version minimale, une cinquantaine de modules et une douzaine de couches d'importation). Cette solution n'est donc pas satisfaisante.

Une autre solution consiste à recopier la signature d'`Anneau` à l'intérieur de la signature d'`AnneauIntegre`. Ainsi la conversion d'`AnneauIntegre` en `Anneau` (ou en `Groupe`) a lieu implicitement, grâce au sous-typage induit par les signatures de module:

```
module AnneauIntegre =
  sig
    type t                    val egal: t*t -> bool
    val plus: t*t -> t        val mult: t*t -> t
    val exquo: t*t -> t       ...
  end
```

Un prototype réduit fondé sur cette idée a été réalisé (25 modules, 8 couches). La conclusion a été sans appel : cette solution n'est pas viable, dans la mesure où toute modification d'un module doit être reportée à la main dans tous les modules qui le déplient. Pour être réalisable, il faudrait automatiser cette méthode de "copier-coller" par exemple à l'aide d'un préprocesseur. Si cela semble simple pour les signatures de module, cela parait plus compliqué pour les structures de module et les foncteurs. En effet, cela nécessite une analyse sémantique du code.

3.2.2. Liaison tardive et définitions par défaut

Il est crucial de pouvoir redéfinir au cours de l'héritage certaines fonctionnalités (pour les optimiser par exemple). La liaison tardive permet de ne redéfinir que le strict nécessaire. Prenons l'exemple de `mult_extern` qui multiplie un élément a de l'anneau A par le polynôme p de $A[X]$. Sur une représentation creuse des polynômes, `mult_extern` doit tester lors de l'exécution que tout cœfficient de $a \times p$ est non nul. Si A est intègre, ce test est inutile dès lors que a est non nul. On peut donc écrire une version optimisée de `mult_extern` dans la collection paramétrée `PolyCreuxIntegre` qui hérite de `PolyCreux`.

Une espèce est caractérisée par des opérations *primitives* et peut posséder des opérations par *défaut* construites à partir des opérations primitives. Une collection implantant une espèce doit fournir une implantation des opérations primitives. Le pgcd est une opération primitive des anneaux factoriels. Il est une opération par défaut des anneaux euclidiens, car il peut être dérivé de l'opération primitive de division euclidienne. On souhaite qu'une collection implantant un anneau euclidien puisse être reconnue comme une collection implantant un anneau factoriel sans avoir à l'expliciter. On recherche donc une cœrcion implicite d'une espèce vers une autre qui sache utiliser indifféremment toutes les opérations de l'espèce cœrcée. Dans les modules, une espèce est implantée par une première signature (de module) ne contenant que ses opérations primitives. Ainsi, définir les collections de l'espèce requiert seulement l'implantation des opérations primitives. Il faut aussi une seconde signature comportant toutes les opérations de l'espèce afin de permettre toutes les cœrcions envisageables. Il faut ensuite un foncteur de passage entre ces deux signatures. Tout doit être explicité et tout ajout d'opération par défaut doit être reporté. La gestion devient d'une lourdeur fastidieuse.

4. Programmation avec les objets

Dans notre cadre, le mécanime d'héritage semble donc indispensable. C'est pour cela que nous nous sommes tournés vers la programmation avec les classes de OCAML.

4.1. Programmation par encapsulation

L'idée la plus courante consiste à coder les espèces par des classes virtuelles, les collections par des classes concrètes et les entités par des objets, en fait par des variables d'instance des objets. Cette représentation peut néanmoins s'éloigner plus ou moins du cahier des charges, comme le montrent les différents modèles décrits ci-après.

4.1.1. Version naïve : le modèle M1

Dans ce modèle, l'espèce `anneau` peut se décrire par :

```
class virtual anneau =
object (_:'a)
  method virtual egal:'a -> bool  method virtual plus:'a -> 'a
  method virtual mult: 'a ->'a    method virtual opp: 'a
  method virtual un: 'a           method virtual print: string
end
```

Notons que `un` a le même type que `opp` alors que leurs sémantiques sont tout à fait différentes. En effet, `opp` exprime l'opérateur *"opposé"* et dépend de l'entité implantée, alors que `un` représente l'entité *"élément neutre de la multiplication"* et ne dépend que de la collection. Cela tient au fait que le type des fonctionnalités ne correspond pas à leur arité. Elles sont en effet déjà appliquées à l'entité sous-jacente. Pour que `un` apparaisse dans la spécification `anneau`, on a ajouté implicitement et artificiellement un argument à cette fonctionnalité 0-aire. De même les opérations binaires deviennent des méthodes à un argument, ce qui est peu conforme au cahier des charges.

