STUDIA UNIV. "BABEŞ-BOLYAI", INFORMATICA, Volume I, Number 1, March 1996

USING MAPLE IN PROCESSING POISSON EXPRESSIONS

B. PÂRV

Abstract. This paper presents an implementation of the Poisson Symbolic Processor by using the Computer Algebra System Maple and object-oriented facilities introduced by version 1.0 of Gauss package. An hierarchy of abstract classes (called *categories* in Gauss) is defined; each class represents an entity which is used to construct a Poisson expression (which is the partial sum of a Poisson series): monomial part, trigonometric part, term, expression. From each category there is derived at least one "concrete" class (called *domain* in Gauss). Each domain derived from the same category has its own representation (unknown at category level) and, possibly, new code for its operations (in order to achieve efficiency). This paper describes an experimental work; it constitutes the beginning of a long-term research. Computer algebra system Maple is used here as testing tool, due to its interactivity, user-friendly interface, and ready-to-use capabilities. After the experimentation work will be finished, we intend to develop an operational version of the processor using a high-level programming language.

1. Poisson series and expressions

A *Poisson series* have the form:

$$S = \sum_{i=0}^{\infty} C_i x_1^{j_1} x_2^{j_2} \cdots x_m^{j_m} \begin{matrix} sin \\ cos \end{matrix} (k_1 y_1 + k_2 y_2 + \cdots + k_n y_n), \tag{1}$$

where: C_i are *numerical coefficients*; $x_1, x_2, \cdots, x_m$ are *monomial variables*; $y_1, y_2, \cdots, y_n$ are *trigonometric variables*; $j_1, j_2, \cdots, j_m$ and $k_1, k_2, \cdots, k_n$ are exponents and coefficients, respectively. The summation index i covers the set of all possible combinations of exponents j and coefficients k ($j \in \mathrm{Z}^m$, $k \in \mathrm{Z}^n$).

One can write the form (1) of a Poisson series as follows:

$$S = \sum_{i=0}^{\infty} T_i, \tag{2}$$

Received by the editors: July 27, 1996.

1991 *Mathematics Subject Classification.* 68Q40, 68Q65.

1991 *CR Categories and Descriptors.* D.1.5 [Programming Techniques]: Object-Oriented Programming; D.3.2 [Programming Languages]: Language Classifications – object-oriented languages; D.3.3 [Programming Languages]: Language Constructs and Features – abstract data types; I.1.3 [Algebraic Manipulation]: Languages and Systems – special-purpose algebraic systems.

in which T_i is a *term* of this series:

$$T_i = C_i P_i F_i,$$

where the *polinomial part* P_i is:

$$P_i = x_1^{j_1} x_2^{j_2} \cdots x_m^{j_m}, \tag{3}$$

and the *trigonometric part* F_i have the form:

$$F_i = sin/cos(k_1 y_1 + k_2 y_2 + \cdots + k_n y_n), \tag{4}$$

In practice one does not operate with Poisson series, but with partial sums of these ones, the so-called *Poisson expressions*:

$$S = \sum_{i=0}^{N} T_i, \ N \in \mathrm{N} \tag{5}$$

This new version of the Poisson Symbolic Processor presented here is able to manipulate Poisson expressions of the form (5).

2. Maple and Object-Oriented Programming

The Maple programming language [2, 3, 4] is a procedure-based one. Despite this fact, due to the powerful data types available, one can simply implement all the basic concepts of the object-oriented programming. At least two Maple features facilitate this task:

1. interpretative nature of Maple, which means, from our viewpoint, dynamic binding of function calls, and
2. Maple table, in fact a data structure corresponding to the hash table, which is well-suited for implementing class definitions.

The Gauss package (see [5]), presented in the next section, constitutes an example of implementing object-oriented concepts in Maple.

Why object-oriented computer algebra? This question is not a new one, but possible answers were given only in the last decade [1, 5, 7]. The practical answers appeared a few years ago, especially IBM's AXIOM, and the Gauss package in Maple.

