AXIOM,
Langage fonctionnel à développement objet.

Jean-Louis BOULANGER
Laboratoire d'Informatique Fondamentale de Lille
U.A 369 du C.N.R.S, Université de Lille I
59655 Villeneuve d'Ascq Cedex. FRANCE.

mail: boulang@lifl.lifl.fr

10 Septembre 1993

Abstract

Nous profitons de cet article pour présenter une étude que nous avons faite d'un langage dédié au calcul formel qui présente des concepts de génie logiciel très intéressants. Ce langage est un langage fonctionnel dont le développement est orienté objet, ceci a pour première conséquence de trés bonnes capacités pour modéliser les mathématiques (Hiérarchie) et une exécution claire au sens mathématique (Tout est fonction). Nous soulignerons les faiblesses du modèle au travers d'une analyse du comportement de l'exécution et des problèmes de développements. Il faut noter que le langage Axiom [1] est toujours le seul de sa catégorie.

Mots clés:
Langage fonctionnel, typage polymorphe, héritage simple et multiple, coercion et inférence de type.

[1] Axiom (anciennement SCRATCHPAD II) est un produit qui est distribué par NAG et conçu à l'origine par IBM.

1 Introduction.

Il est important de bien connaître la structure d'un langage de programmation afin d'utiliser au mieux celui-ci et d'en comprendre le comportement. Axiom est un langage de programmation ayant un certain nombre de concepts très intéressants et novatifs, mais dont l'utilisation est parfois périlleuse. Il faut à tout prix bien en comprendre tous les mécanismes de développement et d'exécution qui permettent une grande souplesse mais qui peuvent entrainer des exécutions peu controlées. C'est pourquoi Axiom peut grâce à sa souplesse d'utilisation modéliser toute la rigueur des mathématiques.

Il serait appréciable d'obtenir des exécutables aussi petits que possible ou mieux un code portable. On trouve là, la source de nos motivations[2], un système de calcul formel efficace et générant des applications indépendantes.

2 Axiom : langage fonctionnel.

Axiom est avant tout un langage fonctionnel, qui a pour père un lisp particulier et qui a été nourri des divers bouleversements qu'a ressenti le monde informatique depuis quinze ans, on trouvera d'ailleurs un historique dans [6] et [7]. En effet, la notion de catégorie et de domaine est héritée des travaux de Barbara Liskov sur le langage CLU. On retrouve dans Axiom tout l'arsenal d'outils utilisés par les langages fonctionnels de la famille de KRC, ML et de MIRANDA. L'ensemble des notions présentées ici trouvent leur origine dans [9].

2.1 Un monde de fonction.

Comme son nom l'indique, le modèle fonctionnel est basé sur la manipulation de fonctions. Ce qui se traduit par le respect de deux propriétés :

1. la fonction se trouve être le seul objet manipulé par programme,
2. la structure de contrôle essentielle se trouve être l'appel de fonction.

```
fib 1 == 1                             -- Definition au cas par cas
fib 2 == 1
fib n == fib(n-1) + fib(n-2)

PolyLegendre n == n=0 => 1             -- Definition a l'aide de garde
                  n=1 => x
                  ((2*n-1)*PolyLegendre(n-1)-(n-1)*PolyLegendre(n-2))/n


-- Exemples de fonctions utilisant les ZFexpressions

ProduitCart(x,y) == [[a,b] for a in x for b in y]

Trie (ll | ll=nil) == []
Trie ll            == append(Trie([b for b in rest(ll) | b<=first(ll)]),
                       cons(first(ll),
                             Trie([b for b in rest(ll) | b>first(ll)])))
```

Figure 1: Quelques fonctions Axiom.

Le caractère fonctionnel d'Axiom, permet de considérer que tout est fonction. Quelques exemples de fonctions :

- toutes les fonctions définies par l'utilisateur (voir figure 1),
- un type est une fonction qui renvoie une structure de données,
- l'affectation est une fonction [2].

```
    square n == n*n                         -- Une fonction calculant le carre.

    derive(f,x,dx) == (f(x+dx)-f(x))/dx     -- Une approximation de la derivee
                                               d'une fonction si dx tres petit.
    derive(square,10,0.00000000001)         -- Un exemple d'utilisation.

-- Operation de reduction d'une liste.
    reduce(x,f,a) == if x=nil then a
                          else g(car(x),reduce(cdr(x),g,a))

    add(x,y) == x+y
    sum(x)   == reduce(x,add,0)             -- Somme des elements d'une liste.
```

Figure 2: Manipulation de fonction.

En fait, on dit que les fonctions sont des objets du premier ordre, c'est à dire qu'elles sont manipulables (voir les figures 2 et 3) par programme.

La définition suivante

```
t : (float-> Float, Float) -> Float
t(fun,x) == fun(x)**2+sin(x)**2
```

permet d'exécuter l'instruction *t(cos, 5.2058)* qui a pour résultat *1.0 : Float*

Figure 3: Manipulation de fonction.

Axiom est un système qui permet les différentes formes de polymorphisme [3], ce qui entraine une certaine complexité dans la gestion des fonctions (voir Section 2.3).

2.2 Notion de type.

Axiom est un langage fonctionnel fortement typé, ce qui implique que toutes les variables dispose d'un type. Ce typage est statique, le type d'une variable ne peux pas évoluer au cours du temps.

Axiom propose toute la palette des types de bases habituels, tel que *Integer*, *Rational*, *Float*, *List*, *Record* et d'autre tel que les *Streams*. Mais Axiom est un système beaucoup plus riche, car il propose une hiérarchisation des types correspondant à celle des mathématiques. On trouve alors les monoides, groupes, anneaux et tout autre structure mathématique classique.

Afin de faciliter, la construction de programme, l'utilisateur n'est pas obligé de typer toutes ces variables. Ce qui entraine la mise en place d'un système d'inférence de type et d'outils associés tel que les coercions de type et la résolution de type.

[2] Il est bien évident que l'affectation n'est pas implémentée de cette façon, mais qu'elle profite de la sous couche LISP.