Souvent le codage de ces opérations binaires demande d'accéder à la représentation interne de l'argument. Celle-ci doit être rendue accessible par une méthode, appelée ici `rep`. Par exemple, la collection des entiers s'écrit :

```
class entiers =
object
  inherit anneau
  val ma_rep = 0
  method rep = ma_rep
  method plus x = < ma_rep = ma_rep + x#rep >
  method un = < ma_rep = 1>
  method print = string_of_int ma_rep
  ...
end
```

On peut l'utiliser de la manière suivante (ici dans le toplevel) :

```
# let (++) x y = x#plus y;;
val ++ : < plus : 'a -> 'b; .. > -> 'a -> 'b = <fun>
# let un = (new entiers)#un;;
val un : entiers = <obj>
# let trois = (un ++ un ++ un);;
val trois : entiers = <obj>
```

Notons que l'opérateur infixe `++` se comporte comme un opérateur "surchargé". Le polymorphisme de OCAML permet de l'appliquer à tous les objets ayant une méthode nommée `plus`. En revanche, on n'a aucun contrôle sur le bien fondé de l'utilisation de cette méthode.

Voici un problème plus grave. Une représentation d'entités est souvent construite en maintenant un invariant implicite, qui doit être préservé par les méthodes de la classe. Par exemple, les entités de $\mathbb{Z}/2\mathbb{Z}$ peuvent être codées par `0` ou `1` :

```
class z2z =
object (moi)
  inherit anneau
  val ma_rep = 0
  method rep = ma_rep
  method plus x = let tmp=ma_rep+x#rep in
                  if tmp=2 then < ma_rep = 0 >
                           else < ma_rep = tmp >
 method print = (string_of_int ma_rep)^"[2]"
```

```
 ....
end;;
```

On peut alors effectuer les calculs suivants :

```
# let unM2 = (new z2z)#un ;;
val unM2 : z2z = <obj>
#let troisM2 = unM2 ++ unM2 ++ unM2 ;;
val troisM2 : z2z = <obj>
# (unM2 ++ trois)#print;;
- : string = "4[2]"
```

L'invariant implicite a donc été contourné car `unM2 ++ trois` a bien un type :

```
< egal:'a->bool; mult:'a->'a; plus:'a->'a; print:string;
      rep:int; un:'a; zero:'a > as 'a
```

En effet, `z2z` et `entiers` ne sont que des abréviations du type objet ci-dessus.

Pour remédier à ce problème en restant conforme au cahier des charges, on ne peut qu'agir sur la méthode "`rep`", d'où deux solutions. La première M2 utilise une convention sur le nom des méthodes "`rep`". La deuxième M3 se sert du mécanisme d'abstraction de type des modules (voir 3.1) pour abstraire le type des méthodes "`rep`". Nous allons considérer ces deux techniques.

4.1.2. Version M2 avec convention sur les noms de méthode

On peut imposer que le nom de méthode "`rep`" diffère pour chacune des collections : `rep_`A pour la collection A. On aura ainsi `rep_entiers` et `rep_z2z`, ce qui rend incompatibles les types `entiers` et `z2z` (et on suppose dans l'exemple suivant que le changement est fait et que `un` et `unM2` ont respectivement les types `entiers` et `z2z`). Cette convention, fort lourde, peut être utilisée pour les classes paramétrées :

```
class ['a] polycreux =
object (_:'b)
  constraint 'a=#anneau
  val ma_rep : ('a*int) list = []
  method rep_polycreux = ma_rep
  method mult_extern (x:'a) :'b = ...
  ...
end;;
# let p1 = new polycreux;;
val p1 : _#anneau polycreux = <obj>
# p1#mult_extern un;;
- : entiers polycreux = <obj>
# p1;;
- : entiers polycreux = <obj>
# p1#mult_extern unM2;;
This expression has type
```

```
  < egal : 'a -> bool;...; rep_z2z : int; un : 'a > as 'a
but is here used with type
  entiers =
  < egal : entiers -> bool;...; rep_entiers : int;  un : entiers >
Only the first object type has a method rep_z2z
```

Comme l'illustre cet exemple, un inconvénient majeur de ce style est le suivant. Un "polynôme" créé avec `new` ne connaît son type que lors de la première opération, comme `mult_extern`, faisant intervenir ses cœfficients. La variable libre `_#anneau` est alors définitivement unifiée avec `entiers`. D'où l'erreur de typage. Pour créer directement un polynôme de type souhaité, on peut certes appliquer `mult_extern` dès le `new`. Mais cela est peu pratique et assez dangereux.

