AXIOM,
A Functional Language with Object Oriented Development.

Jean-Louis BOULANGER
Laboratoire d'Informatique Fondamentale de Lille
Universite of Lille 1
59 655 Villeneuve d'Ascq Cedex. FRANCE.

Tel : 20 43 47 19
Fax : 20 43 65 66
mail: boulang@lifl.lifl.fr

December 16, 1993

Abstract

We present in this paper, a study about the computer algebra system Axiom, which gives up many very interesting Software engineering concepts. This language is a functional language with an Object Oriented Development. This feature is very important for modeling the mathematical world (Hierarchy) and provides a running with mathematical sens (All objects are functions). We present many problems of running and development in Axiom. We can note that Axiom [1] is the only system of this category.

Keywords:
Functional language, polymorphic type, simple and multiple inheritance, coercion and type inference, Object Oriented Development.

[1] Axiom (in last SCRATCHPAD II) is a product distributed by the NAG society and developped in past by IBM.

1 Introduction.

It's very important to know the structure of a programming language for a good and efficient use and for understanding the behaviour. Axiom is a programming language with many interesting and new concepts.

It's very important to understand all the mecanismes of development and of running which permit many facilities but which provide running with no control. These possibilities allow that Axiom can modeling the mathematical world with all the rigourous.

The source of our motivations [3], a Computer Algebra System efficiency and that generate of stand alone programs.

2 Axiom: functional language.

Axiom is a functional language, this father is a lisp developped by IBM with addition of all the news ideas that the informatic world generated since fiveteen years. The interested people find a story in [7] and [8]. Axiom provides all tools you can use in the functional language of the KRC and MIRANDA family. The set of notions present here, use notion presented in [10].

2.1 A world of function.

The functional model is based on the manipulation of functions. This fact introduces the respect of two properties:

1. the functions are first class object, they are the only objects manipulable by program,
2. the structure of control is function application.

```
fib 1 == 1                              -- Definition by rewrite rule.
fib 2 == 1
fib n == fib(n-1) + fib(n-2)

PolyLegendre n == n=0 => 1              -- Function definition with guard
                  n=1 => x
                  ((2*n-1)*PolyLegendre(n-1)-(n-1)*PolyLegendre(n-2))/n
-- Examples of functions with use of 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: Some functions in Axiom.

Some examples of functions :

- all the user's functions (see figure 1),
- a type is a function which sends a data abstraction,
- the affectation is a function [2].

```
square n == n*n                  -- A function who compute the square.
-- An approximation of the derivating of a function if dx very small.
derive (f,x,dx) == (f(x+dx)-f(x))/dx
derive (square,10,0.00000001)   -- An example of using.
```

Figure 2: Manipulation of function.

```
 -- Reduction of a list.
reduce(x,f,a) == if x=nil then a
                          else f(first(x),reduce(rest(x),f,a))
-- Definition of Sum and Product of element of a list.
sum(l)      == reduce (l,(x,y)+->x+y,0)   -- With lambda expression.
product(l)  == reduce (l,(x,y)+->x*y,1)
```

Figure 3: *reduce* a function for many use.

In fact, we say that the functions are objects of first order, (see figures 2 and 3).

Axiom allows many forms of polymorphism [3], causing more complexity in the manage of functions.

Remark 2.1 *In presentation of function, we say that the types are functions. In this version of Axiom, type are particular function.*
In fact, I can't

- *overladed a type,*
- *type are value.*

2.2 Notion of type.

```
(1) ->I : Integer := 2
 (1)  2                                          Type: Integer
(2) ->R := 3.14
 (2)  3.14                                       Type: Float
(3) ->T := true
 (3)  true                                       Type: Boolean
(4) ->P := 1+x*y**2+x**3*y
         2    3
 (4)  x y  + x y + 1                             Type: Polynomial Integer
```

Figure 4: Examples of session.

Axiom is a functional language with a strong typing, this implys that all objets have a type (see figure 4). This typing is statical, the type of a variable can not evoluate in time.

[2] It is obvious that the affectation is not implement, but it use the LISP layer.
[3] This question will be treat in Section 4.

All objects have a type imply that you can compute the type of a function, and the types are functions.

Axiom offers all the collection of basic types, such that *Integer*, *Rational*, *Float*, *List*, *Record* and over such that *Streams*. But Axiom is a system with a very interesting typing, because it offers a mathematical hierarchy of types. For example, it offers the monoids, the groups, the rings and all other classical mathematics structures.