[3] Cette question sera traitée plus longuement (voir Section 4).

succ x == x+1

Figure 4: La fonction successeur.

La déclaration de fonction de la figure 4, nous indique que l'on peut appliquer l'opération *succ* à tout ensemble de valeur comportant une opération +. Mathématiquement on peut en conclure que $succ : L -- > L$ avec L qui est un Monoide Abélien.

2.3 Gestion des fonctions.

Le nombre de fonctions et la présence des différents types de polymorphisme entraine un nombre de fonctions important. C'est pourquoi, l'interpréteur utilise une base de donnée qui effectue la gestion et le choix des fonctions.

Définition 2.1 *Une signature Axiom (ou modemap) est une extension de la notion courante de signature.*

$$nom_fonction(module, type_res, type_par_1, ..., type_par_n) \quad if \ predicat$$

que l'on doit lire

$$nom_fonction \ : (type_par_1, ..., type_par_n) - > type_res \ from \ module \ if \ predicat$$

Les signatures :

```
trace : Matrix(R) -> R
   if R has Ring from Matrix(R)
inverse : Matrix(R) -> Union(R,"Failed")
   if R has Field from Matrix(R)
```

deviennent :

```
trace : *1 -> *2
   if *2 has Ring and *1 is Matrix(*2) from *1
inverse :*1 -> Union(*2,"Failed")
   if *2 has Field and *1 is Matrix(*2) from *1
```

Figure 5: Illustration de la standardisation des signatures.

Définition 2.2 *La base de donnée ne contient que des signatures standardisées. La standardisation permet d'homogénéiser la structure des signatures. La figure 5 présente un exemple de standardisation des signatures.*

Définition 2.3 *La standardisation est basée sur deux opérateurs d'appartenance. L'opérateur* is *est une égalité explicite, tandis que l'opérateur* has *est un lien du type* est_un [4].

Recherche de fonction. La base de donnée devra alors être capable de répondre à une requète de recherche d'une fonction définie par sa signature. Le polymorphisme entraine différents cas.

[4] On parle en général de lien *is_a*. T_1 *is_a* T_2 indique que T_1 doit avoir pour surtype T_1.

1. une fonction convient, le cas idéal,
2. plusieurs fonctions valident la signature, on choisit alors la première trouvée,
3. aucune fonction ne correspond à la signature, on approfondit la recherche, on renvoie l'ensemble des fonctions de nom et d'arités correspondantes,
4. pas de fonction ayant ce nom d'où génération d'erreur.

2.4 Coercion et conversion.

Axiom est un système conçu autour d'un interpréteur de commandes. C'est pourquoi il n'y a pas de primitive d'entrée/sortie mais des fonctions qui permettent de convertir les entrées en structure de données et celles-ci en affichage.

Définition 2.4 *Une* conversion *est une opération qui construit un objet O_2 d'un type T_2 à partir d'un objet O_1 de type T_1, l'objet O_2 ayant des propriétés dans T_2 équivalentes à celle de O_1 dans T_1.*

Afin de rendre cette tâche transparente, le système peut effectuer des transformations de façon implicite ou explicite.

Définition 2.5 *On appelle* coerce *toute fonction de conversion que le système peut exécuter de façon implicite* [5]*. On parle en général de coercions.*

Remarque 2.1 *Etant donné le caractère implicite des coercions, il est préférable d'utiliser des opérations ayant une validité algébrique ou ne détruisant pas d'informations.*

Par validité des coercions, on pense au fait que tout *Integer* peut être transformé en *Real* mais que l'inverse introduit une perte d'information. Dans [?], on trouve une breve definition et utilisation des opérations *coerce* dans le cadre de la compilation.

On dispose aussi de fonctions *convert* mais elles doivent être utilisées explicitement.

Définition 2.6 *L'opérateur* :: *permet d'indiquer une demande de conversion. La conversion pouvant entrainer l'utilisation de* coerce *et* convert*.*

$$Variable :: Type$$

Le caractère implicite des fonctions *coerce* est très important et en fera l'outil central de l'inférence de type.

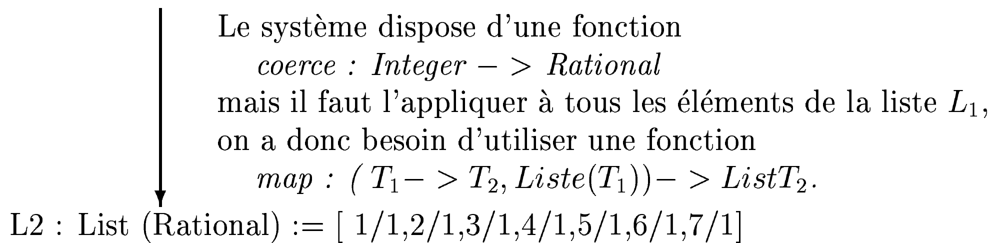


Figure 6: Exemple de coercion par mapping.

Définition 2.7 *On parle de* coercion par mapping *lorsque le système est amené à combiner les fonctions* map *au coercions, ceci afin de convertir un type $D(T_1)$ en $D(T_2)$* [6]*.*

[5] Implicite signifie que le système peut les utiliser quand bon lui semble et sans demande de l'utilisateur.
[6] Avec D qui est un constructeur quelconque.

Remarque 2.2 *Axiom est actuellement un des seuls systèmes à offrir la notion de coercion, en effet, dans la pratique, les autres langages* [7] *offrent une notion de coercion entre types simples* [8] *et sans possibilité d'ajout pour l'utilisateur.*

Il existe deux sources pour ces opérations :

- les fonctions définies par programme,
- les fonctions intégrées au système [9].

Nous avons présenté dans la section 2.3 la standardisation et nous avons vu qu'elle introduisait un prédicat que le type des objets mis en oeuvre dans la validation d'une signature devaient vérifier.