First, an object-oriented approach is well-suited for real-world modelling. In the case of computer algebra systems, the real objects are mathematical ones (numbers, functions, operators, functionals, sets, structures, more generally analytical expressions)

These objects have a well-defined behaviour (by axioms, rules, theorems, properties), and, finally, the goal of the mathematical science is to study this behaviour.

Second, the object-oriented approach uses classification mechanisms as tools for managing real world complexity. These mathematical objects can be grouped in classes, and the inheritance relations between classes can be expressed by the class hierarchies.

Third, an other mathematical-specific thing, the parameterization, was borrowed in the world of programming languages; the term used here is genericity. By using object-oriented programming and genericity, one can express, in a natural way, the class hierarchies of mathematical objects.

2.1. **Gauss Package.** The Gauss Package introduces a new approach of programming, based on parameterized (generic) data types.

There are two main concepts: category and domain. A *category* is an *abstract* and *parameterized* (*generic*) class, and a *domain* is a *concrete* class in the object-oriented terminology. An object is an instance of a domain. The category does not know the representation of objects - this is essentially the meaning of the *abstract* word; conversely, the domain (which represents a concretization of a category) defines the representation of its objects. The category definition contains: ancestor(s) specification, definition of signatures for all specific operations, and the implementation of all operations which can be implemented at category level (without knowing the representation). The domain definition defines domain category (categories), states the representation and implements the remaining operations (unimplemented at category level). Using multiple inheritance, Gauss implements a wide hierarchy of categories, all of them being algebraic objects. The root of this hierarchy is **Set** category, with the following operations:

1: `=, <>` – comparison operators

2: `Input` – conversion from Maple representation to domain representation (known as *initial constructor* in the object-oriented terminology)

3: `Output` – conversion from domain representation to Maple representation

4: `Random` – generates a pseudorandom value from domain

5: `Type` – predicate which defines the domain structure; it is used to test if an expression belongs to domain (from representation point of view).

2.2. **Abstract and parameterized classes.** Gauss domains are in fact parameterized (generic) classes. From implementation point of view, a domain is a Maple function, which

returns a Maple table with operation names as table entries. The domain ancestor(s), the class instances, and the properties of defined operations are also table entries.

For example, the domain `Integer()` returns the table in which the entries are: integer addition ('+'), subtraction ('-'), multiplication ('*'), greatest common divisor (gcd), zero and unit elements, etc.

The parameters of every domain are any valid Maple expressions, including other domains. The Gauss package contains a number of *usual* domains (see [5]). For example, the domain `DUP(R, x)` from Gauss (`DenseUnivariatePolynomial(R,x)` i.e. the domain of univariate polynomials with real coefficients) has two parameters: the coefficient ring (`R`) and the variable name (`x`). The first parameter, `R` must be a Maple domain, while the second must be a Maple name.

From programming viewpoint, a Maple category is the Maple (Gauss) implementation of an abstract data type. One can derive several domains from the same category, simply by choosing different representations (and then by implementing accordingly the corresponding operations). For example, Gauss package includes the `ExponentVector(X)`, `EV(X)` category, where `X` is a list of Maple names. This category defines the *exponent vector* abstract data type: if `X` contains the symbols $x_1, x_2, \ldots, x_n$, then `EV(X)` will define the set of all monomials in the variables `X`, i.e. monomials of the form:

$$x_1^{e_1} x_2^{e_2} \cdots x_n^{e_n} \tag{6}$$

where x_j are variable names, and e_j are exponents (natural numbers). If the list `X` is ordered, and if this list is a parameter of the domain being considered, then for every monomial of the above form it is necessary to consider only the list (vector) of exponents $E = \{e_1, e_2, \cdots, e_n\}$.

