Object Oriented Method for Axiom.

Jean-Louis Boulanger
Laboratoire d’Informatique Fondamentale de Lille
Université de Lille 1
59 655 Villeneuve d’ascq Cedex
FRANCE
Mail : boulang@lifl.fr
January 1995

Abstract

Axiom ¹ is a very powerful computer algebra system which combines two languages paradigms (functional and OOP). Mathematical world is complex and mathematicien use abstraction to design it. This paper presents some aspects of the object oriented development in Axiom. The axiom programming is based on several new tools for object oriented development, it uses two levels of class and some operations such that coerc, retract or convert which permit the type evolution. These notions introduce the concept of multi-view.

Keywords:
Functional Language, Coercion, Object Oriented Development, Simple and Multiple Inheritance.

1 Introduction.

Axiom is a very powerful Computer Algebra System, mixing two programming methods.

1. Functional programming :
   • All objects manipulated by a program are functions (function are first-class objects),
   • The control structure is function application.

2. Oriented Object Development :

¹Axiom (in past SCRATCHPAD II) is a product distributed by the NAG society and developed in past by IBM.

• All Objects have a type and the types provide a hierarchy,
• The control structure is message sending.

We present in [5] these notions and their implication on programming. But Object Oriented Development is the most interesting part of Axiom, and provides many problems. Because the mixing of two programming paradigms can not protect all typical properties. For example, the message sending does not exist, and user must use function application which is not equivalent.

2 Functional programming.

Many aspects of functional programming can be found in literature see for examples [8], [9] and [10]. For Axiom, we can find some information in [4], [12], [14] and [17].

The next figure describes some Axiom functions in interpreting mode. This definition introduces the parametric polymorphism. These functions are also called generic functions.

```lisp
fib 1 == 1
fib 2 == 1
fib n == fib(n-1) + fib(n-2)
ProduitCart(x,y) ==
    [[a,b] for a in x for b in y]
reduce(x,f,a) ==
    if x=nil
        then a -- Reduction of list.
    else f(first(x),reduce(rest(x),f,a))
```

1
The mixing of two paradigms introduces new notions in interpretation,

- All objects have a type, but user can miss the type,
- User can define transformation operation on type that interpreter can use to define type of object, (this point is more described in [5] and [13].) See 2.1.
- The message sending does not exist, the interpreter must choose the operation to use.

**Definition 2.1** The user can define two type transformation operations:

1. *Coerce*: The coercion is an implicit function that the interpreter can used by the interpreter when necessary.

2. *Convert*: The conversion is an explicit function with explicit use.

3. *Retract*: The basic type can be degenerate to another type.

The Axiom type transformation have similarity with the constructor notion in C++.

```
loop
    read_entry()
    type_eval_entry()
    print_entry()
end loop
```

The type inference in Axiom is more complex than in ML. In fact ML can not support user's converts and provide some basic coerce (example Integer to Real or Character to String). In Axiom, coercion is a kind of polymorphism. The interpreter loop of Axiom defines a step of typing.

![Polymorphism Diagram](image)

This figure is extracted from [6] and presents the different polymorphism forms. Axiom provides all polymorphism forms.

### 3 Oriented Object Development.

#### 3.1 Introduction.

In Axiom, all objects have a type, and all objects are functions, the interesting question is "What is a type?". In this context, a type is the mode map of function, which is an extended notion of map. In type theory, many schools exist, and Axiom uses the next notions:

1. The Categories are the abstract types or type specification,
2. The Domains are class or type implementation,
3. The Packages are functions collections.

Category and Domain are types and are defined by a mode map. In fact, the type definition is equivalent to the function definition. Some interesting problem reside in *What is a Coercion?* (see Section 3.5) and in how to define it.

#### 3.2 Basic Principles.

Mathematician constructs many abstractions, to control the mathematics world. These abstractions are based on two notions:

- The mathematical structure (Monoid, Group, Ring and Field),
- The mathematical object (Real, Complex, Matrix, or \(\mathbb{Z}_2\)).

##### 3.2.1 The paradigm of this programming.

All new Axiom modules:\footnote{Module includes Category, Domain and Package.}

- are created by inheritance, this provides two forms of polymorphism (overloading and inclusion)
- can use genericity by parametrization of module by variables or by functions,
- can be conditioned by the type of parameters,
The first three principles are known in all Object Oriented Languages but the last is a generalization of conversion notion (see definition 2.1). The conversion notion is a very powerful tool.

3.2.2 Category or abstract type.

Mathematical structures are defined by

- The set of operations,
- The set of axioms that operations must verify.

This definition represents the specification that use some languages to generate proof and code (see OBJ, VDM or larch). In Axiom, the set of axioms gives many information:

1. a set of links with other mathematical structures,
2. a set of default implementations,
3. a set of constraints on the behavior of operations.