4.1.3. Version M3 avec utilisation du mécanisme d'abstraction de type des modules

On peut utiliser le mécanisme d'abstraction de type offert par les modules pour masquer le type de `rep` (ou d'autres méthodes), au moment du `new`, afin de renforcer le typage lorsque l'on calcule sur les entités.

La classe peut être encapsulée dans un module et le type de `rep` peut être masqué dans la partie de la signature du module décrivant la classe. Mais on a alors trop peu d'informations sur `rep` pour définir valablement (cf. 4.1.1) de nouvelles méthodes au cours de l'héritage.

Pour remédier à cela, on peut seulement encapsuler un objet privilégié de la classe, qui servira de germe à la collection et permettra d'abstraire le type `rep`. Par exemple à partir de la classe `anneau` précédente, on obtient :

```
class virtual ['b] anneau_r =
object (_:'a)
  inherit anneau
  method virtual rep: 'b
end;;
```

```
module type Anneau =
sig
   type rep
   val zero: rep anneau_r
end;;
```

On convient de choisir toujours le zéro de l'anneau comme objet privilégié. Dans l'exemple, la collection des entiers est construite par le module `Entiers` avec une seule instanciation de la classe `entiers` :

```
module Entiers:Anneau=
struct
  type rep=int
  let zero=new entiers
end;;
```

```
# let un = Entiers.zero#un;;
val un : Entiers.rep anneau_r
# let trois = (un ++ un ++ un);;
val trois : Entiers.rep anneau_r
```

Ce mécanisme s'étend bien aux foncteurs. Mais avec ce modèle, la notion de collection devient floue : la société des entiers correspond-elle à `entiers`, à `Entiers`, ou à `Entiers.zero` ?

4.2. Vers une programmation ADT

La solution "objet" précédente ne nous convient pas, pour essentiellement deux raisons.

Tout d'abord, ce style de programmation n'est pas très efficace. L'encapsulation des données dans l'objet a un coût à l'exécution, dû aux appels de "`rep`" (coût de la désencapsulation), et aux copies d'objet (coût de la ré-encapsulation). Par contre, dans l'approche "module" de 3.1, l'encapsulation des données est remplacée par un mécanisme du typage sans surcoût à l'exécution.

Ensuite, les constructions syntaxiques implantant les espèces et collections n'ont pas une sémantique mathématique très claire. Par exemple, la collection des entiers est implantée par la classe `entiers` de M1. Un objet de cette classe correspond à une entité de la collection, mais porte en lui l'unité, et toutes les fonctionnalités qui font que cette collection appartient à l'espèce des anneaux. La correspondance objet-classe ne peut pas recevoir la sémantique individu-société. Cela se traduit notamment par la perte d'une arité des fonctionnalités évoquée en 4.1.1, avec l'inconvénient mentionné au niveau des fonctionnalités 0-aires. C'est incompatible avec une programmation de notions algébriques, mais surtout cela rend difficile l'expression des propriétés comme l'associativité ou la commutativité des opérations.

Pour remédier à cela, dans le modèle M4 suivant, nous rétablissons la correspondance entre type et arité des opérations. Cela permet notamment de créer un objet qui correspond à une collection.

```
class virtual anneau =
object (_:'a)
  method virtual egal:'a*'a->bool    method virtual plus:'a*'a->'a
  method virtual mult:'a*'a->'a      method virtual opp:'a->'a
  method virtual un:'a               ...
end

class entiers =
object
  inherit anneau
  val ma_rep=0
  method rep_entiers = ma_rep
  method egal (x,y) = x#rep_entiers=y#rep_entiers
  method plus (x,y) = < ma_rep=x#rep_entiers+y#rep_entiers>
  ....
end;;

# let entiers = new entiers;;
val entiers : entiers = <obj>
# let zero = entiers#plus (entiers#un,entiers#opp entiers#un);;
```

```
val zero : entiers = <obj>
# entiers#egal (entiers,zero);;
- : bool = true
```

Dans ce modèle, les entités restent encapsulées dans des objets. La distinction entre l'objet `entiers` et l'objet `zero` est une convention de programmation. Ce modèle a été développé jusqu'à l'implantation des polynômes distribués. Il ne dépend pas du mécanisme d'abstraction de type des modules. On ne le détaillera pas plus ici (voir [5]).

