Theoretical Computer Science 104 (1992) 89–107
Elsevier

Abstract specification of structures and methods in symbolic mathematical computation*

C. Limongelli

Dipartimento di Informatica e Sistemistica, Università di Roma, "La Sapienza", Roma, Italy

M. Temperini

Dipartimento di Informatica e Sistemistica, Università di Roma, "La Sapienza", Roma, Italy and IASI, CNR Viale Manzoni 30, 00185 Roma, Italy

Abstract

Limongelli, C. and M. Temperini, Abstract specification of structures and methods in symbolic mathematical computation, Theoretical Computer Science 104 (1992) 89–107.

This paper describes a methodology based on the object-oriented programming paradigm, to support the design and implementation of a symbolic computation system. The requirements of the system are related to the specification and treatment of mathematical structures. This treatment is considered from both the numerical and the symbolic points of view. The resulting programming system should be able to support the formal definition of mathematical data structures and methods at their highest level of abstraction, to perform computations on instances created from such definitions, and to handle abstract data structures through the manipulation of their logical properties. Particular consideration is given to the correctness aspects. Some examples of convenient application of the proposed design methodology are presented.

1. Introduction

The aim of this work is to present an object-oriented approach to support the design and implementation of a new generation symbolic computation system.

The requirements of such a system are related to the specification and treatment of mathematical objects, e.g. functions, relations, logical formulae and general data

Correspondence to: C. Limongelli, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy. Email addresses: limongelli@iasi.rm.cnr.it and temperini@iasi.rm.cnr.it.

*This work has been partially supported by MURST under the projects: "Calcolo Algebrico, Metodi e Strumenti per l'Elaborazione non Numerica"; and by CNR under the project: "Sistemi Informatici e Calcolo Parallelo".

structures. This treatment should be considered from both the numerical and the symbolic points of view. In this context of symbolic computation, logical computation such as the verification and deduction of properties of given data structures is also included.

The resulting programming system should be able to:
- support the formal definition of abstract mathematical data structures,
- perform computations and deductions on instances of actual mathematical data structures created from given definitions.

Some symbolic computation systems, such as Reduce [5] and Macsyma [17], have been used successfully in a variety of applications, but they are basically built to manipulate a stated number of computational structures, by a stated set of algorithms; they can be viewed as a collection of algorithms on these data structures. The addition of new structures, and of new algorithms, may require extensive modification of the whole system. In these systems most of the existing code may not be reusable. Moreover serious correctness problems depend on the methodological inadequacy of the system design and organization [13].

In order to face these correctness problems, a different approach has been followed by using data abstraction techniques. An example of such an approach is found in the design of the Scratchpad II system [8], that allows a very high level of data structure specification, by providing the concepts of *category* and *computational domain.* Through these concepts, the definition of a variety of operations over a set of related data structures is supported in a fully modular way.

This approach has been considered as a first step towards the design and development of a fully object-oriented system for the manipulation of mathematical data structures. An example of an object-oriented approach to algebra system design is presented in [1]. Moreover, the aim has been to enlarge the usual features of symbolic computation systems by the manipulation of logical properties of the data structures.

To obtain these goals, a system should exploit the following basic characteristics:
- abstract data type management;
- parametric and functional polymorphism;
- inheritance mechanism;
- manipulation of logically expressed properties of data structures.

These characteristics can be used to point out the relations among different data types, to guarantee the correctness of the entire system and to increase the code reusability and maintainability. Moreover, in the proposed approach, the above characteristics can support the treatment of data structures through the manipulation of logical properties. These properties can be used both for the definition of the structures and for their direct use.

In this paper the methodological aspects related to the abstract design of mathematical structures and methods are covered, while the more specific aspects related to automated deduction can be found in [6, 3].

The proposed methodology is founded on the evolution of techniques for data abstraction [4, 16]. The data type system is designed using inheritance by exploiting

the existing relations and by pointing out the dependencies and commonalities among the definitions of data structures [15].

On this basis the Object-Oriented Programming (OOP from now) paradigm ([11, 14, 18]) is considered here for the deep consistency between the "way of thinking" induced, and the "style of working" in mathematical objects specification, as given in formal algebra. How this paradigm naturally fits into the activity of defining and implementing abstract data structures, will be recognized and stressed.

A special taxonomy of formal structures will be defined in algebraic terms and then reconstructed in programming language terms by using *classification* and *inheritance* techniques.

In such a classification the definition of the methods operating on mathematical structures plays a fundamental role: algorithms for the manipulation of abstract data structures can be specified in the definition of such structures and polymorphic methods defined over general structures may be applied to less general ones. The OOP approach allows the location of the methods at their highest level of abstraction, according to their related data structures.

It will be shown that this approach by abstraction acts at two different levels. At the first level, methods characterize attributes of the data structure (e.g. addition and multiplication in a ring structure). The second level corresponds to methods which are not necessary for the complete formal specification of a data structure, while being necessary to enrich the structures in order to perform computations (e.g., the Hensel lifting method, see Section 4.2). These methods are defined at the highest level of abstraction, by enriching the most abstract of the structures on which they can operate.

The next section will present a classification of formal data structures, made in algebraic specification terms, in order to state the requirements that a system should satisfy. Section 3 shows how the main programming features of the OOP paradigm can support the definition and the correct manipulation of mathematical objects, as previously classified. In Section 4 examples of convenient application of the proposed design methodology are presented.

2. Algebraic structures

The specification of an algebraic structure is defined by means of the notion of signature. A signature Σ consists of a triple $\langle S, O, P\rangle$: S is a disjoint union of sets (sorts) on which the corresponding algebra is defined; O is a proper set of operators (function symbols) such that each operator is associated to a mapping type $s_1 \ldots s_k \rightarrow s$ where $s_1 \ldots s_k$ and s belong to S; P is a set of properties defining the relationships among operators, expressed by a logic formalism, e.g. by formulae of FOL (as in [19]) or expressions of algorithmic logic (as in [9]).