Définition 2.8 *Soit un prédicat* P *et un ensemble de type* S, *on appelle* forçage de type *l'ensemble des conversion mis en jeu afin que* S *satisfasse le prédicat* P.

Exemple: Soit le prédicat $P =$ **1 has Field*, et supposons que l'on veuille substituer *Integer* à *1. Les *Integer* ne sont pas un corps mais il existe un constructeur *QuotientField(.)* qui permet d'obtenir un corps à partir des *Integer*. Il y a donc un forçage de type.

2.5 Rétraction d'un type.

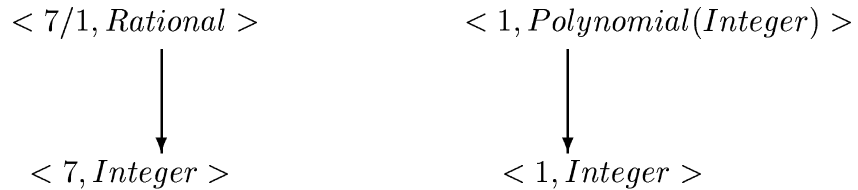


Figure 7: Exemples de Retraction.

Définition 2.9 *La rétraction est une opération qui permet de faire passer le couple* $< V_1, T_1 >$ [10] *au couple* $< V_2, T_2 >$ *s'il existe une forme dégénérée du type* T_1 *dans laquelle* V_1 *peut être coercé.*

La retraction est basée sur deux opérations que l'utilisateur devra implémenter pour chaque nouveau type :

1. *Retractable? $ − > Boolean* ,
2. *Retract $ − > R.*

Ces opération sont définies dans la catégorie *RetractTo(R).*

2.6 Résolution de type.

Disposant des conversions et de la composition de fonction, on peut maintenant introduire la notion de chemin et de résolution de type.

Définition 2.10 *Un* chemin *entre les types* T_1 *et* T_2*, est défini par une chaine de conversion, permettant de transformer un élément du type* T_1 *en type* T_2.

[7] Par langage, on entend ceux du calcul formel mais aussi les langages de programmation tels que CAML.
[8] On dispose en général de coercion tel qu'entier vers réel.
[9] L'utilisateur ne disposant pas du contrôle de ces fonctions.
[10] Le couple $< V, T >$ se lit $< Valeur, Type >$.

Définition 2.11 *Soit* deux *types différents, on appelle* résolution *le processus de recherche d'un type d'arrivée commun aux* deux *types de départ et des chemins d'accés respectifs.*

Propriétés 2.1 *Quels que soient les deux paramètres :*

- *Une résolution réussira toujours, grâce au type ANY,*
- *L'opération de résolution est symétrique.*

Il faut noter que la résolution de type d'Axiom correspond à l'unification de Prolog. Le mécanisme de résolution est basé sur :

- les opérations de conversions,
- un ensemble de manipulations prédéfinies [11].

2.7 Inférence de type.

Axiom est un système fortement typé, c'est-à-dire que toute variable a un type. Il faut souligner que ce typage est statique [12]. Mais l'utilisateur n'est pas obligé de déclarer toutes les variables utilisées dans sa session. Comme on dispose d'un système supportant différentes formes de polymorphisme, le choix de la fonction à utiliser peut être multiple.

L'inférence de type fut d'abord introduite dans le langage ML (1976), mais Axiom en offre une forme plus évoluée.

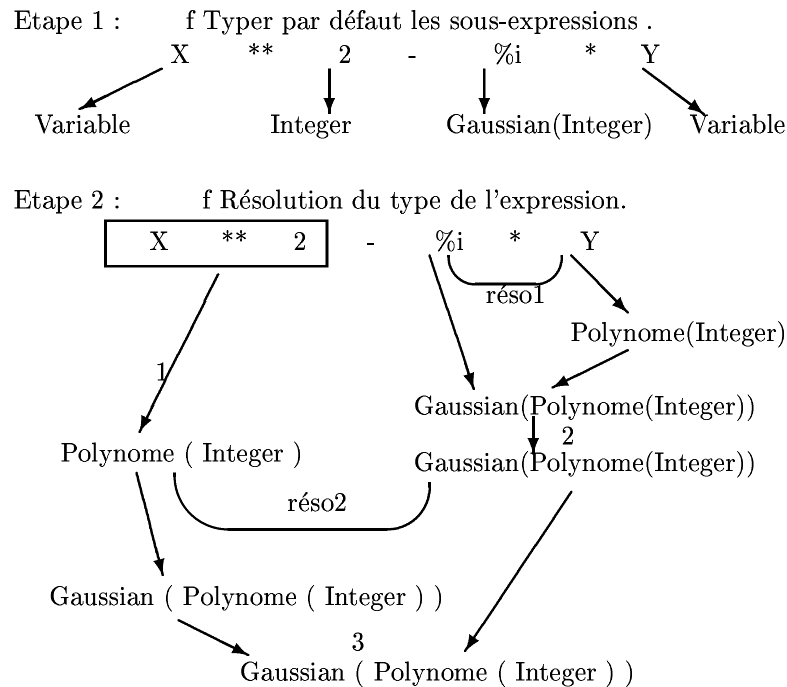


Figure 8: Un exemple d'inférence de type.

L'interpréteur doit alors prendre en charge le calcul du type des expressions et la gestion des variables. Il utilisera alors les coercions et les informations de la base de données des fonctions. On voit que la vérification de type des langages impératifs ne suffirait pas, car elle ne fait qu'utiliser l'information existante et que dans notre cas il faut extraire l'information de types du programme lui-même.

[11] L'utilisateur ne disposant pas du contrôle de ces manipulations.

[12] Le type d'une variable n'évolue pas au cours du temps.

Règles de base de l'inférence de type :

1. On type toutes les feuilles de l'arbre représentant l'expression.
2. On va typer chaque sous-arbre, en commençant par le plus interne.
3. On type de gauche à droite.