```
(5) ->succ n == n+1
                                                    Type: Void
(6) ->succ 3
  Compiling function succ with type PositiveInteger -> PositiveInteger
 (6)  4
                                                    Type: PositiveInteger

(7) ->succ
 (7)  succ n == n + 1
                                                    Type: FunctionCalled succ

(8) ->succ (-3)
  Compiling function succ with type Integer -> Integer
 (8)  - 2
                                                    Type: Integer
```

Figure 5: Example of multiples instanciations.

The construction of program is easy, the user can forget the type of some (all) variables. The system must use the inference of type and associates tools such that type coercion and type resolve.

The declaration of function of figure 5, we show that it's possible to use the operator *succ* with all set of value which offer an operator $+$.

Mathematically, we says that $succ : L - - > L$ with L an abelian monoid.

```
(9) ->compose : (I-> I,PI ->I, PI) -> I
                                                    Type: Void
(10) ->compose (g,f,x) == g(f(x))
                                                    Type: Void

(11) ->compose (x+-> x*2, succ, 4)
  Compiling function compose  with type ((I -> I),(PI -> I), PI) -> I
 (11)  10
                                                    Type: PositiveInteger
```

Figure 6: Example of complex modemap.

The notion of type is to link to the notion of modemap [4].

Definition 2.1 *The modemap is equivalent to the type.*

[4]The modemap notion is define in section 2.3.

2.3 Functions managing.

The number of functions and the differents types of polymorphism provide a great number of functions. The interpreter use a database which manage and choose functions.

Definition 2.2 *An Axiom modemap is an extension of the current notion of modemap.*

$$nom\_function(module, type\_res, type\_par_1, ..., type\_par_n) \ \ if \ predicat$$

you listen this

$$nom\_function \ : (type\_par_1, ..., type\_par_n) -> type\_res \ from \ module \ if \ predicat$$

The modemaps :

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

become :

```
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 7: Example of the standardization of modemaps.

Definition 2.3 *All database modemaps are standardizes. The standardization allows a common structure for the modemaps. The figure 7 purpose an example.*

Definition 2.4 *The standardization uses two menberships operators. The* is *operator is an explicite equality, and the* has *operator is a link of type* is_a.

Research of function. The database may be able to answer to a query defines by a modemap with pattern matching.

The polymorphism gives many cases.

1. one function is correct, the ideal case,
2. many functions validate the modemap, it chooses the first function,
3. if no function validate the modemap, it continues the research, it gives the set of functions with the same name and arity,
4. if no function with this name is found, then generate an error.

2.4 Coerce and convert.

Axiom is an interpreter system. It doesn't provide primitive of input/output but functions which provide the convertion of input form to data abstraction and this in output form.

Definition 2.5 *A* convertion *is an operation which construct an object O_2 of type T_2 with an object O_1 of type T_1, the object O_2 with same properties in T_2 that O_1 in T_1.*

To facilitate the development, the system can run this alteration in an implicit or an explicit way.

Definition 2.6 *It has called* coercion *all functions of conversion that the system can run on implicit way* [5].

Remark 2.2 *The implicit nature of coerces, show that is better to define operations with a algebraic validity or which don't destroy information.*

It offerts also the functions *convert* but it must be used explicitly.

Definition 2.7 *The* :: *operator allows to show a request of conversion. The conversion can use the* coerce *and* convert.

Variable :: Type

The implicit way of functions *coerce* is very important and it's the central tool of the type inference.

L1 : List (Integer) := [1,2,3,4,5,6,7]

The system can use the function
coerce : Integer − > Fraction Integer
but it must be apply this at all elements of the list L_1,
it must be use a function
map : $(T_1 - > T_2, List(T_1)) - > T_2$.

L2 : List (Fraction (Integer)) := [1/1,2/1,3/1,4/1,5/1,6/1,7/1]

Figure 8: Example of coercion par mapping.