The choice of an axiomatic specification mechanism characterizes this paper, and it will satisfy the design requirements of a symbolic computation system, as specified in the introduction.

In this section a classification of the algebraic structures is proposed, defining three planes for their static definition. This subdivision will point out the different specification requirements of structures laying on different planes, leading to a higher level of correctness in their treatment. In the sequel the term symbolic computation will be used for any computation involving the manipulation of properties of a structure, regardless of the abstraction level of its definition. The term numeric computation will refer to computation that acts over structures having sorts completely specified. Moreover, it will be shown how the OOP methodology naturally fits into the specification scheme discussed above.

2.1. Abstract structures

The definition of algebraic structures is carried out by a given scheme supporting the declaration of sorts, operators and their related axioms. The first step towards the definition of such a scheme should include mechanisms for the classification of the algebraic structures that find place at the highest level of abstraction as stated in classical algebra. In this way a hierarchy of structures (see Fig. 1) is established. The elements of this hierarchy will be named *abstract structures.*

The hierarchy specializes itself by adding new properties and operations, starting from a very general structure (e.g. semigroup). From now on the single arrows in the figures represent inheritance relations: the direction of an arrow is meant to be pointing from the "inherited" class (*parent*) towards the "inheriting" one (*successor*).

At a general specification level, as in classical algebra, sorts are not specified, while only operations and axioms related to the presentation of an algebraic structure are defined. Abstract structures cannot be employed for numeric computations: only symbolic computations can be carried out on them. In fact numeric computations are performed only when the sorts of the involved algebraic structures are completely specified.

In Fig. 2 the specification of the algebraic abstract structures of Fig. 1 is shown. In this specification the possibility to obtain the definition of a structure taking sorts, operators and properties **from** another definition, possibly by **redefining** some denotations, is exploited. This characteristic, which is classical in algebra, will directly match with the inheritance OOP features as shown in Section 3.

It should be noted how in this specification the use of the inheritance mechanism by means of the "**from**" construct makes it possible to obtain shorter but equally expressive definitions.

2.2. Parametric structures

In order to actually use the specified abstract structures, the need arises for more specialized properties and operations. A first step towards this objective is the

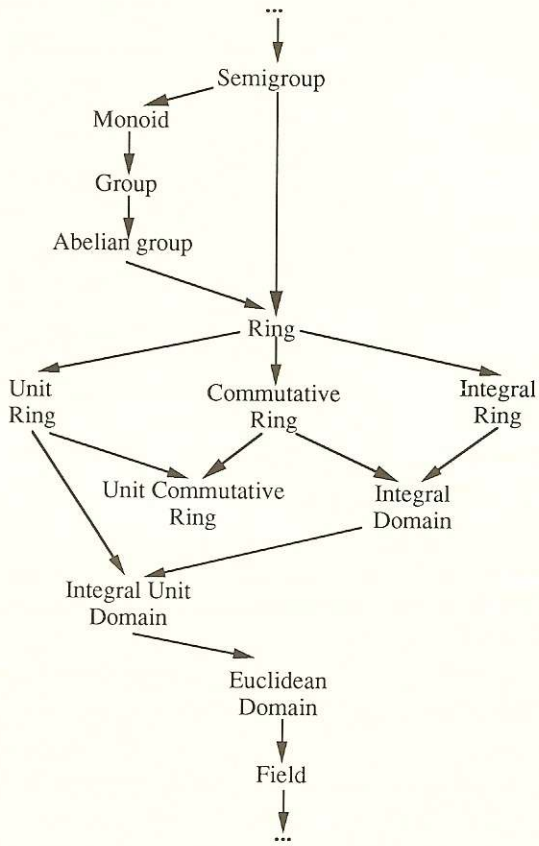

Fig. 1. Classification of algebraic abstract structures.

specification of an intermediate level of specialization, where the requirements are essentially the following:

- the specialization of the sorts which is accomplished by the specification of sorts containing parameters, i.e. the complete specification of sorts depends on the specification of other sorts;
- the complete semantic definition of the operations, i.e. the methods which are characteristic of these structures.

These structures will be named *parametric structures.* Matrices, polynomials and formal power series are examples of parametric structures. The correspondence with the related abstract structures is shown in Fig. 3. Here, for the sake of clarity, a distinction among abstract and parametric structures is outlined by placing these structures into different planes: an *abstract plane* and a *parametric plane*, respectively.

Let us consider the case of the matrix structure (Fig. 4). A general square matrix of order n (n is a natural number), is an element of the set

$$\mathrm{MAT} = \{\{a_{i,j}\}_{i,j \in I_n},\ a_{i,j} \in \mathbb{S}\},$$

Semigroup	
sorts	$\$$
operations	$op: \$ \times \$ \to \$$
properties	$\forall a, b, c \in \$, (a\ op\ b)\ op\ c = a\ op\ (b\ op\ c)$
Monoid	
from	Semigroup
operations	$1: \ \to \$$
properties	$\forall a \in \$, 1\ op\ a = a\ op\ 1 = a$
Group	
from	Monoid
properties	$\forall a \in \$, \exists a^{-1} \in \$: a\ op\ a^{-1} = a^{-1}\ op\ a = 1$
Abelian Group	
from	Group
properties	$\forall a, b \in \$, a\ op\ b = b\ op\ a$
Ring	
from	Abelian Group **redefining** op **as** $+$, 1 **as** 0, a^{-1} **as** $-a$
from	Semigroup **redefining** op **as** $*$
properties	$\forall a, b, c \in \$, a * (b + c) = a * c, (a + b) * c = a * c + a * b$
Unit Ring	
from	Ring
operations	$1: \ \to \$$
properties	$\forall a \in \$, a * 1 = 1 * a = a$
Commutative Ring	
from	Ring
properties	$\forall a, b \in \$, a * b = b * a$
Commutative Unit Ring	
from	Unit Ring
from	Commutative Ring
Integral Ring	
from	Ring
properties	$\forall a, b \in \$, a * b = b * a = 0 \Leftrightarrow a = 0 \vee b = 0$
Integral domain	
from	Commutative Unit Ring
from	Integral Ring
Euclidean Domain	
from	Integral Ring
operations	$\mathrm{div}: \$ \times \$ \to \$, \mathrm{mod}: \$ \times \$ \to \$$
properties	$\forall a, b, c \in \$, \exists c, r \in \$: a = c * b + r \Rightarrow (a\ \mathrm{div}\ b = c) \wedge (a\ \mathrm{mod}\ b = r)$
Field	
from	Commutative Unit Ring
from	Integral Domain
properties	$\forall a \in \$, a \neq 0\ \exists a^{-1}: a * a^{-1} = a^{-1} * a = 1$

Fig. 2. Abstract structure specification.

where

$$I_n = \{i \in \mathrm{NAT} : 1 \leqslant i \leqslant n\}$$

and NAT is the sort of natural numbers.

