STUDIA UNIV. BABEŞ-BOLYAI, INFORMATICA, VOLUME XLIII, NUMBER 2, 1998

OBJECT-ORIENTED SYSTEM FOR IMPLEMENTING SYMBOLIC COMPUTATIONS WITHIN AN ALGEBRAIC HIERARCHY

ALINA ANDREICA AND ADRIAN MONEA

Abstract. The paper describes an independent software system which performs symbolic computations within the hierarchy of algebraic structures that contains semigroups – monoids – groups – abelian groups – rings – (abelian rings, fields – abelian fields). The system was designed by using an object-oriented approach as a flexible system that models the above-mentioned algebraic hierarchy, the present version being implemented in Microsoft Visual C++. The system provides symbolic computation facilities that are not offered within commonly-used symbolic computation systems by giving the possibility of operating in abstract domains, independently of the massive symbolic computation systems based on type theory.

1. The system's motivation and its working principles

The problem of performing formal calculus within abstract algebraic domains, unsolved by the most wide-spread symbolic computation systems, initiated the development of type theory and of symbolic computation systems (SCS) based on it [3], the most important of them being AXIOM [5]. Within these SCS, usual computation domains are particular cases of the abstract ones, being defined by the rigorous principles of the type theory, using a hierarchical structure with multiple inheritance [4]. The drawbacks of these systems, which introduce all working domains using the constructive and functional principles of domain and category theory, are their dimension and complexity. Therefore, one of the orientations within this research domain is to extend commonly used SCS with abstract facilities [2]. The system that is further presented aims at introducing an object-oriented model of reduced complexity for the semigroup – monoid – group – abelian group – ring – (abelian ring, field – abelian field) algebraic hierarchy and at outlining the design problems which arise from the implementation of such a SCS in a usual programming language.

The system was initially implemented in Borland C++ 3.1 for DOS [6], [7] and was later ported to Visual C++ 5 for Windows [8] in order to add a graphical, more user-friendly, interface and to facilitate access to a larger memory space. The choice of C++ language is motivated by the fact that the possibility of introducing inheritance relations is very important for the implementation of the above-mentioned hierarchy of algebraic domains.

From the user's point of view, the working principle of the system is a declarative one: after declaring a structure of a certain type, all subsequent calculus will be done within that structure, until a new structure is declared. If one wants to return to the context of a previously declared structure, that structure (domain) will be redeclared; consequently, the system will perform the computations in the previously defined context (for example, the user will be able to use variables that were defined within that domain). The user's identifiers will contain at most 20 letters and the reserved keywords for declaring a domain are: Semigrup, Monoid, Grup, GrupCom, Inel, InelCom, Corp, CorpCom. The spaces within user's commands will be ignored.

Thus, the commands entered by the user can be of the following types:

1. *Domain declaration*, of the form (X,+) : Grup or (Ccom, +, *) : CorpCom. Algebraic domains will be named with user identifiers, and the characters for operators can be chosen from the following list: +,-,*, &, ^, %, $, #, @, !, |. A domain declaration may contain the neutral element(s) of the domain, if the operator(s) has (have) neutral elements, for example: (X,+, e) : Grup or (Ccom, +, *, e, eo) : CorpCom. When neutral elements are not declared, they will have implicit values: e for the first operator and en for the second.
2. *Assignment*, of the form variable = expression, where expression contains operators from the current domain, identifiers (with or without a previously assigned value) and possibly parentheses. The expression will be evaluated by replacing already assigned variables with their values and canceling neutral or symmetrical elements; the result will be returned to the user. The result's form will depend on the axioms associated with the current domain's operators (associativity, commutativity, distributivity, etc). Thus, in the case of commutative operators, the subexpression operands will be lexicographically sorted.
3. An *expression* will be evaluated according to the principles stated earlier.
4. *Symmetrical element declaration*, of the form Sim(a) = as, if the current domain has only one operator or Sim(a,+) = as if the current domain has two operators. The lists of symmetrical elements from each structure will be retained and processed afterwards in order to produce correct results.
5. *Equality test*, of the form expression1==expression2, where the two expressions are given in the usual syntax, containing domain operators, user identifiers (or implicit neutral elements) and, optionally, parentheses. The result returned will be True or False, depending on the test's value of truth. The validations will be performed on the expressions brought to canonical form, depending on the axioms associated with the current domain's operators.
6. *Difference test*, of the form expression1!=expression2. The result will be True if the canonical forms of the two expressions are different and False, otherwise.
7. *Display the variables from the current domain*, of the form _Listavar.
8. *Display of all declared domains*, of the form _Listadom. For each domain, the following elements will be displayed: name, operator(s), neutral element(s), if case be, and type (for the latter there will be used the reserved keyword for domain type).
9. *Program termination*, of the form exit or _exit.