Definition 2.8 *we says* coercion by mapping *when the system compose the functions* map *and coerce, for convert a type D(T_1) in D(T_2)* [6].

Remark 2.3 *Axiom is now the only system that provides this general notion of coercion, the other languagues* [7] *offer a notion of coercion between simple types* [8] *and without possibility of user adding.*

It exists two sources for coerce operations:

- the functions defined by program,
- the functions integrated to the system [9].

We presented in section 2.3 the standardization and we see it introduces a predicat that the type of parameter must be validate.

Definition 2.9 *Given a predicat* P *and a set of type* S*, the forcing of type is an operation which convert one or several types of* S *such that* S *satisfy the predicat* P.

Example: Given the predicat *P = *1 has Field*, and suppose that it wants substitute the type *Integer* to the free variable **1*. The *Integer* are not a field but it exists a constructor *QuotientField(.)* which allows to obtain a field with a *Integer*. It is a type forcing.

[5] Implicite, it means that the system can use this when it wants and with no user query.
[6] With D which is a plain constructor.
[7] By language, it understands Computer Algebra system but also programming language as ML, Miranda,...
[8] In general, the coercion as integer in real exist.
[9] The user have no all the control of this fonctions.

2.5 Retract of a type.

Definition 2.10 *The retraction is coercion of the pair* $< V_1, T_1 >$ [10] *in the pair* $< V_2, T_2 >$ *if there exists a degenerate form of* T_1 *to which* V_1 *can be coerced.*

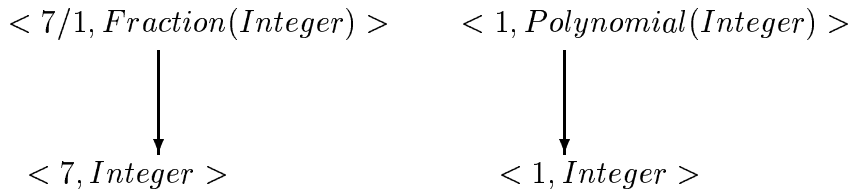

Figure 9: Examples of Retract.

The retraction use two operations that the user will implement for all new datatype :

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

Those operations are defines in the category *RetractTo(R).*

2.6 Resolve of type.

The use of convert operation and composition of function, introduce the notion of a way and type resolving.

Definition 2.11 *A* way *between the types* T_1 *and* T_2*, is defined by a chain of conversion, which process an element of type* T_1 *in type* T_2*.*

Definition 2.12 *Given* two *differents types, it calls* resolve *the process of research of a common type of* two *sources types and the associate ways.*

Proposition 2.1 *Whatever the two parameters :*

- *A type resolving always gives a success, it exists type ANY,*
- *The type resolving is symmetric.*

The type resolve of Axiom is equivalent to the Prolog unification.

2.7 Type Inference.

Axiom use type inference because :

- Axiom is a strong typed system and user can forget variable type,
- Axiom povides many forms of polymorphism,
- Axiom can manipulate types of variables.

The inference of type existe in the ML language (1976), but Axiom offers a more evoluate form.

The interpreter must be then take care of the type calculus of expressions and the management of variables. It uses coerce operations and the informations of the functions database. It sees that the type checking of imperative languages are not enougth, because it use the existing information and in our case the types informations is in program.

[10] The pair $< V, T >$ is listning $< Value, Type >$.