The use of the **from** construct with the specification "**on** MAT" means that the operations ($+$ and $*$), inherited from the Ring structure, are applied to the parametric

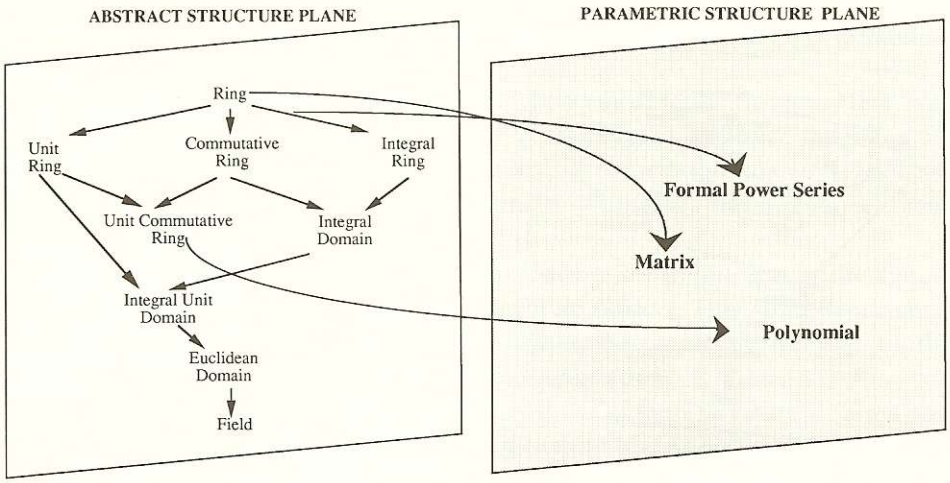

Fig. 3. Example of parametric structure.

Matrix

sorts	NAT, I_n, MAT, \$: Ring
operations	**from** Ring **on** MAT **redefining** + as $+_{\mathrm{MAT}}$ and $*$ as $*_{\mathrm{MAT}}$
	$\mathrm{PUT}: \mathrm{MAT} \times \mathrm{NAT} \times \mathrm{NAT} \times \$ \rightarrow \mathrm{MAT}$
	$\mathrm{GET}: \mathrm{MAT} \times \mathrm{NAT} \times \mathrm{NAT} \rightarrow \$$
properties	$\forall i, j \in I_n, \mathrm{GET}(0_{\mathrm{MAT}}, i, j) = 0_{\$}$;
	$\forall M \in \mathrm{MAT}, \forall i, j \in I_n, \forall e \in \$, \mathrm{GET}(\mathrm{PUT}(M, i, j, e)\ i, j) = e$;
	$\forall M_1, M_2 \in \mathrm{MAT}, M_1 +_{\mathrm{MAT}} M_2 = \{\mathrm{GET}(M_1, i, j) +_{\$} \mathrm{GET}(M_2, i, j)\}_{i,j \in I_n}$
	$\forall M_1, M_2 \in \mathrm{MAT}, M_1 *_{\mathrm{MAT}} M_2 = \{\mathrm{GET}(M_1, i, 1) *_{\$} \mathrm{GET}(M_2, 1, j) +_{\$} \cdots +_{\$}$
	$+_{\$} \mathrm{GET}(M_1, i, n) *_{\$} \mathrm{GET}(M_2, n, j)\}_{i,j \in I_n}$

Fig. 4. Matrix structure specification.

sort MAT: they will be denoted respectively as $+_{\mathrm{MAT}}$ and $*_{\mathrm{MAT}}$. In the definition of Fig. 4, just a partial specification of matrix coefficients is given by the "parameter sort" \$ (so the coefficients are elements of a Ring structure in which 0_{MAT} is the null element for $+_{\mathrm{MAT}}$; $0_{\$}$ represents the null element of \$).

2.3. Ground structures

When sorts are completely specified, the resulting structure allows a direct variable instantiation (in this case the structure will be named *ground*). Therefore it can be used both for symbolic and numeric computations. Moreover, it can be noted that new computational methods and properties can be added into the definition of a ground structure, in order to reach a higher level of efficiency in the implementation, both for computation and deduction, see Fig. 5.

The characteristics of the structures defined in Fig. 5 are summarized as follows.

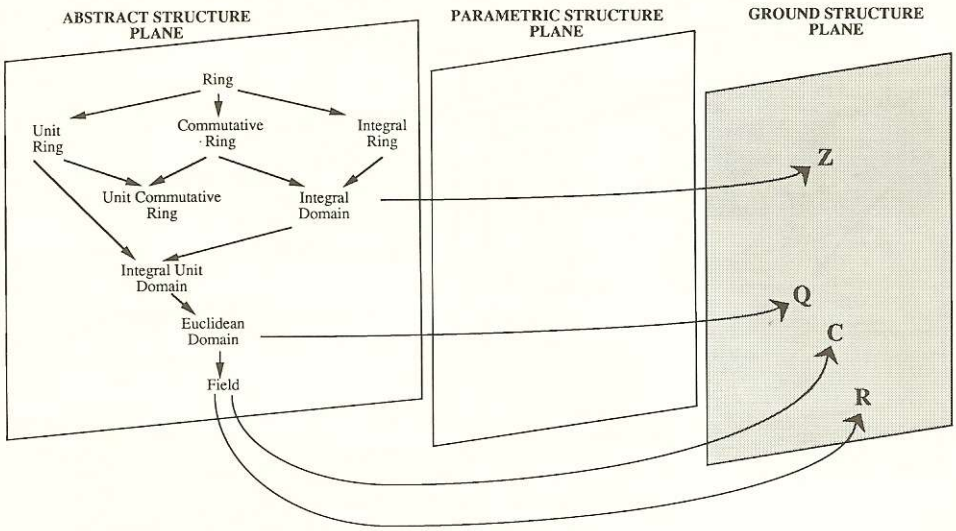

Fig. 5. Ground structure plane.