5. Solution retenue

5.1. Présentation

Pour distinguer par le typage entre collections et entités, nous allons développer un nouveau modèle M5 en supprimant la variable d'instance `ma_rep`. L'information sur l'ensemble support de la structure est passée en paramètre de type et les classes `anneau`, `entiers` et `z2z` de M4 deviennent :

```
class virtual ['a] anneau =
object
  method virtual egal:'a*'a->bool
  method virtual plus:'a*'a->'a
  ...
end

class entiers =
object
  inherit [int] anneau
  method egal (x,y) = x=y
  method plus (x,y) = x+y
  ...
end
```

```
class z2z =
object
  inherit [int] anneau
  method egal (x,y) = x=y
  method plus (x,y) =
    let tmp=x+y in
    if tmp=2 then 0 else tmp
  method mult (x,y) = x*y
  method opp x = x
  method un = 1
  method print x =
    (string_of_int x)^"[2]"
end
```

La classe `entiers` implante l'espèce des entiers qui regroupe les collections étendant les entiers et qui seront donc définies par héritage de cette classe. La collection qui définit la société des entiers doit permettre l'utilisation de ses opérations, mais doit masquer le type du support. Elle sera donc définie par un module `Entiers`. De même on définit la collection `Z2z` :

```
module type Anneau = sig   type t   val meth: t anneau   end
```

```
module Entiers : Anneau =
struct
  type t=int
  let meth=(new entiers)
end
```

```
module Z2z: Anneau =
struct
  type t=int
  let meth=(new z2z)
end
```

Les calculs dans ces collections sont faits avec le champ `meth` et le typage transcrit bien la spécification :

```
# let entiers = Entiers.meth;;
val entiers : Entiers.t anneau
# let un = entiers#un;;
val un : Entiers.t
```

```
# let z2z = Z2z.meth;;
val z2z : Z2z.t anneau = <obj>
# let unM2 = z2z#un;;
val unM2 : Z2z.t = <abstr>
```

```
# z2z#plus (unM2,un);;
This expression has type Z2z.t * Entiers.t but is here used
     with type    Z2z.t * Z2z.t
```

Une collection A est ici un couple `(t,meth)` où `t` est le type des entités, et `meth` est un objet "contenant" les méthodes de la collection. La représentation de `t` ne doit être connue que de la collection A et des collections qui étendent A et elle est masquée à tous les utilisateurs de cette collection. Les classes comme `entiers` servent de générateurs de collections. L'accès à la représentation reste possible à l'intérieur de la classe.

Ce mécanisme s'étend aisément aux collections paramétrées comme les polynômes. De plus, cette encapsulation des structures dans les modules pourrait sans doute être automatisée, comme le suggère la double utilisation de l'identificateur `entiers`.

Dans ce modèle M5, contrairement aux utilisations plus classiques de la programmation objet, un objet n'a pas d'état interne. Ce qui est essentiel ici est que la classe décrit complètement les fonctionnalités de l'espèce ou de la collection, un peu dans le même esprit que les types abstraits algébriques. C'est pourquoi nous l'appelons "modèle ADT". Il est à noter toutefois que l'on utilise dans cette réalisation toute la puissance d'expression fonctionnelle fournie par OCAML.

5.2. Autres traits utilisés

La hiérarchie des classes comprend actuellement une cinquantaine d'espèces : ensembles, ensembles ordonnés, treillis, monoïdes additifs, monoïdes multiplicatifs, groupes additifs, différentes variétés d'anneaux et d'algèbres.

Le développement de cette hiérachie utilise fortement les notions détaillées précédemment : l'héritage, les méthodes virtuelles/concrètes, la liaison tardive. On précise ici d'autres traits qui nous semblent importants.