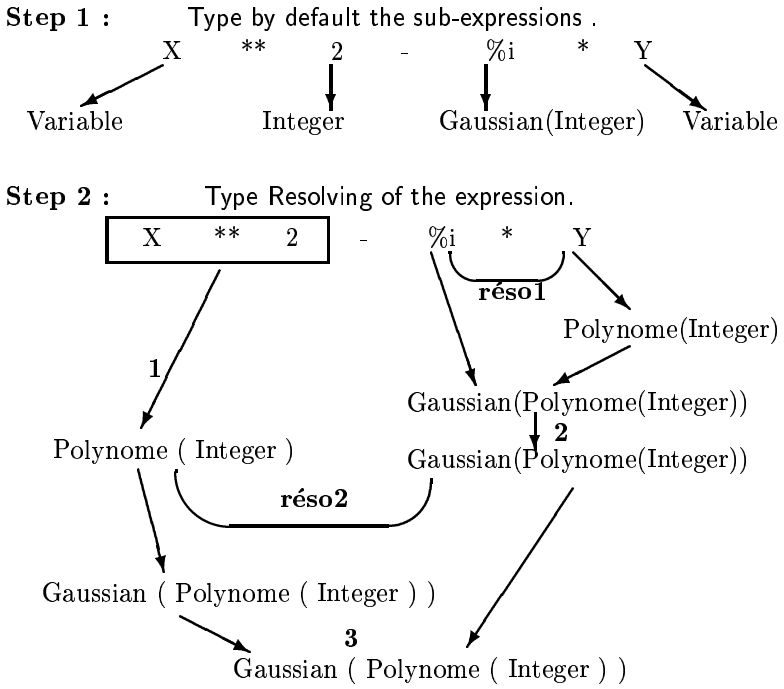

Figure 10: An example of type inference.

Basic rule of type inference :

1. It types by default, all leaf of tree.
2. It types all subtree, by the more internal.
3. It types from left to right.

Explicitation of the figure 10 :

1. It exists an operation ***(Polynome,Polynome(R),Variable,R)* which provide *R=Integer*.
2. For using alone reasonable operation **(??,R,R,R)*, it process type resolving. It finds *R=Gaussian(Polynome(Integer))*.
3. The next step, it process a type resolve because the alone reasonable operation is *-(??,R,R,R)*, and find *R=Gaussian(Polynome(Integer))*.

It remarks, that in type inference, it is possible to read new module and process to the instanciation of generic modules.

The type inference and implicite coercion are important tools of Axiom. It's very important to think at this notion when you work with interpreter.

For example in this case :

- Variables are not declared,
- Type of parameters are different to function modemap,
- Call of polymorph functions :
  - functions with many declarations,
  - functions with different modemaps.

The type calculus can doing choice that the user doesn't do and imply results which appears bad in first view but which are good in the interpreter context. It's very important to give a maximum of information by typing the variables and giving more precisely what is the function you want to use.

Comparison with ML. The fundamental difference reside in the fact that Axiom manipulates and/or transformes the types of variables and that this work are doing for all interpretation of expression. In ML, the type inference is process once for all and the axiom coercion doesn't exist.

2.8 Conclusion.

One of important features of functional languages reside in more free syntax which allows to write programs smaller and in mathematical form. This allows to use the classical demonstration method to prove the programs.

There are two sensitives points :

1. The capacity of system to manipulate itself the types of variables,
2. Some manipulations are in library with no user control.

The mathematic context of type inference and the sense of coerce operation is not formally defined. The type inference is very powerfull but exceeds running time.

3 Axiom : an Object Oriented Developpement.

Axiom offers on development level, all tools for object oriented programming. It offers the notion of class (which is at two level), the notion of simple and multiple inheritance, but the running is process in a functional model [11]. In fact, the notions of category and domain inheriting of Barbara Liskov work's on the language CLU. We want to give in this section a more intuitive approuche of these notions that [12] and [11].

3.1 Notion of type and of class.

3.1.1 The Category or the specifications.

A category is the type specification, in general, it says abstract type [12].

```
)abb category CATS CatSomme

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

Figure 11: A category with implementation by default.

[11] The model of running is presented in section 2

[12] The objects oriented languages introduce, in general, the notion of abstract class.

In the cadre of the formal specification, it introduces some axioms which characterize the action of operations, some of this axioms are constructif and provide the default implementation of operation (see the figure 11).

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

)abb category CP2DC CatPoint2DColored
CatPoint2DColored(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 CatPoint2DMobileColored
CatPoint2DMobileColored(R:AbelianGroup) : Category  ==
   Join(Catpoint2DColored(R),CatPoint2DMobile(R))
```

Figure 12: Introduction to the abstract inheritance of category.

The figure 11 introduces the notion of abbreviation with the operator *)abb*. The figure 12 introduces the notion of simple and multiple inheritance. The multiple inheritance is introduced by the word *Join (...)*, see the category *CatPoint2DMobileColored*.

Axiom allows to define and conditionning the existence of operations in the conception of a category or a domain (see figure 13), this conditionning can be done in the specifications, in implementation and in function definition.