- *Abstract structures*: classical algebraic structures described by inherited properties. Sorts are not specified; only symbolic computations can be performed.
- *Parametric structures*: enrich the definition of abstract structures by partial (parameterized) sorts, and by additional operations and properties.

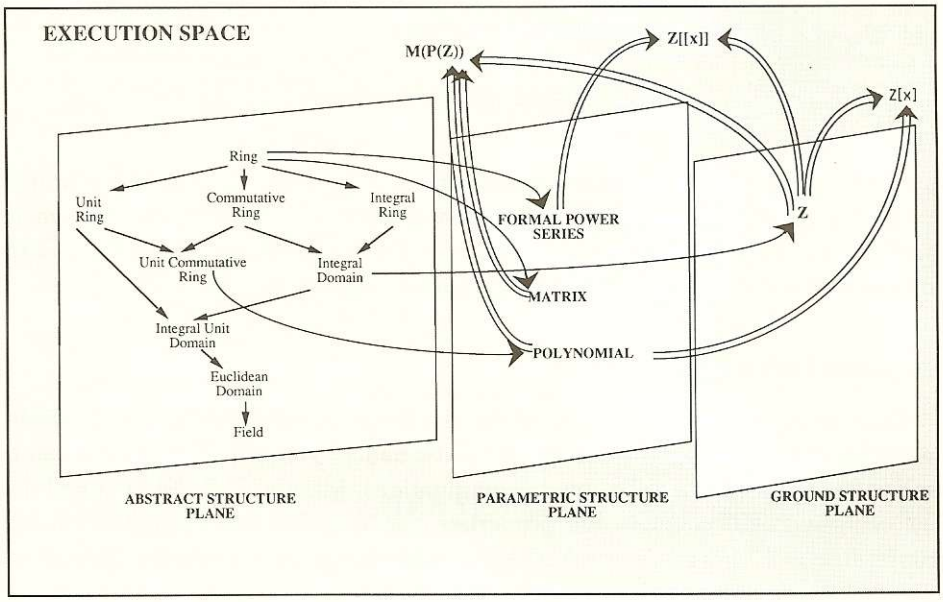

Fig. 6. The execution space.

- *Ground structures*: completely specified; both symbolic and numeric computations are allowed.

The *definition space* is defined by the three previously described planes. Here two different levels of hierarchy can be imagined: a first level is limited to the plane of abstract structures, and defines their hierarchy; a second interplanar level acts respectively between the abstract and parametric structure planes and between abstract and ground structure planes.

Out of the definition planes the *execution space* takes place; that is the space where the structures are instantiated into algebraic objects. This space represents the connection between a single object and its generating structure (see Fig. 6). In this figure the double arrows stand for object instantiation.

For example, an integer number is joined with its ground structure $\mathbb{Z}$, a matrix of integers is joined both with the parametric structure Matrix and ground structure $\mathbb{Z}$. A matrix of integer polynomials is joined with the structure $\mathbb{M}(\mathbb{P}(\mathbb{Z}))$.

It must be noted that an abstract structure is never instantiated. The chain of inheritance acts only on the abstract structure plane. On the other hand the elements that belong respectively to the parametric structure plane or to the ground structure plane, are not related between themselves. Their correlations are established by transversal inheritance from the abstract plane to the parametric and ground planes.

3. Class inheritance and algebraic structures

The main features of OOP can be expressed by the following concept [9]:

Object + Class + Inheritance.

Here the three most important abstraction mechanisms are comprised: classification, generalization and aggregation.

An object can be considered as a dynamic entity of a program. It is an instance of an abstract data type.

The execution of an object-oriented program may be seen as the interaction among all the objects which have been instantiated. Moreover, an object-oriented program can be seen as the description of such a cooperation activity. In this description an essential aspect is the definition of the model for the object instantiation.

A class is the static counterpart of an object, and corresponds to such a model. Each object created during the execution belongs to a stated class, which in turn determines the characteristics of that instance.

At design level a class is a formal description of its objects, while at the implementation level it specifies a template for instance building, in terms of object internal state and behaviour.

Class inheritance can be viewed, at implementation level, as a way for system building, while at a higher level it is the capability of expressing the relationships

between classes. Class inheritance allows the derivation of a new class from one or several existing ones. That is, objects modelled by a class can have properties in common with the objects modelled by other classes.

3.1. Strict versus nonstrict inheritance

As a matter of fact, two main relations do exist between ancestor and descendant classes, namely "is-a" or "is-like", according to strict or nonstrict inheritance, respectively, cf. [18].

In a strict inheritance framework the class mechanism can powerfully support the use of ADT in the manipulation of mathematical objects. It is also important to stress how this hierarchical organization principle offers a relevant general method for code reuse.

In Fig. 7 each node inherits properties while adding some others. Straight lines have the usual meaning of strict inheritance, while dashed lines indicate non-strict inheritance. The properties inherited by a parent structure are indicated between brackets; the added properties are indicated inside the node, out of the brackets; expressions like $u =$ "0" or $op =$ "*", indicate redefinition of attributes.