Les variables de classes Ces variables vont être partagées par tous les objets de la classes. Elles sont notamment utiles dans le cas des espèces paramétrées, pour effectuer des opérations sur le paramètre, au moment de son instanciation. Par exemple, l'espèce `algebre` ci-dessous utilise la variable `under_moins_un` :

```
class virtual ['r,'a,'b] algebre (r:'r) =
  let under_moins_un = r#oppose r#un
  in
    object (the_alg)
      constraint 'r = ('a) #anneau_commutatif
      inherit ['b] anneau
      method virtual mult_extern : ('a*'b) -> 'b
      method opp =
        let ( * ) x y  = (the_alg#mult_extern)(x,y)
        in
          function x -> under_moins_un * x
end;;
```

Cette espèce étend la notion d'anneau avec une multiplication externe `mult_extern`, ce qui permet d'implanter la méthode `opp` (qui à été déclarée dans `annneau`). Le paramètre `r` désigne l'anneau sous-jacent, dont le type est `'r`. Le paramètre `'a` désigne le type support des entités de `r` et `'b` celui du type support des entités des collections instances de l'espèce `algebre`.

Les objets récursifs Étudions quelle en est l'utilité au travers de l'implantation récursive des polynômes à plusieurs variables. On suppose disposer d'une implantation, dite distribuée, nommée `algebre_polynomiale`, qui généralise la représentation creuse pour une seule variable. La construction récursive se fait par itération de la représentation distribuée. Ainsi, un polynôme de $A[X,Y,Z,U]$ peut être considéré comme un polynôme de $(((A[X])[Y])[Z])[U]$ ou de $(A[X,Y])[Z,U]$.

Nous devons d'abord donner une structure de données pour représenter les entités polynômes récursifs :

```
type ('a,'b) rec_struct =
  | Base of 'a
  | Composed of string*((('a,'b)rec_struct*'b) list);;
```

Les types `'a` et `'b` sont les structures de données utilisées pour représenter les coefficients et les degrés.

Les méthodes simples s'écrivent en utilisant cette récursion sur la structure de données. Par exemple, `mult_extern` multiplie un polynôme par un élément de l'anneau de base :

```
method mult_extern =
  let rec up_mult = function
    | (a,Base b) -> Base (under_multiplie(a,b))
    | (a,(Composed (v2,a2))) ->
        Composed(v2,
                 (List.map (fun (v,d) -> (up_mult(a,v), d)) a2))
  in fun (a,b) -> if under_is_zero(a) then zero else up_mult(a,b)
```

`under_is_zero` est ici une variable de classe qui teste l'égalité à zéro dans l'anneau de base.

Les méthodes plus compliquées doivent appeler les opérations sur les polynômes distribués. Il faut donc créér un objet de la classe `algebre_polynomiale`:

```
method dist_coll = new algebre_polynomiale (rec_rng,degs)
```

où `rec_rng` est l'objet de la classe en cours de définition (souvent nommé "`self`") et `degs` désigne la collection des exposants. Le type de `dist_coll` est de la forme `(...,'a,...) algebre_polynomiale`, où `'a` désigne le type de `rec_rng`. Remarquons que `rec_rng` est un objet récursif.

La méthode `dist_coll` sert à écrire les opérations comme l'impression :

```
method print = function
      | Base a -> under_print a
      | Composed (v,p) -> ((rec_rng#dist_coll)#output)(p,v)
```

La méthode `output` est définie dans `algebre_polynomiale`. Elle affiche un polynôme en attribuant un nom aux variables et en appelant elle-même la méthode `print` de `rec_rng`. La fonction `print` effectue donc une itération sur toutes les couches de distribution par ce seul appel à `output`.

Il est à noter qu'on travaille par "paquets" de variables, ce qui est plus général que ce qui se fait habituellement en calcul formel. Cette implantation des polynômes récursifs serait difficile à réaliser à l'aide uniquement de modules, car il faudrait itérer l'appel au "foncteur `algebre_polynomiale`".

5.3. Adéquation du modèle

Ce modèle répond bien au cahier des charges. Les espèces sont des classes virtuelles : un groupe est une sous-classe d'un monoïde, donc apparaît bien comme un monoïde enrichi. Les différents mécanismes d'abstraction utilisés (héritage, paramétrage, liaison tardive) permettent de ne faire figurer dans la définition de l'espèce que ce qui est prescrit dans la définition du genre qu'elle implante. Malgré toutes ces dépendances, on conserve un typage très fort jusqu'à l'implantation des collections. Avec ce modèle M5, notre projet d'implantation de structures algébriques peut se faire directement dans un langage d'usage général. On économise ainsi tout le travail de construction d'un langage dédié et on bénéficie des nombreuses librairies déjà développées.