3.1.2 The Domain or implementation.

The notion of domain is linked at the implementation of an abstract class, the figure 14 purpose an implementation of type *CatPoint2D(R)* realise by the domain *Point2D(R)*. A domain will give a representation and a paticular realization of behaviour.

The figure 15 introduces a separation between inheritance of specification and implementation. To limit the conflict of implementation, this implementation inheritance is always a simple inheritance. We see that *Translate* operation use the basic operation of domain in use. It's possible to use this definition how a default definition for the category *CatPoint2DMobile*.

One of problems of developpement in Axiom resides in the conflict of representation. In fact, the structure of an object is introduced by the word *Rep* and a structure can not to be augmented or reduced. All modifications involve the redefinition of associates operations. In the example of the figure 15 for domain *PointColored2D* [13] we must to redefine all the functions which touche at the structure of the object, example for the functions *X(.)* and *Y(.)*. An other solution consist in addition of functions *coerce* which transform *PointColored2D* to *Point2D*. This proves that the inheritance link is not a link *is_a* but that it is an inheritance behavioural.

[13] The inheritance of implementation in the class *PointColored2D* is unecessary in this case.

```
)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 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 13: An quantified Category .

3.1.3 The packages or the function collections.

The packages are a facility to manage the functions on common objects. Their are collections of functions which is parametrizing by types and/or functions.

The package presente in figure 17 allows to construct a generic function [14] which correctly instanciate, generate the factorial function or list recopy.

3.1.4 Difference between Category and Domain.

It's difficult to find a difference between the notion of category and these of domain. But this difference, it's fundamental, the categories describe the notion of type/subtype and the domains notion of the class/subclass.

The languages oriented objects can do two levels of definition (class and metaclass) but in this case a notable difference exists. In fact, a metaclass gives a behaviour for manipulate the class and the set of metaclass is integrated to the existing hierarchy, it doesn't introduce an additional hierarchy.

3.2 Question on inheritance.

3.2.1 Simple and Multiple Inheritance.

We see that Axiom allows to define a new module [15] by creating multiple links with existing structures for the specifications (see figure 12) and simple link for implementations (see figure 14 for domain *PointMobile2D*). We see that exist many types and level of inheritances.

3.2.2 Three Inheritance Tree.

The difference between

[14] The function of example is presented in [1] page 105-107.

[15] A module can be at choice a category, a domain or a package.

```
)abb domain P2D Point2D

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

Figure 14: An implementation of *CatPoint2D* type.

```
)abb domain PM2D PointMobile2D

PointMobile2D(R:AbelianGroup) : Specif == Imple where
   Specif ==> CatPoint2DMobile(R)
   Imple  ==> Point2D(R) add
         Translate(pt,xx,yy) == coerce([X(pt)+xx,Y(pt)+yy])
```

Figure 15: Example of implementation of some point.

1. Category and Domaine,
2. Specification and Implementation.

introduce three differents inheritances trees which are :

- Abstract hierarchy of categorys,
- Default hierarchy (say chaine add),
- Implementation hierarchy (simple inheritance).

3.2.3 Algorithm of lookup for tree inheritance.

We present now the lookup algorithm of inheritances trees that it is presented in [12]. We dont criticize it efficacity or it adequation, the object oriented language litterature provides many works which analyze this subject (see by example [13] and [6]).

The research of operations is done in the order :

1. implementation hierarchy,
2. default hierarchie,
3. abstract hierarchie.

For the multiple inheritance the order is introduced by the entry user, in general we name this priority order.

3.2.4 A small problem.

The notion of inheritance graph is very interesting and allows a good reusability. This principle of reusability is the most important to construct very big software.

```
)abb domain PC2D PointColored2D

PointColored2D(R:AbelianGroup) : Specif == Imple where
 Specif ==> CatPoint2DColored(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)::OutputFoum,
                              hconcat(hconcat(" ",Y(pt)::OutputFoum),
                              hconcat(" ",pt.c::OutputFoum)))
        coerce l      == [l.1,l.2,0]                      -- List are indexed.
        Init(pt,co)   == pt.c:=co
                         pt
        C pt          == pt.c
```

Figure 16: Example of implementation of colored point.