2. Designing the object-oriented algebraic hierarchy

The implementation of the structures within the algebraic hierarchy semigroup – monoid – group – commutative group – ring – (commutative ring, field – commutative field) is based on a three auxiliary modules, which implement:

- The representation of the expressions in a tree form and various operations on this form (the operations will be performed according to the axioms associated with the operators of the domain within which the operations are performed)
- User command validation
- Specific operations in the String class, used for identifiers, returned expressions, commands etc.

The module for expression storage and analysis contains functions for:

- The creation of the binary tree associated with an expression and of the multiple-descendants tree into which a binary tree containing associative operators can be converted
- Conversions between binary and ordinary forms of a tree
- Tree copying (necessary for the constructors of the classes describing expressions from various algebraic structures)
- The elimination of neutral and symmetrical elements from the tree form associated with an expression
- The conversion of an expression containing commutative operators to a canonical form, by lexicographically sorting the operands of each sub-expression
- The explicit application of the distributivity property
- The replacement of an expression identifier with a sub-expression represented in the tree form. This function will be used to evaluate, within various domains, expressions containing previously-assigned identifiers
- The generation of the external form associated with a tree representation, taking into account the possible existence of operators' priorities (as in the case of rings and fields)
- The process of freeing the dynamically allocated memory for tree representations.

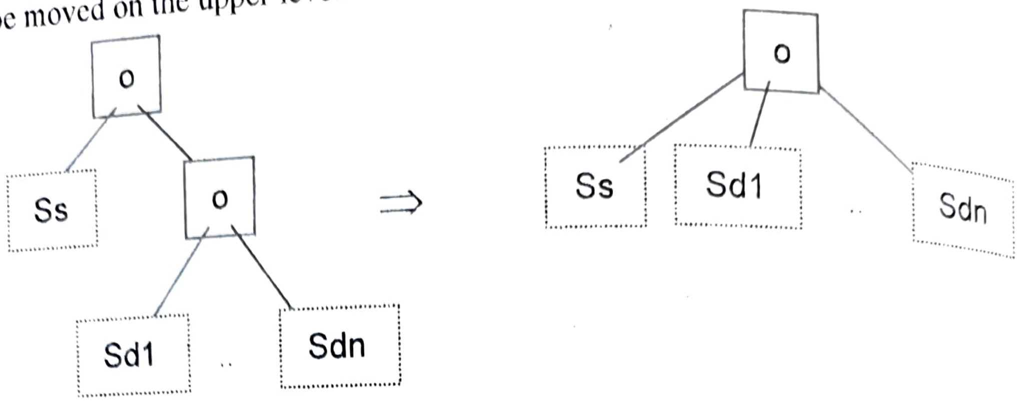
Pointer assignments are extensively used, in order to speed up the tree functions, especially for value replacements or eliminations.

An expression will be represented using a binary tree or, alternatively, an equivalent, ordinary one. The construction of the associated binary tree can be done in two steps: the first involves the storage of the expression's operands and operators together with their priorities (the information given by parentheses is stored into an array of priorities) and the second contains the recursive generation of the binary tree associated with the array of operands and operators by successively introducing the lowest priority operand into the current root. The transition from the binary representation to the n-ary representation (and conversely) can be done by taking into account the associativity of the operator(s) and transferring on the same level the descendants between which the aforementioned operation appears.

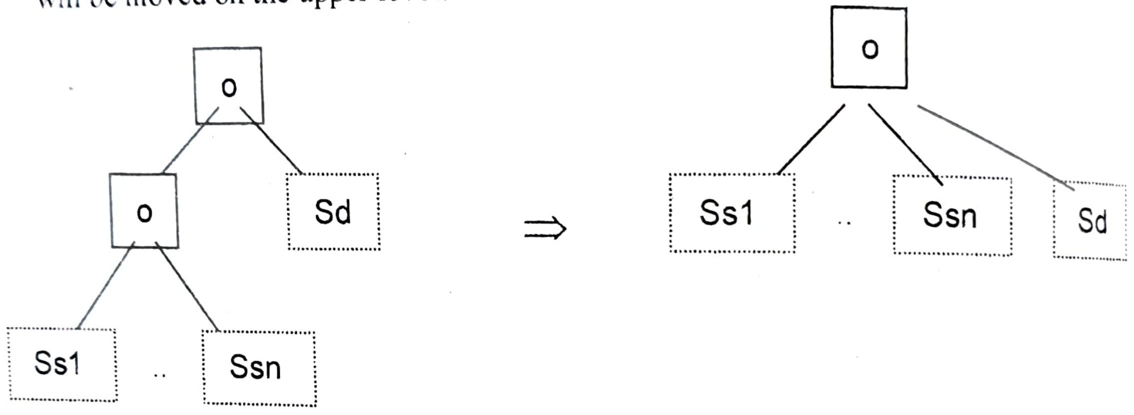
Thus, in order to obtain the internal n-ary tree representation of an expression over a domain with an associative operator o, we can distinguish the following cases, which will be applied recursively, from the terminal nodes (leafs) to the root, for a complete transformation