6. Conclusion

Nous avons tenté dans cet article de dégager, à partir de nos expériences, les traits nécessaires à la programmation de l'algèbre. Les développements réalisés pour cela sont de taille relativement importante. Nous avons ainsi introduit 50 espèces que nous considérons comme essentielles. À titre de comparaison, Axiom en possède 46. Ces 50 espèces ne nécessitent que 700 lignes de Ocaml (1800 lignes en Axiom). Nous avons eu constamment le souci de rester aussi proches que possible de la formulation mathématique habituelle.

La réalisation de ces différents prototypes nous a permis de concevoir une architecture de développement originale, qui tire au mieux parti des traits objet et modulaires de Ocaml, pour offrir un système souple, ouvert, suffisamment puissant pour exprimer les algorithmes de calcul formel. Par exemple, à notre connaissance, la représentaion récursive des polynômes offre beaucoup plus de possibilités que ce qui est proposé par d'autres systèmes. Les résultats du point de vue efficacité sont encourageants.

En conclusion, le débat "module" versus "objet" ne nous paraît pas bien posé. Il nous semble qu'il faut plutôt opposer une conception "par objet" et une conception "ADT", qui se traduisent plus ou moins facilement dans les deux styles. Le modèle retenu implante le style ADT à l'aide des objets de Ocaml. Il pourrait être intéressant de disposer d'un système de modules plus puissant. En effet, les systèmes d'aide à la preuve savent mieux traiter les modules que les objets. Par exemple, J. Courant, dans [4], a montré comment étendre le système de modules de Ocaml pour Coq. Dans la cadre d'une bibliothèque de calcul formel, il est fortement souhaitable d'étendre les systèmes de modules avec des traits des classes de la programmation objet. Une recherche active s'effectue dans ce sens avec les travaux de J. Vouillon [10], les théories comme les mixins [2] ou en utilisant les substitutions explicites [9].

7. Remerciements

Nous tenons à remercier le projet CRISTAL, particulièrement Xavier Leroy, Didier Rémy, François Rouaix et Jérôme Vouillon. Nous remercions également les autres membres de l'action incitative INRIA-CFC, et paticulièrement Loïc Pottier et Laurent Théry.

Références

[1] Guillaume ALEXANDRE. *D'Axiom à Zermelo.* Thèse de l'université Paris 6, février 1998.

[2] Davide ANCONA et Elena ZUCCA. An Algebra of Mixin Modules. In *WADT'97*, pp. 92-106, LNCS numéro 1376, Springer-Verlag, 1998.

[3] Richard D. JENKS and Robert S. STUTOR. *AXIOM, The Scientific Computation System.* Springer-Verlag, 1992.

[4] Judicaël COURANT, *MC : un calcul de modules pour les systèmes de types purs.* Thèse de doctorat, École normale supérieure de Lyon, février 1998.

[5] S. BOULMÉ, T. HARDIN, V. MÉNISSIER, R. RIOBOO. *Rapport Foc.* Rapport interne LIP6. À paraître.

[6] Xavier LEROY. *The Objective Caml system, release 2.0.* Logiciel et documentation : `http://pauillac.inria.fr/ocaml/`, 1998.

[7] Didier RÉMY. *Des enregistrements aux objets.* Mémoire d'habilitation à diriger des recherches en informatique. Université Paris 7, Septembre 1998.

[8] Didier RÉMY et Jérôme VOUILLON. Objective ML : An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 1998, À paraître.

[9] Anne SALVENSEN. Modules as classes in logical frameworks with explicit substitution. *Selected papers from the 8th Nordic Workshop on Programming Theory*, pp. 199-214. Oslo, Norway, December 1996.

[10] Jérôme VOUILLON. Using modules as classes. In *Informal proceedings of the FOOL'5 workshop*, 1998. Disponible à `http://pauillac.inria.fr/~remy/fool`

[11] S. WATT, P. BROADBERY, S. DOOLEY, P. IGLIO, S. MORRISON, J. STEINBACH, et R. SUTOR. *AXIOM Library Compiler User Guide.* NAG Ltd, March 1995.