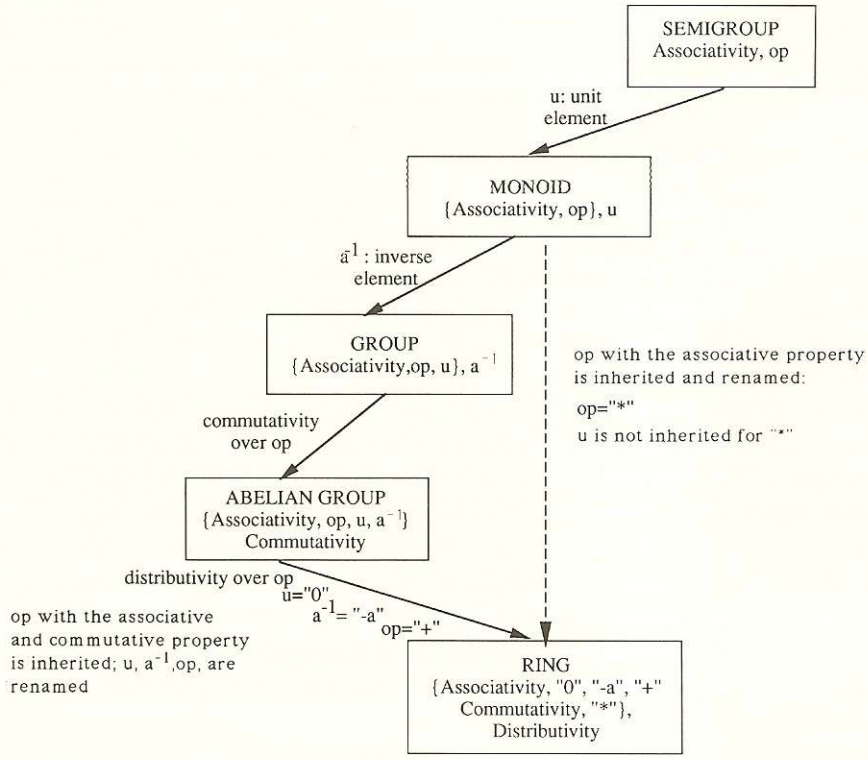

Fig. 7. Nonstrict inheritance among algebraic structures.

For instance, Group is obtained by strict inheritance from Monoid (i.e. inheriting all the properties of Monoid) by adding a new property, namely the existence of the inverse of each element. Ring can be obtained by multiple and nonstrict inheritance from Monoid and Abelian Group. All the attributes of Abelian Group are inherited, while the inheritance from Monoid is incomplete because of the deletion of unit element for "*". Let us note that this is an enforcement chosen in order to show the use of nonstrict inheritance. Ring could be also obtained following a more proper algebraic pattern, via multiple strict inheritance, by aggregating the Abelian Group and Semigroup properties, as it is shown in Fig. 8.

Type hierarchy is a powerful tool for incremental design: it is a way to express similarities amongst various ADTs and a basis for implementing a simple and safe type system. However what strict inheritance can preserve in terms of correctness of the data system, can be lost in terms of flexibility of the software production. Indeed, strict inheritance is not a very flexible mechanism to report ADT correlations. Moreover, if these relations are discovered only after the ADT hierarchy has been completely specified, then it may be necessary to reconsider the type hierarchy and the implementation itself [10]. On the other hand, relations among mathematical

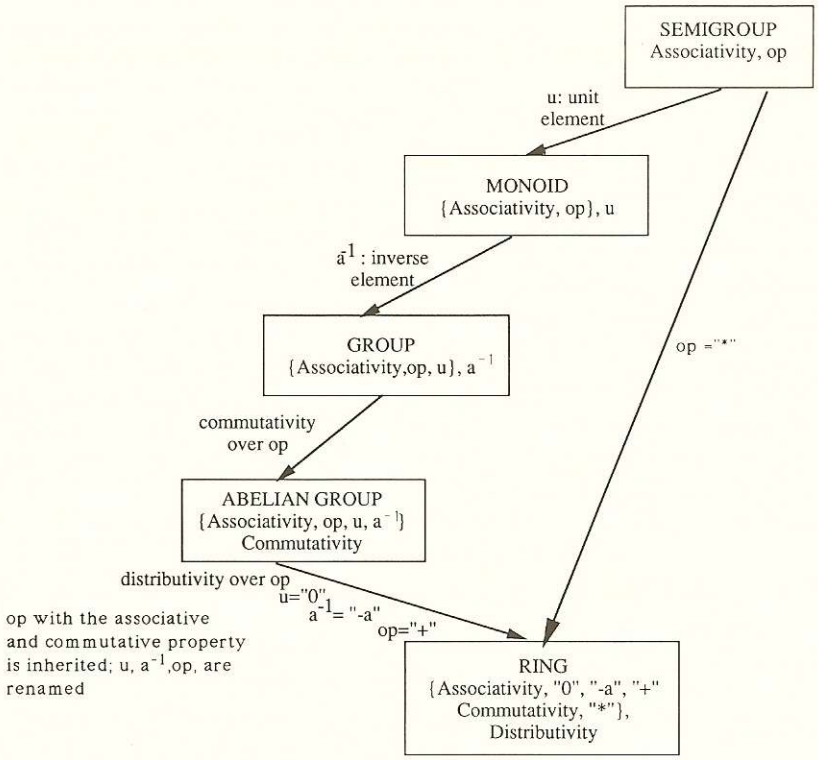

Fig. 8. Strict inheritance among algebraic structures.

objects are often known before their implementation, and new classes are always specializations of the existent ones. Therefore system rigidity induced by strict inheritance is mainly a way to preserve correctness.

To conclude, aggregation mechanism of abstraction seems very useful for the purposes of this work, provided that it is achieved by a *multiple strict inheritance* mechanism.

3.2. Towards a higher level of correctness

The characteristics of correctness and expressive power of the OOP paradigm, can be fruitfully exploited in supporting the definition of algebraic structures. One of the reasons that this is possible, is that a correspondence can be established between the three different types of algebraic structure (as described in Section 2), and the program entities employed for the data abstraction in the OOP. In fact the specification of each abstract structure finds correspondence in a suitable kind of class, by the definition of data structures provided according to classification and inheritance concepts.

Three different kinds of class can be outlined enabling the definition of the three different specification planes.