Axiom offers new solutions for reusable (see polymorphism and conditionning) but a small problem resists. The user is obliged to break these trees and to define some classes with same behaviour (see figure 18) or with no sense.

Mathematicaly we define some abstract model (for example Group) with some properties (associativity, commutativity or over) and operations (an internal operation + for the group,..) in instanciation you can considerate all structure with operations which validate the properties.

For example:

1. $(\aleph, +)$ is a monoid.
2. $(\aleph, *)$ is also a monoid.
3. $(M_n, *)$ where $*$ is matrix multiplication is also a monoid.

In fact, Axiom doesn't support the notion of *Perspective* which is presented in [2]. This notion is also called *point of view*, in some languages (see OBJ language).

3.3 Conclusion.

The difficulties of developpement for programs Axiom is provide by

- complexity of inheritance notion,
- the difference between running model and developpement model.

But the mathematic use also the notions of abstraction and inheritance. And Axiom provides a good modeling of mathematic world and all this generality by use of poilymorphism.

Axiom doesn't provide tool for visualize the hierarchies graphs, the state of functions (not define, define or inherited of). But Axiom generates a hypertext help very usefull but too linear. The information of help is provided by user by program annotations.

4 Many forms of polymorphism.

The notion of type is the more importante, but it must be associate at the polymorphism notion who offer all the power at type system. It exists four forms of polymorphism:

```
)package ABST Abstract

Abstract (R:SetCategory ,
          vide?   :R->Boolean,
          Sivide  :R->R,
          compose :(R,R)->R,
          first   :R->R,
          rest    :R->R) : public == private where
   public ==> with
      Abstract: R -> R
   private ==> add
      Abstract(entity) ==
                  if vide?(entity)
                    then Sivide(entity)
                    else compose(first(entity),Abstract(rest(entity)))
```

Figure 17: The package *Abstract*.

$$Polymorphism = \begin{cases} Universel & \begin{cases} Parametric \\ Inclusion \end{cases} \\ Adhoc & \begin{cases} Overloading \\ Coercion \end{cases} \end{cases}$$

This diagram is extracted from [4] and presents the different polymorphisms forms. Axiom provides all polymorphism forms.

4.1 Polymorphism Ad hoc.

The polymorphism Ad hoc is a writing facility, which allows to group in one name some functions with differents processing. In fact, the processing is different and is type depending.

4.1.1 The overloading.

The main form of polymorphism Ad hoc is the overloading such that it is used in ADA. It give the same name at a set of function but the type (of result and parameter) and/or the arity are differents and with function body differents.

The figure 19 give example of this polymorphism. It's true that the overload exist in all languages, but in inacheved form.

4.1.2 The coerce.

In the section 2.4, we introduce the notions of coercions, but we don't say that is polymorphism.

The figure 20 we say that the type point represented by the category *CatPoint* provide some implementations, which are the domains *PointCartesian* and *PointPolar*, this two representations are equivalents. They provide some operations *coerce* which define this equivalence.

The figure 21 using the definitions of the figure 20 and purpose an use of coerce polymorphism.