One of the major benefits of the object-oriented programming is software reuse. Every (object-oriented) application framework contains a class hierarchy, from which, by using inheritance, one can extend (derive) new classes needed for an application-specific domain, with a minimum programming effort. In most situations, the existing classes are almost all what we need. By applying these considerations to the Gauss package (which is, in fact, an example of application framework), the programming steps are the following:

1. Define all the necessary categories

2. Define all the corresponding domains
3. Instantiate the domains by setting properly the generic parameters
4. Manipulate the obtained objects by using their operations.

2.3. **Deriving new categories and domains.** For a new category (for example `ExponentVector`), does not matter the representation of its objects; we are only interested in choosing the specific operations we need for manipulating the objects. The *natural* operations with monomials are multiplication, division, and common factor of two monomials. Consequently, besides the already inherited operations, the `ExponentVector` category must contain the definition of the following operations:

- addition of two exponent vectors (which corresponds to monomial multiplication)
- subtraction of two exponent vectors (which corresponds to monomial division)
- the minimum of two exponent vectors (which corresponds to monomial greatest common divisor).

The operations inherited from `Set` must be re-implemented. There are two alternatives: to implement general algorithms at category level (which, in turn, must invoke the representation-specific functions, defined at domain level), or *to postpone* (the used term is *to deferre*) this implementation to the domain level (where the representation is known).

After the category is completely defined, one can derive from it the corresponding domains. Usually, one can derive many domains from the same category. For example, in the `Gauss` package, there are presented four domains derived from the `ExponentVector` category: `DenseExponentVector`, `PrimeExponentVector`, `MapleExponentVector`, and `MacaulayExponentVector`. Every such a domain uses a different representation, and, consequently, specific algorithms for the operations it implements.

3. The class hierarchy for the Poisson Symbolic Processor

Based on the definition (1), the elements of a Poisson series (expression) can be defined in an hierarchical way. First, we consider the term components (factors): the coefficient, monomial part, and trigonometric part.

One can easy observe the elements well-suited for parameterization: the lists of monomial and trigonometric variables. For a concrete problem, these variables (symbols) are parameters of its corresponding mathematical model.

The coefficient is a rational number. Because Maple contains rational data type as a builtin data type, there are no problems in working with rational numbers; besides this, the **Gauss** package contains the definition of the **Rational** domain.

The monomial part can be considered as an instance of an exponent vector. Because the existing **ExponentVector** category deals with exponents considered natural numbers, one must redefine this category in order to manipulate integer exponents.

The trigonometric part has two components: the trigonometric function to be applied (it sin or *cos*) and its argument (a linear combination of trigonometric variables, the with integer coefficients). For the same reason as in the case of **ExponentVector**, the argument can be expressed in the terms of a list of coefficients, if the list of trigonometric variables is considered as parameter. The **CoefficientVector** category described below implements the definition of this list (vector) of coefficients. Based on this category, one can define **TrigPart** category, which describes the behaviour of the trigonometric part.

Finally, the **PoissonExpression** category defines the behaviour of the Poisson expressions, taking the coefficient ring, the **ExponentVector**, and the **TrigPart** categories as parameters.

3.1. **The ExponentVector category.** This category is parameterized with the list of monomial variables (list of names). The operations are:

- **Variables** - returns the list of monomial variables
- **Index** - returns the index of the argument - a monomial variable name - in the list of monomial variables
- **Exponent** - returns the exponent of the argument - a monomial variable name -in the current object
- **Dim** - returns the number of monomial variables
- **Vect2List** - conversion from internal representation to list of integers
- **List2Vect** - conversion from list of integers to internal representation.
- *addition* - multiplication of monomial parts
- *subtraction* - division of monomial parts
- *comparison* - returns -1, 0 sau 1, i.e. less than, equal, or greater than.

The last two operations, `List2Vect` and `Vect2List`, are defined but not implemented in this category (by using object-oriented terminology, these function can be referred as *pure virtual functions*). By using these functions, one can implement, at category level, some of the defined operations (despite the fact one does not know the representation). For example, the algorithm for the addition of two exponent vectors consists of the following steps:

1. operand conversion from internal representation (unknown at this point) to list representation (natural representation of a vector) using `Vect2List`;
2. adding lists, element by element;
3. result conversion from list representation to internal representation using `List2Vect`.