Explicitation de la figure :

1. On dispose d'une opération **(Polynome, Polynome(R), Variable, R) qui peut être appliquée pour R = Integer.
2. Comme la seule opération exécutable est *(??,R, R, R) on a effectué une résolution de type. On a R = Gaussian(Polynome(Integer)).
3. A nouveau on effectue une résolution de type car la seule opération acceptable est -(??, R ,R ,R), avec R = Gaussian(Polynome(Integer)).

Il faut remarquer, que pendant la phase d'inférence de type, on est amené à lire de nouveaux modules et à effectuer l'instanciation des modules génériques. L'inférence de type est un des outils centraux d'Axiom, de part son caractère implicite. Il faut y penser constamment, surtout si on laisse l'interpréteur faire un maximum de travail.

Par exemple dans les cas suivants :

- Variable non déclarée,
- Le Type des variables est différent,
- Appel de fonction polymorphe :
  - fonction ayant plusieurs déclarations,
  - fonction ayant différentes signatures.

Le calcul de type peut entrainer des choix que l'utilisateur n'aurait pas fait et entrainer des résultats qui semblent erronés à première vue mais qui sont justes dans le contexte de l'interpréteur. Il faut penser à donner un maximum d'informations en typant les variables et en indiquant plus précisement quelle fonction on désire utiliser.

Comparaison avec ML. La différence fondamentale réside dans le fait qu'Axiom manipule et/ou transforme le type des varaiables et que ceci se fait à toute interprétation de l'expression. En ML, l'inférence de type est effectuée une fois pour toute et il n'existe pas de coercion.

2.8 Conclusion.

Un des caractères importants des langages fonctionnels réside dans une syntaxe plus libre qui permet d'écrire des programmes concis et très proches de leur forme mathématique, ce qui permet d'utiliser les méthodes de démonstration classique pour les preuves de programmes.

Pour arriver à ce résultat, Axiom utilise un certain nombre d'outils trés intéressants, mais qui ont tendance a ralentir les traitements. Qui plus est l'utilisateur doit penser à fournir les outils nécessaires à l'inférence de type mais cela ne lui confère pas tout le contrôle des transformations effectuées par le système.

3 Axiom : langage à développement objet.

Comme nous allons le voir, Axiom fournit, au niveau développement, tous les outils de la programmation objet. On retrouve la notion de classe (qui est à deux niveaux), la notion d'héritage simple et multiple, mais l'exécution s'effectue dans un modèle fonctionnel [13]. On trouve dans [10] et [11] d'autres exemples concernant les sections suivantes.

[13] Le modèle d'exécution a été présenté dans la section 1.

3.1 Notion de type et de classe.

3.1.1 Les Catégories ou les spécifications.

Une Catégorie se trouve être la spécification d'un type, on parle en général de type abstrait [14].

```
)abb category CATS CatSomme

CatSomme() : Category == with
      "+" :($,$) -> $
      "-" :($,$) -> $
      "-" : $ -> $
  add
      x-y == x + (-y)
```

Figure 9: Une catégorie avec implémentation par défaut.

Dans le cadre de la spécification formelle, on introduit des axiomes caractérisant l'action des opérations, parmi ces axiomes certains sont constructifs et permettent la définition d'opération par défaut comme dans la figure 9. La figure 9 introduit la notion d'abbréviation au travers de l'opérateur *)abb*.

```
)abb category CP2D CatPoint2D
CatPoint2D(R:AbelianGroup) : Category == SetCategory with
   coerce    :  List(R) -> $
   D         :  $       -> R

)abb category CP2DC CatPoint2DColore
CatPoint2DColore(R:AbelianGroup) : Category == CatPoint2D(R) with
   Init      : ($,COULEUR) -> $
   C         :  $          -> COULEUR

)abb category CP2DM CatPoint2DMobile
CatPoint2DMobile(R:AbelianGroup) : Category == CatPoint2D(R) with
   Translate : ($,R,R) ->$

)abb category CP2DMC CatPoint2DMobileColore
CatPoint2DMobileColore(R:AbelianGroup) : Category ==
   Join(Catpoint2DColore(R),CatPoint2DMobile(R))
```

Figure 10: Introduction de l'héritage abstrait des catégories.

La figure 10 introduit la notion d'héritage simple et multiple. L'héritage multiple étant introduit par le mot *Join(...)*, voir la catégorie *CatPoint2DMobileColore*. Axiom permet de définir et de quantifier l'existence des opérations lors de la conception d'une catégorie ou d'un domaine (voire figure 11), cette quantification peut-être faite dans le cadre des spécifications ou de l'implémentation.

[14] Les langages orientés objets introduisent en général la notion de classe abstraite.

```
)abb category AMR AbelianMonoidRing

AbelianMonoidRing(R:Ring, E:OrderedAbelianMonoid): Category ==
  Join(Ring,BiModule(R,R)) with
    leadingCoefficient: $ -> R
    leadingMonomial: $ -> $
    degree: $ -> E
    map: (R -> R, $) -> $
    monomial?: $ -> Boolean
    monomial: (R,E) -> $
    reductum: $ -> $
    coefficient: ($,E) -> R
    if R has Field then "/": ($,R) -> $
    if R has CommutativeRing then
       CommutativeRing
       Algebra R
    if R has CharacteristicZero then CharacteristicZero
    if R has CharacteristicNonZero then CharacteristicNonZero
    if R has IntegralDomain then IntegralDomain
    if R has Algebra Fraction Integer then Algebra Fraction Integer
  add
    monomial? x == zero? reductum x
    if R has Algebra Fraction Integer then
       q:Fraction(Integer) * p:$ == map(q * #1, p)
```

Figure 11: Une Catégorie quantifiée.

3.1.2 Les Domaines ou les implémentations.

La notion de domaine est liée à l'implémentation d'une classe abstraite, la figure 12 montre d'ailleurs une implémentation du type *CatPoint2D(R)* réaliser par le domaine *Point2D(R)*. Un domaine devra donc fournir une représentation et une réalisation particulière du comportement.