**Definition 3.1** We call \((S, \circ)\) a semigroup

1. \(\circ\) is closed on \(S\),
2. for all \(x, y, z \in S\), \((x \circ y) \circ z = x \circ (y \circ z)\) \((\circ\ \text{is associative})\).

```plaintext
)abbrev category ABELGRP AbelianGroup
AbelianGroup() : Category ==
  AbelianMonoid with
  "-": $ \rightarrow $
  "-": ($, $) \rightarrow $
  unitsKnown
  add
  x:$ - y:$ == x+(-y)
```

This definition is true for all function that verify the associativity. But in programming, you must give the actual name of function and you can’t change it.

---

**Figure 1:** An extract of basic hierarchy.

**The first problem:** This introduces some differences with mathematics example you must break inheritance tree for define Group and Abelian Group (see figure 1). But this problem is general to programming languages. You can’t give the next definition:

1. \((\mathbb{Z}, +), (\mathbb{Z}, \ast), (\mathbb{Q}, +)\) and \((\mathbb{R}, \ast)\) are all semigroups.
2. \((\mathbb{Z}, -), (\mathbb{Z}, \div)\) and \((\mathbb{Q}, -)\) are not semigroups.

Axiom provides a notion of properties that the operation must verify. They are introduced by operators `associative`, `commutative`, ... These operators are just a mark with no verification.

**Definition 3.2** A ring \(R\) is an object with two operations `+` and `*` that respect

1. \((R, +)\) is an abelian group,
2. \((R, \ast)\) is a semigroup,
3. for all \(a, b, c \in R\), \((a + b) \ast c = a \ast c + b \ast c\) and \(c \ast (a + b) = c \ast a + c \ast b\)

```plaintext
)abbrev RING Ring
Ring() : Category==Join(SemiGrp, AbelianGroup)
)abbrev COMMUT RingCommutativeRing
CommutativeRing() : Category == Ring with
  commutative("*")
```

In the definition of Ring, the name of operations is a convention and I can say \([R, +, \ast]\) is a ring or just \(R\) is a ring.

Figure 2 is an effective construction of some categories and introduces some new notions as

---

\(^3\)The output is managed by the **OutputForm** domain.
a module.

```lisp
)abbrev MODULE Module
Module(R:CommutativeRing):Category ==
   BiModule(R,R) add
   if not(R is $) then x:$+$r:R == $*r
```

In this example, the parameter is used for conditioning the default implementation.

### 3.2.3 Domain or concrete type.

The computational objects are defined by

- A Category which belong to,
- The set of values,
- The set of operations on values,
- The set of links with mathematical structures.

Some mathematical objects are not structure representation but basic object of mathematics (example Real, Complex or Matrix). I present a Domain called Sturm, which provide the respect of mathematical properties of Sturm sequence. You find more information about Sturm sequence in [3]. The principal results are:

- For all polynomial $P \in \mathbb{R}[X]$, we can construct the Sturm sequence $S = (f_1, ..., f_k)$.
- We called Variation of sequence $S$ in the point $x$, the number of changes of sign in $S(x)$, denoted $V(S(x))$.
- We define the number of real roots of polynomial $P$ generator of Sturm sequence $S$ on interval $[a, b]$ by $V(S(a)) - V(S(b))$.
- It exists a Rational bound for real root of polynomial $P$.
- Localization of real root of polynomial $P$ is constructed by computation of variation of Sturm sequence on interval.

This definition gives:

1. Set of Values $\text{List UPolynomial}(R,X)$.

---

- The definition of a category introduces simple and multiple inheritance. We call this inheritance tree Abstract Hierarchy.
- The definition of Monoid introduces the notion of identity $e$. In Axiom, we use property and define constant name.
- The definition of Group introduces the existence of an inverse for all elements of group. This is coded by a definition of the $^{-1}$ operator. This definition introduces an implementation by default for $a^{-1}b$ operator.
- The self reference are provided by the operator $\$.

Ring defined in figure 2 use some basic types as Type, Coercible To and SetCategory which are very small but provide reusability and more generality of code. It’s very important to use basic hierarchy for a good code evolution.

Attributes and parameter can be use for conditioning the behavior or/and the implementation of

---

Figure 2: Inheritance for Ring.
coerce : Polynomial(R) → $,
Bound : $ → Fraction Integer,
Variation : ($,R) → Integer,
NumberOfRoot : ($,R,R) → NonNegative.