Besides the above-discussed operations, the `ExponentVector` category definition also contains the implementation of some conversion functions, inherited from `Set`: `Input` and `Output`. Actually, these operations are implemented in all categories we discuss.

The following text represents the definition of `ExponentVector` category:

```
#
# ExponentVector(X)
#
ExponentVector := proc(X:list(name))
  local D, env;
  if not type(X, list(name)) then ERROR('invalid argument') fi:
  D := NewCategory();
  D := OrderedAbelianMonoid( op(D) );
  addCategory( D, ExponentVector );
  defOperation( Variables, List(Name), D );
  defOperation( Index, Name -> Integer, D );
  defOperation( Exponent, [D, Name] -> integer, D );
  defOperation( Dim, Integer, D );
  defOperation( List2Vect, List(Integer) &-> D, D );
  defOperation( Vect2List, D &-> List(Integer), D );
  defOperation( '+', [D, D] &-> D, D );
  defOperation( '-', [D, D] &-> D, D );
  defOperation('<>=', [D, D] &-> Union(-1,0,1), D );
# implement some operations...
  op(D)
end: # ExponentVector Category
```

For the reasons discussed above, there is another *deferred* operation, denoted by '<>='. Like `Vect2List` and `List2Vect`, this operation is used for implementing other comparison operations ('<' and '=').

3.2. **The CoefficientVector Category.** This category is parameterized with the list of trigonometric variables (list of names). The operations include:

- `Variables` - returns the list of trigonometric variables
- `Index` - returns the index of the argument - a trigonometric variable name - in the list of trigonometric variables
- `Coeff` -returns the coefficient of its argument - a trigonometric variable name - in the current object
- `Normalize` - normalizes the argument of a trigonometric function
- `Dim` - returns the number of trigonometric variables
- `Vect2List` - conversion from internal representation to list of integers
- `List2Vect` - conversion from list of integers to internal representation
- *addition* - addition of trigonometric function arguments
- *subtraction* - subtraction of trigonometric function arguments)
- *multiplication* with a rational number
- *comparison* of two trigonometric coefficients.

Because the linear combinations of trigonometric variables are arguments of the trigonometric functions *sin* and *cos*, these combinations need to be *normalized*. Taking into account the parity of these functions, we define the *normal form* of an argument the form in which the first non-zero coefficient is positive. The `Normalize` operation returns the normal form of its argument, and a state value (-1 or 1) which indicates the sign of the first non-zero coefficient in the initial argument.

The following text represents the definition of `CoefficientVector` category:

```
#
# CoefficientVector(X)
#
CoefficientVector := proc(X:list(name))
local D, env;
if not type(X, list(name)) then ERROR('invalid argument') fi:
D := NewCategory();
D := OrderedAbelianMonoid( op(D) );
addCategory( D, CoefficientVector );
defOperation( Variables, List(Name), D );
defOperation( Index, Name &-> integer, D );
defOperation( Coeff, [D, Name] &-> rational, D );
defOperation( Normalize, [D, integer] &-> D, D );
defOperation( Dim, Integer, D );
defOperation( List2Vect, List(rational) &-> D, D );
defOperation( Vect2List, D &-> List(rational), D );
defOperations( {'+', '-'}, [D, D] &-> D, D );
defOperation( '.', [rational, D] &-> D, D );
```

```
defOperation( '<>=', [D, D] &-> Union(-1,0,1), D );
# implement some operations...
op(D)
end: # CoefficientVector Category
```

3.3. **The TrigPart category.** The `TrigPart` category has as argument the domain of coefficient vectors, which in turn must be derived from `CoefficientVector` category. The operations are:

- `List2Trig` - conversion from list to internal format
- `Trig2List` - conversion from internal format to list
- `CoefficientVector` - returns the CoefficientVector domain
- `Variables` - returns the list of trigonometric variables
- `Argument` - returns the coefficient vector of an object
- `Function` - returns the trigonometric function
- `Coeff` - returns the coefficient of its argument - a trigonometric variable name - in the current object
- `Dim` - returns the number of trigonometric variables
- `sin, cos` - apply the trigonometric functions on an `CoefficientVector` object
- *multiplication* of two trigonometric parts
- *comparison* of two trigonometric parts.