La figure 13 au niveau du domaine *PointMobile2D* fait apparaitre une séparation entre héritage de spécification et d'implémentation. On peut aussi dire qu'afin de limiter tout conflit d'implémentation, l'héritage à ce niveau est toujours un héritage simple.

Un des problèmes du développement en Axiom réside dans les conflits de représentation. En effet, la structure d'un objet est introduite par le mot *Rep* et une structure ne peut être ni enrichie ni appauvrie. Toute modification entraine la redéfinition des opérations associées. Dans l'exemple de la figure 13 au niveau du domaine *PointColore2D* [15] nous sommes obligés de redéfinir toutes les fonctions qui touches à la structure de l'objet, même les fonctions *X(.)* et *Y(.)*. Une autre solution consisterait à ajouter des fonctions *coerce* qui effectuerait les conversions. Cela prouve que le lien d'héritage n'est pas un lien *is_a* mais que l'on a un héritage comportemental.

3.1.3 Les paquetages ou collections de fonctions.

Les pacquetages sont une facilité pour gérer les fonctions manipulant des objets communs. Ce sont en fait des collections de fonctions que l'on peut paramétrer comme les catégories et les domaines par des types et des fonctions.

[15] L'héritage d'implémentation dans la classe *PointColore2D* est inutile dans ce cas.

```
)abb domain P2D Point2D

Point2D(R:AbelianGroup) : Specif == Imple where
   Specif ==> CatPoint2D(R) with
     X : $ -> R
     Y : $ -> R
   Imple  ==>
         Rep := Record(x:R,y:R)
         X pt == pt.x
         Y pt == pt.y
         coerce pt ==
                    print(p.x)
                    print(p.y)
         coerce l  == [l.1,l.2]
```

Figure 12: Une implémentation du type CatPoint2D.

Le paquetage présenté figure 14 permet de construire une fonction générique [16] qui instanciée correctement, permet par exemple de calculer le factoriel d'un nombre ou de recopier une liste.

3.1.4 Différence entre Catégorie et Domaine.

Il peut paraitre difficile de trouver une différence entre la notion de Catégorie et celle de Domaine. Mais il semble qu'elle soit fondamentale, les catégories correspondent à la notion de type/sous-type et les domaines à celle de classe/sous-classe. Les langages orientés objets peuvent fournir deux niveaux de définition (classe et métaclasse) mais dans ce cas une différence notable existe. En effet, une métaclasse fournit un comportement pour manipuler les classes et l'ensemble des métaclasse s'intégre à la hiérarchie existante, d'où on n'introduit pas de hiérarchie supplémentaire.

3.2 L'héritage en question.

3.2.1 Héritage simple et multiple.

On voit qu'Axiom permet de définir un nouveau module [17] à partir de ceux existants en utilisant des liens d'héritages multiples pour les spécifications (voir figure 10) et simple pour les implémentations (voir figure 12 au niveau du domaine *PointMobile2D*). On voit qu'il y a différents types et niveaux d'héritages.

3.2.2 Trois arbres d'héritages différents.

La différence entre

1. Catégorie et Domaine,
2. Spécification et Implémentation.

fait apparaitre trois arbres d'héritages bien différents qui sont :

- Hiérarchie abstraite des catégories,
- Hiérarchie par défaut (on parle aussi de chaine add),
- Hiérarchie d'implémentation (héritage simple).

[16] La fonction présentée en exemple a été emprumté à [1] page 105-107.
[17] Un module pouvant être au choix une catégorie, un domaine ou un package.

```
)abb domain PM2D PointMobile2D
PointMobile2D(R:AbelianGroup) : Specif == Imple where
   Specif ==> CatPoint2dMobile(R)
   Imple  ==> Point2D(R) add
         Translate(pt,xx,yy) == [Y(pt)+xx,Y(pt)+yy]

)abb domain PC2D PointColore2D
PointColore2D(R:AbelianGroup) : Specif == Imple where
 Specif ==> CatPoint2DColore(R)
 Imple  ==> Point2D(R) add
        Rep := Record(x:R,y:R,c:COULEUR)

        X pt == pt.x
        Y pt == pt.y
        coerce(pt)    == hconcat(X(pt)::OutputForm,
                            hconcat(hconcat(" ",Y(pt)::OutputForm),
                            hconcat(" ",pt.c::OutputForm)))
        coerce l      == [l.1,l.2,0]
        Init(pt,co)   == pt.c:=co
                         pt
        C pt          == pt.c
```

Figure 13: Exemple d'implémentation des différents points.

3.2.3 Algorithme de parcours des arbres d'héritages.

Nous allons maintenant présenter l'algorithme de parcours des arbres d'héritages tel qu'il est présenté dans [11]. Nous n'allons pas discuter de son efficacité ou de son adéquation, on trouve dans la littérature liée aux langages orientés objets de nombreuses études liées à ce sujet (voir par exemple [12] et [5]).

La recherche d'opérations se fait dans l'ordre :

1. la hiérarchie d'implémentation,
2. la hiérarchie par défaut,
3. la hiérarchie abstraite des Catégories

Pour l'héritage multiple l'ordre de parcours est celui introduit par l'utilisateur dans sa construction, on parle en général d'ordre avec priorité. Pour finir, on peut penser qu'il y a de petites améliorations à apporter ou des choix à discuter, afin que cet algorithme prenne en compte son domaine d'utilisation : *Les mathématiques.*

3.2.4 Un petit manque.

La notion de graphe d'héritage est très intéressante et permet une bonne réutilisabilité. Ce principe de réutilisabilité semble être un des principes les plus importants vus les monstres logiciels actuels. De ce point de vue, Axiom souffre d'un petit manque, en effet l'utilisateur se trouve dans l'obligation de casser ses arbres et de définir différentes classes au comportement identique (voir figure 15).