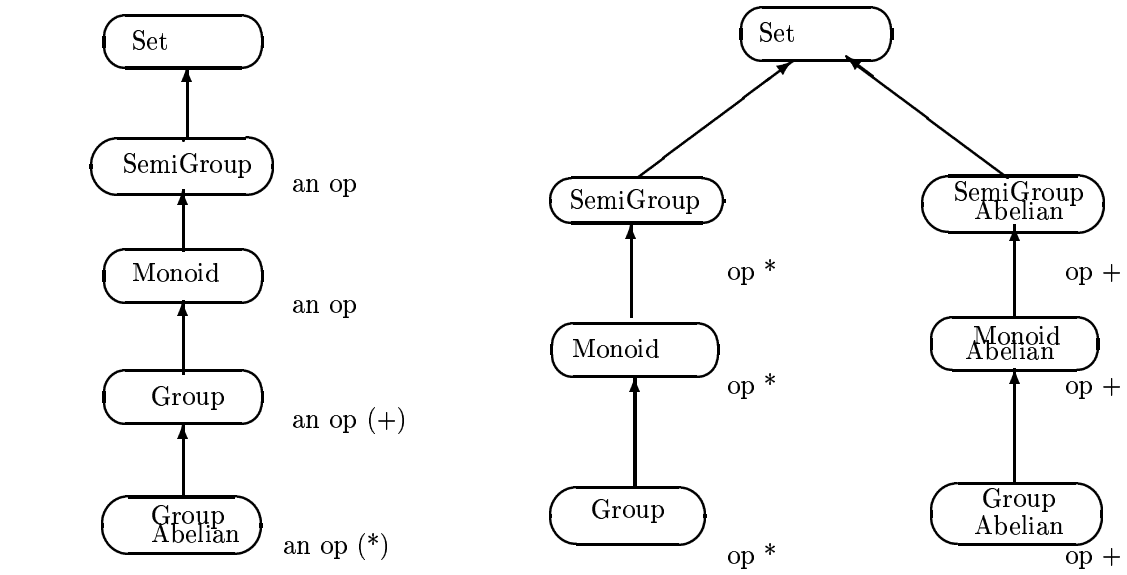

Figure 18: An example of difference between mathematical and implementation.

```
+ : (My_integer) -> My_integer
+ : (My_integer,My_integer) -> My_integer
+ : (My_integer,My_integer,My_integer) -> My_integer
+ : (My_list,My_list) -> My_list
+ : (Matrix,Matrix) -> Matrix
```

Figure 19: Example of overloading.

4.2 Polymorphism Universal.

The universal polymorphism reference to a set of function which runs an identical code, it is a set of types with similar structure.

4.2.1 Polymorphism Parametric.

The polymorphism parametric is introduced when you can parametrize a function. This polymorphism provides definition for *generic functions*.

Figure 22 gives some implicit definitions of functions which introduce the polymorphism parametric. The definition of function constraints the type of parameter.

The genericity : Axiom allows to parametrize a module, we find the wellknown concept of genericity of ADA. In the case of categories and of domains, we can talk of type parametrizing. We say in over section that types are functions, this imply that genericity is parametric polymorphism. But in this version, Axiom can't overloading type and can't construct an operators which works on types.

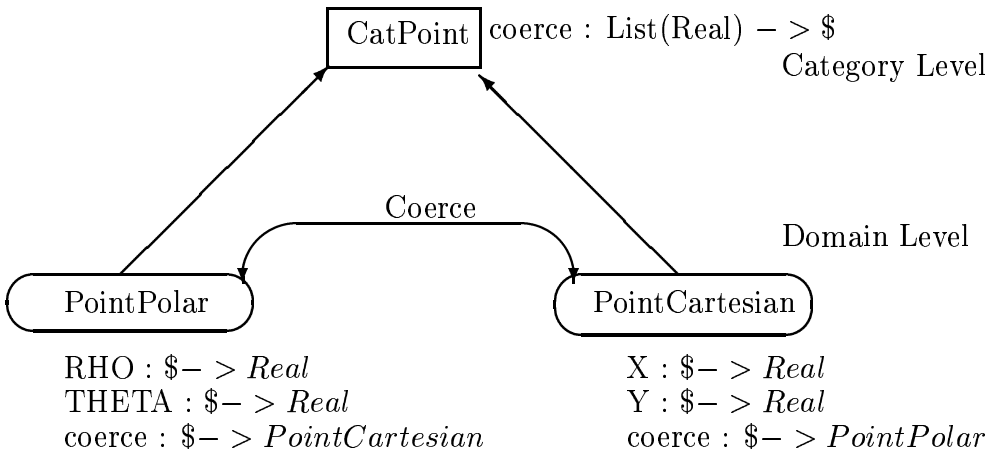

Figure 20: Differents representations of points.

```
pointP :PointPolar
pointP := [1,0]                    -- The power of coercion.

pointC :PointCartesian := pointP   -- Conversion  implicit of a PointPolar
                                      in PointCartesian.

X(pointP)                          -- Application of a function of PointPolar
                                      on PointCartesian .
```

Figure 21: Example of coercion implicit.

4.2.2 Polymorphism of Inclusion.

We say that a language offers the polymorphism of inclusion, if a function is applicable to a collection of types linked by a relation of order. This polymorphism is generaly use in the objects orienteds languages. It finds similarity between the notion of class of object oriented languages and the notions of category and domain, the notion of inheritance introduces an order.