In the case of multiplication, in order to preserve the form of the trigonometric part, one must apply the following rules:

$$sin(x) * sin(y) = (cos(x-y) - cos(x+y))/2,$$
$$sin(x) * cos(y) = (sin(x-y) + sin(x+y))/2,$$
$$cos(x) * cos(y) = (cos(x-y) + cos(x+y))/2.$$

The following text represents the definition of `TrigPart` category:

```
#
# TrigPart
#
TrigPart := proc()
  local S, P, T, env;
  S := args[1];
  if not hasCategory(S, CoefficientVector)
    then ERROR('the argument must be an CoefficientVector') fi;
  P := newCategory();
  addCategory(P, TrigPart);
  defOperation( List2Trig, Record(Name, S) &-> P, P);
  defOperation( Trig2List, P &-> Record(Name, S), P);
  defOperation( CoefficientVector, CoefficientVector, P);
  defOperation( Variables, List(Name), P);
```

```
  defOperation( Argument, P &-> S, P);
  defOperation( Function, P &-> Union(cos, none, sin), P);
  defOperation( Coeff, P &-> Rational, P );
  defOperation( Dim, Integer, P);
  defOperations( {sin, cos}, [Rational, S] &-> [Rational, P], P );
  defOperation('*', [P, P] &-> List(List(Rational, P)), P);
  defOperation('<>=', [P, P] &-> Union(-1, 0, 1), P);
# implement some operations...
  op(P)
end: # TrigPart Category
```

3.4. **The PoissonExpression Category.** The `PoissonExpression` category has the following parameters:

- the coefficient ring (usually the ring of rational numbers)
- the exponent vector domain
- the trigonometric part domain.

and the defined operations are:

- `List2Expr` - conversion from list to internal format
- `Expr2List` - conversion from internal format to list
- `CoefficientRing` - returns the coefficient ring domain
- `ExponentVector` - returns the exponent vector domain
- `TrigPart` - returns the trigonometric part domain
- `PolVariables` - returns the list of monomial variables
- `TrigVariables` - returns the list of trigonometric variables
- `PolDim` - returns the number of monomial variables
- `TrigDim` - returns the number of trigonometric variables
- *multiplication* of a term with a rational number
- *addition* (n-ary operation)
- *multiplication* (n-ary operation)
- `comparison` of two terms
- `Diff` - partial derivative with respect to a specified variable
- `Int` - integration with respect to a specified variable.

The following text represents the definition of `PoissonExpression` category:

```
#
# PoissonExpression
#
PoissonExpression := proc()
  local R, E, S, P, T, env;
  R := args[1];
```

```
  E := args[2];
  S := args[3];
  if not hasCategory(R, Ring)
  then ERROR('1st argument must be a Ring')
  elif not hasCategory(E, ExponentVector)
  then ERROR('2nd argument must be an ExponentVector')
  elif not hasCategory(S, TrigPart)
  then ERROR('3nd argument must be an CoefficientVector') fi;
  P := newCategory();
  addCategory(P, PoissonExpression);
  if hasCategory(R, UFD) then P := UFD(P)
  elif hasCategory(R, GcdDomain) then P := GcdDomain(P)
  elif hasCategory(R, IntegralDomain) then P := IntegralDomain(P)
  elif hasCategory(R, CommutativeRing) then P := CommutativeRing(P)
  else P := Ring(P)
  fi;
  defOperation( List2Expr, List(Record(R, E, S)) &-> P, P);
  defOperation( Expr2List, P &-> List(Record(R, E, S)), P);
  defOperation( CoefficientRing, Ring, P);
  defOperation( ExponentVector, ExponentVector, P);
  defOperation( TrigPart, TrigPart, P);
  defOperation( PolVariables, List(Name), P);
  defOperation( TrigVariables, List(Name), P);
  defOperation( PolDim, Integer, P);
  defOperation( TrigDim, Integer, P);
  defOperation( '.', [R, P] &-> P, P );
  defOperation( '+', Nary(P) &-> Nary(P), P);
  defOperation( '*', Nary(P) &-> Nary(P), P);
  defOperation('<>=', [P, P] &-> Union(-1, 0, 1), P);
  defOperations( {Diff, Int}, [P, Name] &-> P, P);
# implement some operations...
  op(P)
end: # PoissonExpression Category
```