Ceci est dû au fait qu'en mathématique on définit un modèle abstrait (tel que le groupe) avec un certain nombre de propriétés (associativité,...) et d'opérations (une opération $+$ interne pour le groupe,..) et qu'au niveau de l'instanciation on peut considérer toute structure ayant une opération répondant aux propriétées énoncées.

```
)package ABST Abstract

Abstract (R:SetCategory,
          vide?   :R->Boolean,
          Sivide  :R->R,
          compose :(R,R)->R,
          premier :R->R,
          reste   :R->R) : public == private where
   public ==> with
      Abstraire: R -> R
   private ==> add
      Abstraire(entite) ==
                   if vide?(entite)
                     then Sivide(entite)
                     else compose(premier(entite),Abstraire(reste(entite)))
```

Figure 14: Le pacquetage Abstract.

En fait, il manque à Axiom, la notion de point de vue, que l'on trouve dans des langages comme OBJ.

3.3 Conclusion.

Le développement de programmes Axiom est parfois difficile vu la complexité de sa notion d'héritage et la différence notable entre modèle d'exécution et modèle de développement. Il reste à noter que les mathématiques font elles aussi un usage intensif des notions d'abstraction et d'héritage.

En fait, on se rend vite compte qu'il manque à Axiom un certain nombre d'outils tel qu'un Browseur/Editeur qui permettrait de visualiser les hiérarchies d'une classe, de savoir quel est l'ensemble des opérations accessibles par cette classe et d'indiquer qu'elles sont celles qui sont effectivement implémentées.

4 Différentes formes de polymorphisme.

La notion de type est certainement la notion la plus importante, mais elle doit être associée à la notion de polymorphisme qui offre toute sa puissance au système de type. On dispose de quatres types de polymorphisme:

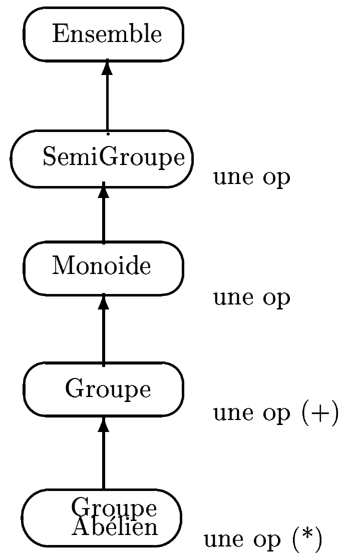
$$Polymorphisme = \begin{cases} Universel & \begin{cases} Parametrique \\ Inclusion \end{cases} \\ Adhoc & \begin{cases} Surcharge \\ Coercion \end{cases} \end{cases}$$

Ce diagramme représente les différentes formes du polymorphisme, on pourra se reporter à [3]. Remarquons qu'Axiom assume les diverses formes du polymorphisme.

4.1 Polymorphisme Ad hoc.

Le polymorphisme Ad hoc est une facilité d'écriture, qui permet de regrouper sous un même nom de fonction différents traitements. En effet le traitement appliqué aux données est différent et dépend de leurs types.

Du point de vue Mathématique :



En Axiom :

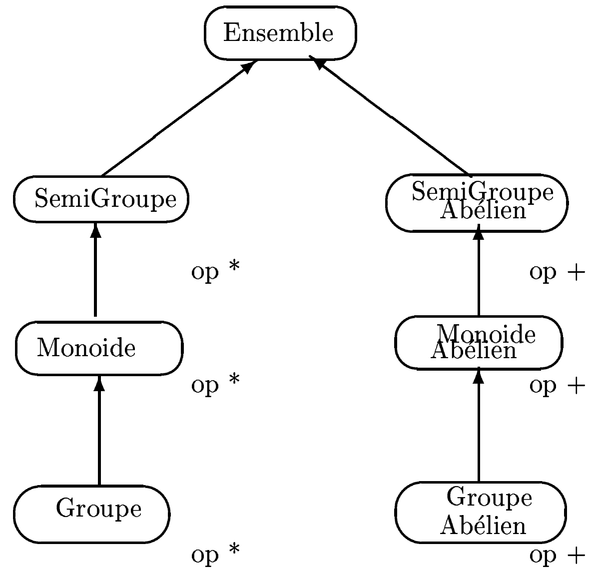


Figure 15: Un exemple de diférence entre mathématique et implémentation.

4.1.1 La surcharge.

La forme la plus connnue du polymorphisme Ad hoc reste la surcharge telle qu'elle est utilisée dans le langage ADA. On dispose de plusieurs fonctions ayant le même nom mais dont le type et le nombre des paramètres sont différents et avec des corps de fonctions au codes différents.

```
+ : (Mon_entier,Mon_entier) -> Mon_entier
+ : (Mon_entier,Mon_entier,Mon_entier) -> Mon_entier
+ : (Ma_liste,Ma_liste) -> Ma_liste
+ : (Matrice,Matrice) -> Matrice
```

Figure 16: Exemple de surcharge.

La figure 16 montre que ce polymorphisme n'est qu'une commodité d'écriture permettant de surcharger le nom d'une opération. Il faut reconnaitre que la surcharge existe dans tous les langages, mais sous une forme inachevée.

4.1.2 La coercion.

Dans la section 2.4, nous avons déjà introduit les notions relatives aux coercions, mais sans signifier alors qu'elle introduisait du polymorphisme.

La figure 17 nous indique que le type point représenté par la catégorie *CatPoint* dispose de plusieures implémentations, qui sont les domaines *PointCartesien* et *PointPolaire*, ces deux représentations étant équivalentes on dispose d'opérations *coerce* effectuant les changements de repère.

La figure 18 utilise les définitions de la figure 17 et montre l'utilisation du polymorphisme dû au coercion.