The concept of abstract class in OOP is well stated. An *abstract class* is a program unit; it defines a given kind of data type in which the common properties of a collection of classes are generalized. Each abstract class may be a particular specialization of other abstract classes, deriving from their definition. Moreover, a defined abstract class can be the ancestor of other collections of definitions.

Using the definition of abstract classes, all the algebraic structures specified in the abstract algebraic plane can be implemented.

An algebraic parametric structure corresponds again to the definition of a class that inherits the characteristics of an abstract class, but in this class the data structure is partially specified and a collection of polymorphic methods that handle the objects of that class can be given. This kind of program unit is defined *parametric class.* Each defined parametric class is related to a correspondent element in the parametric structure plane; its attributes are related to sorts, operators and properties of a correspondent parametric algebraic structure.

In a class specification, data types definition can occur to establish the form of the related variables (instances of that class).

In the abstract class this definition is usually absent: actually abstract classes have not to be instantiated.

In a parametric class just a partial data structure definition is given, which depends on the actual definition of the other data structures. For example, referring to Fig. 4, in the definition of Matrix a partial specification is given for the sort $, stating that the elements of the matrix must belong to a Ring structure ($: Ring).

A class which has completely specified data structures and methods is called a *ground class* and corresponds to the definition of ground structure.

From the point of view of variable instantiation, the concepts exposed above can be summarized as follows:

- *abstract class*: never instantiated, because it has been created in order to generalize a collection of more specialized classes;
- *parametric class*: never directly instantiated. A complete variable instantiation is made only once the instantiation of its parameter structure is accomplished. In Section 4.1 it will be shown how this scheme is the dynamic solution to the problem of parametric structure definition and instantiation, which is statically solved by genericity, through parameterized classes.
- *ground class*: can be directly instantiated, for example by a **new** or a **create** construct, as is usual in OOP languages.

Looking at the static definition of abstract structures, one must notice that the elements of both parametric and ground plane inherit from the structures of the abstract plane (see Figs. 3 and 5). There are no static links between parametric and ground plane: the conformity relations between them are determined by inheritance relations that connect these planes to the abstract one.

Looking at the dynamic situation, the instantiation operations start from the parametric and ground planes, having effects on the execution space (see Fig. 6). These operations could also be, in some sense, recursive as in the case of the definition of the matrix of matrices of . . ..

4. Structure and method abstraction: examples

4.1. Structure abstraction

The following example shows the correspondence between the logical model of data (given by ADT specifications) and the physical one (given by class hierarchy).

Suppose the representation of some algebraic elementary data (e.g. matrices, polynomials, integers) and elementary operations on their instances (e.g. addition, multiplication) is required.

Observe that a matrix or a polynomial can be viewed as composite data built upon some ground or parametric structures. Fig. 9 illustrates the situation in terms of abstract, parametric and ground structures.

Then, a system for this composite data has for instance to deal (in a dynamic way) with matrices having integer polynomials as components, or with polynomials having matrices of integers as coefficients.

The implementation of these data structures can exploit the polymorphism induced by the hierarchy of the data type definition. In Fig. 10 a partial implementation is provided for the abstract class Ring, for the parametric class Matrix and for the ground class Integer-Number.

The used syntax refers to the LOGLAN programming language [2] and should be of simple comprehension, once it is noted that, in LOGLAN, the class structure

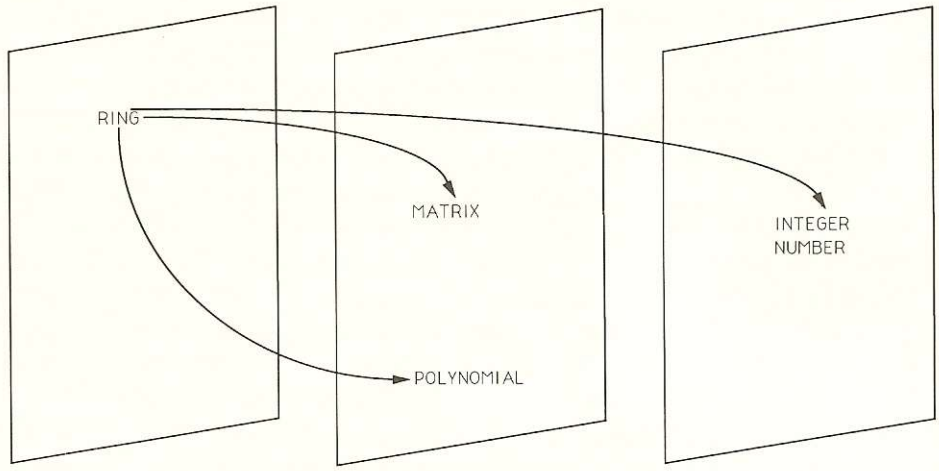

Fig. 9. Ring, Matrix, Polynomial, Integer-number.

is composed by a declaration part and by an execution part, in which the instructions necessary for the class instantiation are implemented.

When a suitable management module is furnished for such a data system, by the exploitation of the parametric polymorphism, the requirements for a dynamic treatment of objects of the specified types are supplied. Therefore dynamic types [7] can be managed.

For example, let us suppose that a suitable software management module is furnished for the simple data system of Fig. 10. At execution time, to allow the use of a matrix of matrices of integers, such a module could perform the instantiation of a variable of class matrix. This variable is an object representing a bidimensional array of elements of class matrix; each of these elements is constituted by a bidimensional array whose elements are integers numbers. So at run-time it becomes possible to use variables of type $\mathbb{M}(\mathbb{M}(\mathbb{Z}))$, while no such a type has been explicitly declared in the data system (actually its definition is implicitly given once the matrix and the integer definitions are given).

4.2. Method abstraction

In Section 2 it has been seen that algebraic structures present some method definitions. In general these methods specify the semantics of fundamental operations in the structure (e.g. the matrix addition method in Matrix parametric structure, or the "div" and "mod" methods defined in the Euclidean Domain abstract structure).