4. Implementation issues

Two different domains were implemented for each of the above-discussed categories. Each implementation uses a different internal representation of the exponent vector and coefficient vector: *list* and *Gödel coding*.

In the case of *list representation* every exponent or coefficient vector is represented as a Maple list, in which all the elements are integer numbers (Maple integers). Consequently, the `List2Vect` and `Vect2List` operations will implement the identity function and the comparison operator is based on the lexicographic ordering.

In the case of *Gödel coding*, an exponent or coefficient vector is represented as rational number, by using the first n prime numbers (n is the number of elements in the vector):

$$(a, b, c, d, \ldots) \rightarrow 2^a 3^b 5^c 7^d \cdots$$

Taking into account the integer nature of vector elements (positive or negative numbers), it follows that the result of this mapping is a rational number; its numerator will contain the product of the powers with positive exponents, while the denominator will include all the powers with negative exponents. The function which define this coding is bijective, so it follows that this representation is unique and it exists the inverse transformation. Thus, one can define the `List2Vect` and `Vect2List` operations as follows: `List2Vect` will implement the direct mapping, while `Vect2List` the inverse one. The considered ordering is the natural ordering of the rational numbers.

Domain implementation also contains the `Type` operation (inherited from `Set`; it returns a boolean value, depending on whether its arguments satisfies the rules stated for the concrete representation chosen), and its zero element `D[0]`.

Finally, the aritmetic operations were redesigned, in order to exploit the specific features of the concrete representation. After all, the invoking of the conversion routines `List2Vect` and `Vect2List` was removed, in order to gain computing speed.

The implemented domains are: `ListExponentVector` and `PrimeExponentVector`, `ListCoefficientVector` and `PrimeCoefficientVector`, `ListTrigPart` and `PrimeTrigPart`, `ListPoissonExpression` and `PrimePoissonExpression`.

References

[1] S.K. Abdali, G.W. Cherry, N. Soiffer, *An Object Oriented Approach to Algebra System Design*, ISSAC-86, 24-30, 1986.

[2] B.W. Char et al., *Maple V – Library Reference Manual*, Springer, 1992.

[3] B.W. Char et al., *Maple – Language Reference Manual*, Springer, 1992.

[4] B.W. Char et al., *First Leaves: A Tutorial Introduction to Maple*, Springer, 1992.

[5] D. Gruntz, and M. Monagan, *Introduction to Gauss*, MapleTech 9, 1993, 23-35.

[6] R. Jenks, and R. Sutor, *AXIOM – The Scientific Computation System*, Springer, 1992.

[7] M. Monagan, *Signatures + Abstract Data Types = Computer Algebra – intermediate expression swell*, PhD Thesis, University of Waterloo, 1989.

[8] B. Pârv, *Poisson Symbolic Processor*, Studia, Mathematica, XXXIV, 1989, No. 3, 17-29.

"Babeş-Bolyai" University, Faculty of Mathematics and Computer Science, RO-3400 Cluj-Napoca, Romania

E-mail address: bparv@cs.ubbcluj.ro