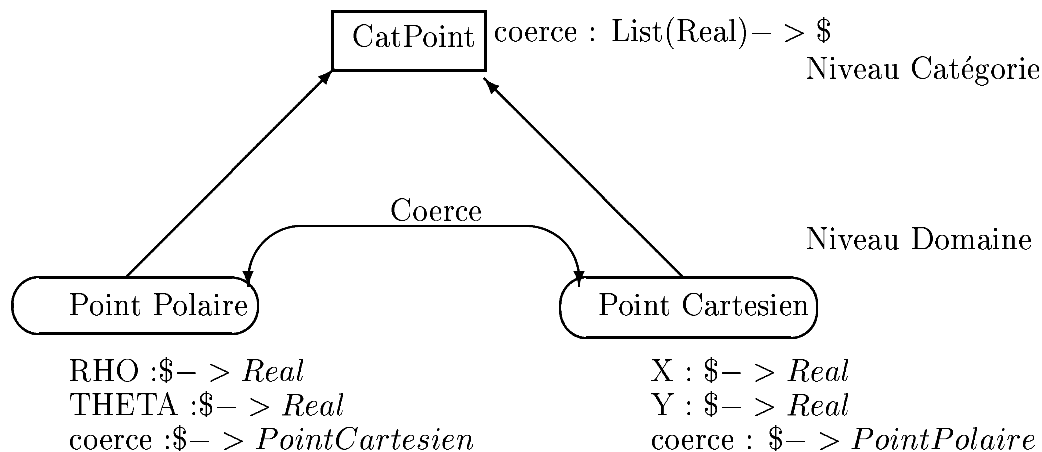


Figure 17: Différentes représentations des points.

```
pointP :PointPolaire
pointP := [1,0]
pointC :PointCartesien := pointP  -- Conversion implicite d'un PointPolaire
                                     en PointCartesien.
X(pointP)                         -- Application d'une fonction des
                                     PointCartesien sur un PointPolaire.
```

Figure 18: Exemple de coercion implicite.

4.2 Polymorphisme Universel.

Le polymorphisme universel fait référence à un ensemble de fonction qui exécute un code identique sur un ensemble de types ayant une structure similaire.

4.2.1 Polymorphisme Paramétrique.

Le polymorphisme paramétrique est introduit lorsqu'on dispose d'outils permettant de ne pas spécifier les types des paramètres utilisés. En Axiom, seul l'interpréteur offre ce style de polymorphisme.

```
id x == x

length [] == 0
length m == 1 + length(rest m)

min(x,y) == if x<y then x else y
```

Figure 19: Exemples de définition implicite.

La Figure 19 montre que la définition implicite de fonction permet d'introduire le polymorphisme paramétrique et que la définition même de la fonction contraint en général le type des paramètres.

A propos de la généricité : Comme nous l'avons vu, Axiom permet de paramétrer un module, on retrouve le concept de généricité tel qu'il est introduit dans des langages comme ADA. Dans le cas des catégories et des domaines on peut parler de type paramétré. La généricité n'est pas une forme de polymorphisme mais plutôt la possibilité de construire des opérateurs agissant sur les types.

4.2.2 Polymorphisme d'Inclusion.

On dit qu'un langage offre le polymorphisme d'inclusion, si une fonction est applicable à une collection de types liés entre eux par une relation d'ordre. C'est pourquoi ce polymorphisme est en général mis en oeuvre par les langages orientés objets. On retrouve à nouveau le parallèle entre la notion de classe des langages orientés objets et les notions de catégories et de domaines, la notion d'héritage introduisant un ordre.

Remarque 4.1 *Le polymorphisme d'inclusion perd sa signification en Axiom. En effet, la notion de coercion permet de se jouer de la notion d'ordre comme le montre la figure 18.*

5 Modèle de compilation pour Axiom.

Dans un premier travail [2], nous avons étudié la possibilité de compiler efficacement des programmes Axiom et de générer des applications portables et autonomes. Le problème principal se trouve être le typage polymorphe, et le passage du modèle objet au modèle fonctionnel. On trouve d'ailleurs des travaux de thèse visant à implémenter différents langages basés sur ces modèles voir [8] et [4].

5.1 Influence du développement objet sur la compilation.

Les propriétés intrasèques des langages objets sont

1. la notion d'objet,
2. la notion d'héritage,
3. et pour structure de contrôle : *l'envoi de messages.*

L'envoi de message L'envoi de message se trouve être le mécanisme qui permet d'introduire le polymorphisme. En effet, on relègue au destinataire le choix du traitement à effectuer. En règle générale, le compilateur essaie de transformer un maximum d'envois de messages en appel fonctionnel, ceci afin d'accélérer l'exécution. L'aspect dynamique des langages orientés objet se trouve au niveau de l'objet : *Qui est exactement le destinataire.*

Nous voyons apparaitre la difficulté engendrée par les différents polymorphismes, nous n'avons que des appels fonctionnels que nous aimerions transformer en envoi de message afin d'expliciter le destinataire. L'aspect dynamique d'Axiom se trouve au niveau des fonctions : *Quel traitement peut correspondre à cette signature.*

Remarque 5.1 *C'est pourquoi, il est expressément conseillé de toujours donner un maximum d'informations à Axiom. On peut alors utiliser un forme équivalente de* l'envoi de messages *pour indiquer à Axiom où chercher la fonction.*

nom_fonction(Ces_Paramètres)$La_Source

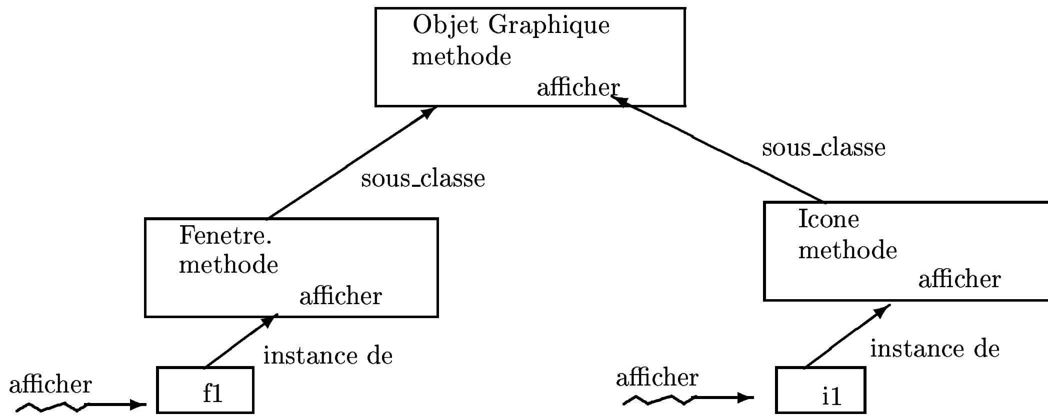


Figure 20: L'envoi de message.