Now it will be shown how OOP methodology permits the enrichment of the definition of algebraic structures, giving rise to abstract structures in which computing methods become available components. In fact as far as the method abstraction is concerned, the inheritance allows the use of the code provided for a higher structure in all its subdomains, without any redefinition.

```
Unit ring: class;
    (* no data structure to be instantiated *)
    (* no implementation of the methods *)
    (* no instantiation operation to be supplied in the execution part *)
  virtual unit ADD: function (A: ring): ring;
  end ADD;
  virtual unit MULT: function (A: ring): ring;
  end MULT;
end ring;

unit integer-number: ring class;
    (* data structure *)
  var n: integer;
  virtual unit ADD: function (A: integer-number): integer-number;
    begin
      result := new integer-number;
      result.n := n + A.n;
    end ADD;
  virtual unit MULT: function (A: integer-number): integer-number;
    begin
      result := new integer-number;
      result.n := n * A.n;
    end MULT;
    (* no particular instantiation instructions are needed for this ground structure,
    besides the instantiation of the defined data structure *)
end integer-number;

unit matrix; ring class (n: integer);      (* data structure *)
var s: array of array of ring;
  virtual unit ADD: function (A: matrix:): matrix:;
    var i, j: integer;
    begin
      result := new matrix:(n);
      for i := 1 to n
        do
        for j := 1 to n
          do
          result.s(i, j) := s(i, j).ADD(A.s(i, j));
          od;
        od;
    end ADD;
  virtual unit MULT: function (A: matrix:): matrix;:
    var i, j:integer;
    begin
      result := new matrix:(n);
      ...
      ...
    end MULT;
begin                      (* execution part of class matrix *)
  array s dim(1:n);        (* partial data structure instantiation *)
  for i := 1 to n
    do
      array s(i) dim (1:n);
    od;
end matrix;
```

Fig. 10. Partial Loglan implementation of RING, MATRIX, INTEGER-NUMBER.

Looking at the previous example on matrix structure definition, the addition and multiplication methods were defined at their natural level of abstraction. Any other substructure (element) of Matrix will use such methods as general algorithms to perform additions and multiplications. Such methods can be seen as characteristic properties of the parametric structure and its definition must be necessarily added to have a complete specification.

In the practice of software development for computer algebra systems, methods can just as well be intended as implementation of general algorithms operating over classes of abstract data types.

These interpretations of method abstractions will be explained with an example: the Hensel lifting method [20].

With this example it will be shown how the Hensel method for polynomial equations can extend an abstract structure by including in it an algebraic computational mechanism. This extension leads to a more powerful algebraic structure in which the computing method becomes a newly introduced characteristic component for the abstract structure itself.

As is well known, this method gives an iterative mechanism to solve equations of the following type:

$$\Phi(G, H) = 0$$

where

$$\Phi : \mathbb{D}[x] \times \mathbb{D}[x] \to \mathbb{D}[x],$$

$\mathbb{D}$ being an abstract Euclidean domain.

The approximate solution G_k, H_k (such that $\Sigma(G_k, H_k) \equiv 0 \bmod I^k$, where I is an ideal in $\mathbb{D}[x]$ and where k is a suitable degree of approximation) of the given equation is obtained starting from an appropriate initial approximation, i.e. $G_1(x)$ and $H_1(x)$, such that

$$\Phi(G_1, H_1) \equiv 0 \bmod I.$$

In the following $+_{\mathbb{D}}, =_{\mathbb{D}}, 0_{\mathbb{D}}$, will indicate respectively the sum operation, the equality relation and the zero element in $\mathbb{D}$. Moreover, the function EVAL($\Phi(G, H)$, G_j, H_j), computes the value of the function Φ at the point (G_j, H_j); the function DERIVE computes the polynomial derivative function; the function $\text{mod}_{\mathbb{D}}$, given $a \in \mathbb{D}$, computes a mod I and the function DUPE (diophantine univariate polynomial equation), solves the diophantine equation that is the fundamental step at each iteration of the Hensel algorithm.

The generality of this method makes it an apt tool to deal with approximated methods in a uniform view, in the same algebraic context. The Hensel algorithm can perform different computational methods according to different specialization of the parameters G and H (i.e. different forms of the equation). The following scheme considers $F \in \mathbb{D}[x]$.

- Factorization: $\Phi(G, H) = F - GH = 0$;

Hensel Lifting Algorithm

Input: $\Phi(G, H), G_1, H_1 \in \mathbb{D}[x], i \in \mathbb{D}[x], k \in \mathbb{N}^+$

Output: $G_k, H_k : \Phi(G_k, H_k) \equiv 0 \bmod I^k$

```
Begin
  j := 1;
  C := EVAL(Φ(G, H), G1, H1);
  while (j < k) ∧ (¬C =D 0D)
    do A := DERIVE(Φ(Gj, Hj), G);
       B := DERIVE(Φ(Gj, Hj), H);
       A := modD(A, I^k);
       B := modD(B, I^k);
       C := modD(C, I^k);
       ΔGj := 0D;
       ΔHj := 0D;
       (ΔGj, ΔHj) := DUPE(C, A, B);
       Gj+1 := Gj +D ΔGj;
       Hj+1 := Hj +D ΔHj;
       C := EVAL(Φ(G, H), Gj+1, Hj+1);
    od
end.
```

Fig. 11. Hensel lifting algorithm.

- Nth root of a function G: $\Phi(G, H) = F - G^n = 0$;
- Legendre polynomial determination: $\Phi(G, t) = (1 - 2x + t^2)G^2 - 1 = 0$;
- Newton's method: $\Phi(G, H) = G = 0$.

This can be considered as a degree of abstraction, intrinsic to the Hensel method. A different kind of abstraction can be appreciated, if this method is inserted into the hierarchy of the algebraic structures seen above. In Fig. 11, it is shown how the Hensel algorithm can be defined at the highest level of abstraction, by integrating its definition into the structure $\mathbb{P}[x]$. Then every instantiation of $\mathbb{P}[x]$ (e.g.: $\mathbb{Z}[x]$) automatically owns this algorithm.