Remark 4.1 *In Axiom, the notion of coercion allows at user to thwart the notion of order, see figure 21.*

5 Model of compile for Axiom.

In a first work [3], we studie the possibility to compile efficiently the Axiom programs. The first problem is the polymorph typing and the transformation of model object in the model functional. The people interested by this domain of work can find many information on the implementation of some languages in [9] and [5].

5.1 Influence of object developpement on the compiler.

The properties of object oriented languages are

1. the notion of object,

```
id x == x

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

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

Figure 22: Examples of implicit definition.

2. the notion of inheritance,
3. and control structure : *send of messages.*

Send of message The message sending is the mechanism which allows the introduction of polymorphism. In fact, it gives to destinataire the choice of processing to do. In general, the compiler transforms a maximum of messages sending in function application, this accelerate the running. The dynamique aspect of object oriented languages is found in the object : *Which is the recipient ?*

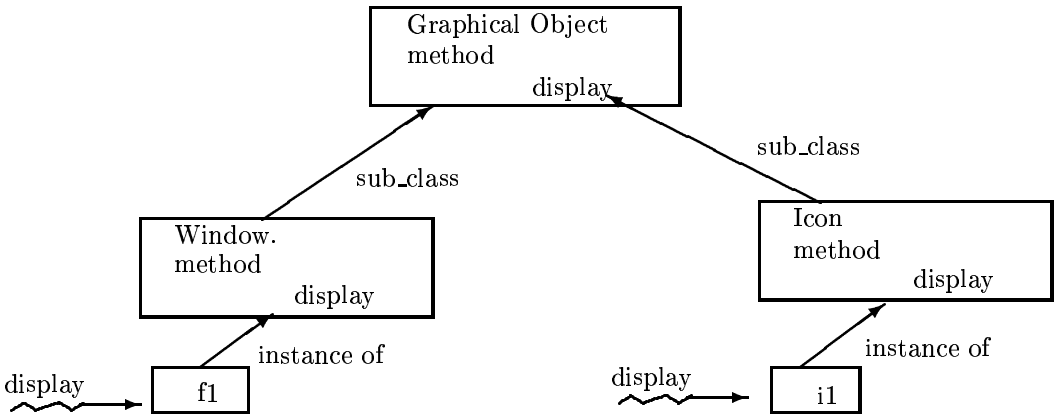

Figure 23: The message sending.

We see to appear the difficulty generated by the differents polymorphisms, we have some function applications and we want to transform it in message sending for explicit the recipient. The dynamic aspect of Axiom is find at level of functions : *What is the processing describeb by the modemap?*

Remark 5.1 *It's why, it is advised to give a maximum of information to Axiom. It can then use a form equivalente of* the messages sending *to show at Axiom where researchs the function.*

nom_function(This_Parameters)$*The_Source*

6 A# or the future of Axiom

IBM have produced a new language, it's called A# and the associate compiler. This language is born of an echec to define the new compiler for Axiom. This syntax have many similarity with Axiom but all sensitive point of Axiom disappear:

- the type transformation (coercion,resolve,..) disappear,
- the type of variable doesn't support conflict,
- the type inference used is similar this of ML.

In this language type are function with same propercies:

- Type can be overloaded,
- Type can be manipulated by function,

Stephen Watt (IBM Yorktown) has presented this language and it compiler at the "Journeys Axiom" at Paris, october 27-28 1993.

This compiler can give :

1. an abstract code (the first pass),
2. a C code (a stand alone program),
3. a Lisp code (for augmented Axiom library or use with Lisp).

7 Conclusion

Axiom is a functional language with object oriented development which modeling the mathematic world. But the type inference of Axiom is very important for flexibility but increase time running.

The efficacing is a very important point when you work with big data structure such Matrix, Polynomial or other which are basic in mathematic. But Axiom provides a flexibility with polymorphism and type inference that over computer algebra system have forgetten.

Contents

List of Figures

References

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

[2] J-L Boulanger and Hasséne Hamroun. Perspective : Un outils pour une representation multiple des objets. *A paraitre*, 1993.

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

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

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

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

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

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

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

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

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

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

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