Pour conclure. On se retrouve donc avec un langage qui a un modèle d'exécution fonctionnel et un modèle de développement objet. Qui plus est, le système doit gérer des notions de polymorphisme très évoluées. La tâche du compilateur semble donc très ardue, car il doit transformer un programme objet en programme fonctionnel. Mais malheureusement, en l'état actuel des travaux, Axiom ne permet pas d'effectuer tout le travail à la compilation, il reste encore beaucoup de choses à faire à l'exécution.
L'interpréteur et ses facilités entrainent la mise en place d'outils très novatifs tels que les coercions, la résolution de type, l'inférence de type et la gestion d'une base de données.Tout cela entraine une certaine lenteur à l'exécution mais beaucoup de souplesse quant à l'écriture des programmes.

5.2 Problème engendré.

Les questions qui se posent alors :

- Comment peut-on passer du modèle objet au modèle fonctionnel ?
- Ce passage peut-il se faire sans trop perdre en efficacité ?

5.2.1 Changement de modèle.

Pour l'instant Axiom effectue le passage d'un modèle à l'autre via une phase de compilation qui linéarise le graphe de classe en une collection de fonctions. Cette collection de fonctions est gérée via une base de données.

5.2.2 Qualité du code.

L'efficacité peut être vue comme l'un des points importants lorsqu'on manipule des structures de données conséquentes (matrice, polynôme ..) ce qui est trés fréquent avec un système de calcul formel.
Pour cela, il faut qu'un maximum de travail soit effectué à la compilation. Mais revenons sur le fait qu'il est trés important de pouvoir générer un exécutable autonome et de taille raisonnable. Pour l'instant, Axiom ne tourne que sur les systèmes RS6000 et Sun même si de très nombreux portages sont prévus. Actuellement IBM travaille sur un compilateur pour Axiom qui irait dans ces sens. En effet, on aurait l'occasion de générer du C et/ou du Lisp, mais il est évident qu'un certain nombre de choses ont dû être modifiées dans la structure du langage et dans le modèle d'exécution.

6 Conclusion

Axiom est un langage fonctionnel à développement objet, qui de part sa souplesse permet de modéliser l'univers mathématique de façon satisfaisante. Sa souplesse d'utilisation entraine des mécanismes d'inférence de type et de coercion que l'utilisateur ne maitrise pas toujours, ce qui permet une critique du mode d'exécution. En attendant la sortie d'un compilateur plus efficace et générant un code exécutable autonome et dont les mécanismes mis en défaut soient effectués à la compilation.

Contents

1 Introduction. 2

2 Axiom : langage fonctionnel. 2
2.1 Un monde de fonction. 2
2.2 Notion de type. 3
2.3 Gestion des fonctions. 4
2.4 Coercion et conversion. 5
2.5 Rétraction d'un type. 6
2.6 Résolution de type. 6
2.7 Inférence de type. 7
2.8 Conclusion. 8

3 Axiom : langage à développement objet. 8
3.1 Notion de type et de classe. 9
3.1.1 Les Catégories ou les spécifications. 9
3.1.2 Les Domaines ou les implémentations. 10
3.1.3 Les paquetages ou collections de fonctions. 10
3.1.4 Différence entre Catégorie et Domaine. 11
3.2 L'héritage en question. 11
3.2.1 Héritage simple et multiple. 11
3.2.2 Trois arbres d'héritages différents. 11
3.2.3 Algorithme de parcours des arbres d'héritages. 12
3.2.4 Un petit manque. 12
3.3 Conclusion. 13

4 Différentes formes de polymorphisme. 13
4.1 Polymorphisme Ad hoc. 13
4.1.1 La surcharge. 14
4.1.2 La coercion. 14
4.2 Polymorphisme Universel. 15
4.2.1 Polymorphisme Paramétrique. 15
4.2.2 Polymorphisme d'Inclusion. 16

5 Modèle de compilation pour Axiom. 16
5.1 Influence du développement objet sur la compilation. 16
5.2 Problème engendré. 17
5.2.1 Changement de modèle. 17
5.2.2 Qualité du code. 17

6 Conclusion 18

References

[1] Habib Abdulrab. *De Common Lisp à la programmation objet.* HERMES, 1990.

[2] Jean-Louis Boulanger. Etude de la compilation de scratchpad 2. *Rapport de DEA Universite de lille 1*, Septembre 1991.

[3] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computer Survey*, December 1985.

[4] Stephane Dalmas. *Un langage fonctionnel polymorphe. Application aux probleme logiciels du calcul formel.* Universite de Nice Sophia-Antipolis, Avril 1991.

[5] Roland Ducournau and Michel Habib. La multiplicité de l'héritage dans les langages à objets. *T.S.I Technique et Science Informatiques*, 1989.

[6] IBM. *Newsletter Vol 1 Num 1: Scratchpad 2.* IBM Research, January 1986.

[7] IBM. *Newsletter Vol 1 Num 2: Scratchpad 2.* IBM Research, May 1986.

[8] Stephane Leroy. *Typage polymorphe d'un langage algorithmique.* Universite de Paris 7, Juin 1992.

[9] R.S Sutor R.D Jenks. The type inference and coercion facilities. *ACM*, July 1987.

[10] S.M Watt R.D Jenks, R.S Sutor. Scratchpad 2: an abstract datatype system for mathematical computation. *Computer Science*, November 1986.

[11] S.M Watt R.D Jenks, R.S Sutor. Scratchpad 2 type system : Domains et subdomains. *IBM Internal Report*, 1987.

[12] J.C. Royer. Un modèle pour l'héritage multiple. *BIGRE No 70*, Septembre 1990.