3. Set of links = SetCategory

The domain DSTURM belong to the category

\[
\begin{align*}
\text{coerce} &: \quad $ \rightarrow OF \\
\text{coerce} &: \quad UP \rightarrow $ \\
\text{Variation} &: \quad ($,RN) \rightarrow Integer \\
\text{Bound} &: \quad $ \rightarrow RN \\
\text{NbrRootIn} &: \quad ($,RN,RN) \rightarrow Integer \\
\text{NbrRoot} &: \quad $ \rightarrow Integer \\
\text{RootIn} &: \quad ($,RN,RN) \rightarrow List(List(RN)) \\
\text{AllRoot} &: \quad $ \rightarrow List(List(RN)) \\
\end{align*}
\]

If you want construct a new domain DomainSturm which optimize some operation or change behavior you can’t write:

The first principle of paradigm is that all domain belongs to a category, not to a domain. You must construct a Sturm Category which defines the category C.

And the principle of black box doesn’t accept access to inherited representation. The redefinition of coerce operation uses the representation, you reintroduce this.

In this case, all functions of DomainSturm are inherit and coerce are redefine.

3.2.4 Representation of object.

In private part of domain, the word Rep introduce the representation of computational object. This representation can use all domains known by the system. This representation can be recursive. Some examples of very interesting implementation of recursifs types are provide by polynomial commutatif or not (see [11]).

SparseMultivariatePolynomial(R : Ring,
3.2.5 Package.

Axiom provides a third module the Package, which is function collections. Packages define some complementary behavior for a type, some transformations from type A to type B or some user’s functions.

In implementation of Sturm domain, I use annex function such that erem : (UP, UP) -> UP which provides the pseudo- remainder of two univariates polynomials. This functions is defined in general package called PackageDiv listed in next figure.

```
)abbrev package PDIV PackageDIV
PackageDIV(S:Symbol, R:OrderedRing): public ==> private
  public ==> with
    
    erem : (UP,UP) -> UP  --rem of euclidian division.
    erem (p,q) ==
      res:UP := p
      while degree(res)>degree(q) repeat
        deg := (degree(res,s)-degree(q,s))
        pretend NNI -- HOOPS
        res := res=LC(q) - monomial(LC(res),deg)*q
      res
```

The package presented in figure allows the construction of a generic function \(^5\) which correctly instantiated, generates the factorial function or re-copy of a list.

3.3 The world of points.

In this section, I construct a hierarchy for point manipulation.

```
   
   The specification : Next figure introduces an example of a point hierarchy. I use this graph as a specification for my world of points.
   
```

This example is simple but introduces all notions of Object Oriented Programming in Axiom.

Some Categories : To respect the principle of abstraction and the programming method, the behavior of point is define by

```
)abbrev category CP2D CatPoint2D
CatPoint2D(R:AbelianGroup):Category ==
  
  SetCategory with
    coerce : List(R) -> $ ++For construct a point.
    D : ($,$) -> $ ++Distance between 2 points

  In behavior, I can’t define the access to coordinate, because it depends to the representation

\(^5\) The function of example is presented in [1] page 105-107.
(Cartesian (X,Y) or Polar (ρ, θ)). To construct a point, I define a coercion with translate a List(R) into CatPoint2D(R), the coercion have always one parameter. But the next definition is also correct.

```c
)abbrev category CP2DB CatPoint2DBis
CatPoint2DBis(R:AbelianGroup):Category==
    SetCategory with
    Init : (R,R) -> $ D : ($,$) -> R ++Distance between 2 points
```

In this version, the object creation is managed by user and the system can't generate automatically this type of object. This version doesn't preserve the introduced paradigm. This technique is similar to constructor define in C++ language.

```c
)abbrev category CP2DC CatPoint2DColored
CatPoint2DColored(R : AbelianGroup,
COLOR : SetCategory):Category==
    CatPoint2D(R) with
    InitColor : ($,COLOR) -> $ C : $ -> COLOR ++For COLOR access.
```

In this type, I add a function *InitColor* because the type R and COLOR are different and perhaps incompatible.

```c
)abbrev category CP2DM CatPoint2DMobile
CatPoint2DMobile(R:AbelianGroup):Category ==
    CatPoint2D(R) with
    Translate : ($,R,R) ->$
)abbrev category CP2DMC CatPoint2DMobileColored
CatPoint2DMobileColored(R:AbelianGroup,
COLOR:SetCategory):Category==
    Join(CatPoint2DColored(R,COLOR),
    CatPoint2DMobile(R))
```

**Some Domains:** We define some domains which give an implementation of point objects. The domain called *Point2D* describes an implementation of cartesian point in two dimensions space.

```c
)abbrev domain PM2D PointMobile2D
PointMobile2D(R:AbelianGroup) :
    Specif == Impl where
    Specif ==> CatPoint2dMobile(R)
    Impl ==> Point2D(R) add
    Translate(pt,xx,yy)==[x(pt)+xx,Y(pt)+yy]:$
```

For the domain *PointColored2D*, I change the representation because I add the property *Color*.