In this case the Univariate Polynomial structure is enriched by the definition of the Hensel method. In the execution space the objects representing, for example, a univariate polynomial over integers are generated.

A formal specification of the Hensel algorithm is given in Fig. 11, assuming it is embedded into the polynomial parametric structure definition. For the sake of simplicity the algorithm when G and H are univariate polynomials will be described (so that only the function DUPE is applied).

5. Conclusions

A programming system for symbolic computation can be designed and implemented following an OOP approach. Basic characteristics of this approach are the

support of data abstraction through inheritance and polymorphism. To support full symbolic computation, logic capabilities must be embedded into the system, at the specification level and at the level of use of what has been specified.

In this work a special classification of the algebraic structures has been elaborated, based on the different specification requirements of *abstract*, *parametric* and *ground* structures. On these diverse requirements the different computational characteristics of these structures are based:

- abstract structures allow only symbolic computation (in its extended meaning of "properties manipulation");
- on parametric structures symbolic computations are allowed, while numeric computations are possible only once a complete variable instantiation is provided;
- ground structures can be directly instantiated and allow numeric and symbolic computation to be performed.

Looking at this classification it has been shown that, following the OOP approach, it is possible to treat the formal algebraic structures maintaining the characteristics of their natural hierarchy.

For the definition and manipulation of mathematical data structures, one of the most important characteristics is represented by the support of dynamic typing. With this feature, the expressiveness of the designed data type system can be fully exploited, using at run-time data structures which have not been explicitly defined; indeed these data are implicitly defined into the hierarchy of the specified structures. This feature allows the same potential of the genericity approach to be obtained, without the necessity of manipulating the definition of an abstract data type for each possible ground structure. Examples of the expressivity of this mechanism have been shown: matrix of integer polynomials, or matrix of integers, or any other composition of the defined structures (see Fig. 9) can be instantiated starting from these fundamental structures; they are instances of dynamic types, occurring at run-time. If a genericity approach should be followed, each single data type must have been defined at compile time, implementing the related type management module. The genericity approach does not preserve the distinction between abstract and parametric structures; on the contrary it has been seen that this distinction can be maintained by differentiating the completeness of the specification.

Essentially the OOP approach presented here, allows many redundancies in the construction of the data type system to be avoided. Moreover the subdivision proposed for the algebraic structures, and its fitness for a fully OOP scheme can improve the correctness of the manipulation.

References

[1] S.K. Abdali, G.W. Cherry and N. Soiffer, An object-oriented approach to algebra system design, in: B.W. Car, ed., *Proc. SYMSAC 86.*

[2] W.M. Bartol et al. *Report on the Programming Language LOGLAN 82* (Polish Scientific Publishers, Warsaw, 1984).

[3] S. Bonamico, G. Cioni and A. Colagrossi, A Gentzen-based deduction method for mathematical problem solving, in E. Balagurusamy and B. Sushila, eds., *Computer Trends and Applications*, Indian Computing Congress Series (TATA McGraw-Hill, Bombay, 1990) 104–113.

[4] G. Cioni and A. Salwicki, eds., *Advanced Programming Methodologies* (Academic Press, New York, 1989).

[5] J. Fitch, Solving algebraic problems with REDUCE, *J. Symbolic Comput.* **1**(2) (1985) 211–227.

[6] G. Forcellese and M. Temperini, The use of a deduction method in an object-oriented frame, in: E. Balagurusamy and B. Sushila, eds., *Computer Trends and Applications*, Indian Computing Congress Series (TATA McGraw-Hill, Bombay, 1990) 124–133.

[7] J.W. Goodwin, Why programming environments need dynamic data types, *IEEE Trans. Software Eng.* **7**(5) (1981).

[8] R. Jenks and B. Trager, A language for computational algebra, IBM T.J. Watson Research Center, Report, Yorktown Heights, NY, 1987.

[9] C. Limongelli, M.B. Mele, M. Regio and M. Temperini, Abstract specification of mathematical structures and methods, in: A. Miola, ed., *Proc. 1st Internat. Symp. on Design and Implementation of Symbolic Computation Systems (DISCO '90)*, Lecture Notes in Computer Science, Vol. 429 (Springer, Berlin, 1990) 61–70.

[10] B. Liskov, Data abstraction and hierarchy. *Proc. OOPSLA '87* (1987).

[11] B. Meyer, Object-oriented software construction (Prentice-Hall, Englewood Cliffs, NJ, 1988).

[12] B. Meyer, Static typing for Eiffel, Interactive Software Engineering Report, 1989.

[13] A. Miola, TASSO: a system for mathematical problem solving, in: E. Balagurusamy and B. Sushila, eds., *Computer Trends and Applications*, Indian Computing Congress Series (TATA McGraw-Hill, Bombay, 1990) 53–62.

[14] A. Snyder, Inheritance and the development of encapsulated software systems, in: B. Shriver and P. Wegner, eds., *Research Directions in Object-oriented Programming* (MIT Press, Cambridge, MA, 1987) 165–188.

[15] M. Temperini and R. Vitale, A formal approach to the specification and manipulation of mathematical objects, R.251 IASI-CNR, Roma, 1988.

[16] P. Thomas, H. Robinson and J. Emms, *Abstract Data Types—Their Specification, Representation and Use* (Clarendon Press, Oxford, 1988).

[17] P. Wang and R. Pavelle, MACSYMA from *F* to *G*, *J. Symbolic Comput.* **1**(1) (1985) 69–100.

[18] P. Wegner, The object-oriented classification paradigm, in: B. Shriver and P. Wegner, eds., *Research Directions in Object-oriented Programming* (MIT Press, Cambridge, MA, 1987) 479–560.

[19] M. Wirsing and M. Broy, A modular framework for specification and implementation, in: *Proc. TAPSOFT 89* Lecture Notes in Computer Science, Vol. 351 (Springer, Berlin, 1989) 1–42.

[20] D.Y.Y. Yun, Algebraic algorithms using P-adic construction, in: *Proc. ACM SYMSAC* (1976).