```c
)abbrev domain PC2D PointColored2D
PointColored2D(R:AbelianGroup,COLOR:SetCategory) :
    Specif == Impl where
    Specif ==> CatPoint2DColored (R,COLOR)
    OF ==> OutputForm
    Impl ==> add
    Rep := Record(x:R,y:R,c:COLOR)
    X pt == pt.x
    Y pt == pt.y
    coerce(pt:$) == hconcat(X(pt)::OF,
          hconcat(hconcat("\",Y(pt)::OF),
          hconcat("\",pt.c::OF)))
    coerce 1 == [1.1,1.2,0]
    -- List are indexed structur.
    InitColor(pt,co) == pt.c:=co
    pt C pt
    pt
```

In axiom, when you redefine a type you must redefine functions associated to the type or inherited methods from another domain. In fact, if you define a type with the constructor *Record* some problems appear, because axiom generates Lisp and in Lisp the Record have different coding according to the number and the length of fields. And Axiom optimizes field access at compile-time.

```c
)abbrev domain PC2DM PointColoredMobile2D
PointColoredMobile2D( R: AbelianGroup, COLOR : SetCategory) :
    Specif == Impl where
    Specif ==> CatPoint2DMobileColored(R,COLOR)
    Impl ==> PointColored2D(R,COLOR) add
    Translate(pt,xx,yy) ==
        InitColor([x(pt)+xx,Y(pt)+yy],C(pt))
```

3.4 **Tree inheritance.**

In section 3.2, we define some notions then we introduce three inheritances trees.
Abstract hierarchy (for categories),
Add hierarchy (for domains),
Implementation hierarchy (simple inheritance).

We present now the look up algorithm of inheritance trees that it is presented in [15]. We don’t criticize it efficiency, the object oriented language literature provides many works which analyze this subject (see by example [16] and [7]).

The research of operations is done in the following order:

1. implementation hierarchy, (simple inheritance)
2. add hierarchy, (multiple inheritance with the respect of the enumeration order)
3. abstract hierarchy. (multiple inheritance)

3.5 Senses of coerce.

3.5.1 Changing the perspective.

\[
\begin{array}{c}
M1 \\
\downarrow \text{Coerce} \\
M2
\end{array}
\]

Some coerce operations define a perspective changing of the object. Perspective describes a view of object and an evolution. An example of a perspective changing is described in figure 3.

\[
\begin{array}{c}
\text{Category Level} \\
\text{Domain Level} \\
\text{Coerce}
\end{array}
\]

- **RHO**: $\$ \rightarrow \text{Real}
- **THETA**: $\$ \rightarrow \text{Real}
- **X**: $\$ \rightarrow \text{Real}
- **Y**: $\$ \rightarrow \text{Real}
- **coerce**: $\$ \rightarrow \text{PointPolar}

**Figure 3**: Different view of Point.

Using the definitions of the figure 3, I purpose some use of coerce polymorphism.

pointC : PointCartesian
pointC := [1,5] — The power of coercion.
--Coercion implicit
PointP : PointPolar := pointC
--Execute a function of PCartesian on PPolar x(pointP)

3.5.2 Projection or Extension.

Some objects are constructed by **Extension of Object Representation** or by **Projection of Object Representation**. The user must define coerce operation to transform the object if possible.

\[
\begin{array}{c}
M1 \\
\downarrow \text{Coerce (Projection)} \\
M2 \quad \text{Coerce (Changing Rep or Extending Rep)}
\end{array}
\]

In Object Oriented Languages, the object is just described by a link is-a but in Axiom, you can transform the representation of an object by extension or projection. You add or remove some properties of object representation. This operation exists because the introduction of word **Rep** in Axiom syntax introduces the possibility to change the object representation (see the domain **PointColored2D**).

\[
\begin{array}{c}
\text{Point2D} \\
X.Y \\
\text{Coerce:2D} \rightarrow 3D \\
\text{Point3D} \\
X.Y.Z \\
\text{Coerce:3D} \rightarrow 2D
\end{array}
\]

It’s very important to define coercions which don’t change the object structure and preserve information. The figure defines two coercions but the projection one is valid but the extending one have many choices for the third coordinates. Another example is done by Integer and Real, all Integer can be coerce in Real but Real are truncate in Integer. In [2], you find a little study of coerce in compiler context. In fact, a true problem resides on the definition of **coerce**, in compiler this notion is linked to the notion of type equivalency.
4 Conclusions

Axiom is a functional language with object oriented development which models the mathematics world. Axiom provides some very interesting tools but the development and debugging of a big application is very difficult. The polymorphism provides a good and an effective reusability of code.

The Object Oriented Programming of Axiom provides two forms of inheritance,

1. the structural inheritance which defines the representation of objects,
2. the behavior inheritance.

Some objects have many representations and one global behavior. This fact introduces the notion of view and use coercion for view evolution. The choice of coercion is very important and can provide some errors. Development of application in Axiom, respect Mathematical structure and general definition, theorem or lemma.

References