- if the sub-tree is terminal or doesn't have the operator o as root, a copy operation will be performed

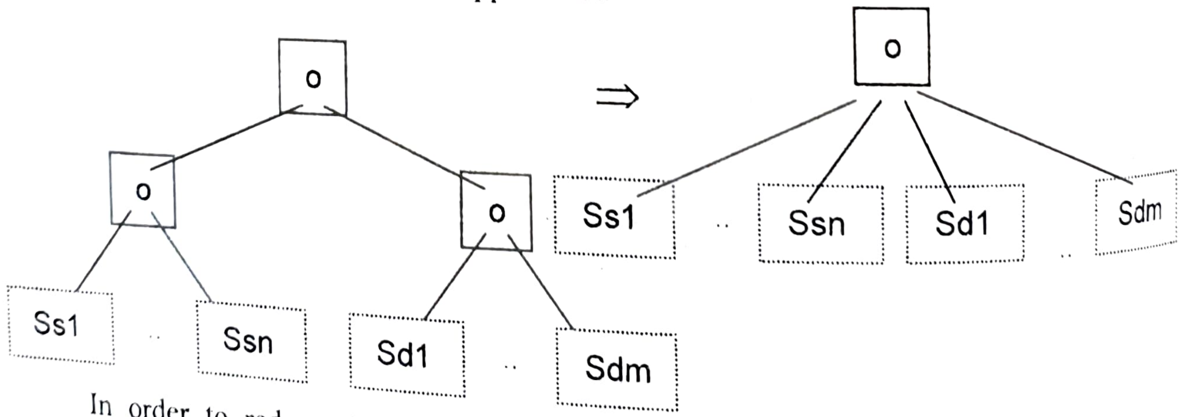
- if the operator *o* appears in the root and in the root of the right sub-tree, its sub-trees will be moved on the upper level:



- if the operator *o* appears in the root and in the root of the left sub-tree, its sub-trees will be moved on the upper level:



- if the operator *o* appears in the root and in the roots of its both sub-trees, all their sub-trees will be moved on the upper level:



In order to reduce the complexity of the program that implements the above-described algorithm, wherever possible, pointer assignments have been used, rather than subtree copying.

In the case of the structures which posses neutral element(s) (at least monoid), we need mechanisms for expression processing, which can apply operator properties for simplifying the expressions we operate on.

The elimination of neutral elements from the binary tree internal form of an expression can be performed by recursively applying (post-order) the following principles:

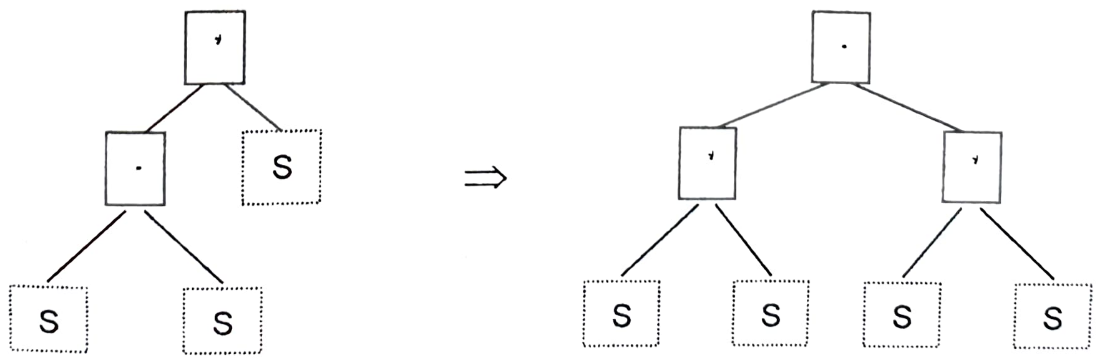
- if both subtrees are identical to the unit, the unit will be returned
- if the left subtree is identical to the unit, the right subtree will be returned
- if the right subtree is identical to the unit, the left subtree will be returned

The elimination of symmetrical elements from structures of (al least) group type can be done similarly, by visiting in postorder the tree that retains the internal form of the expression.

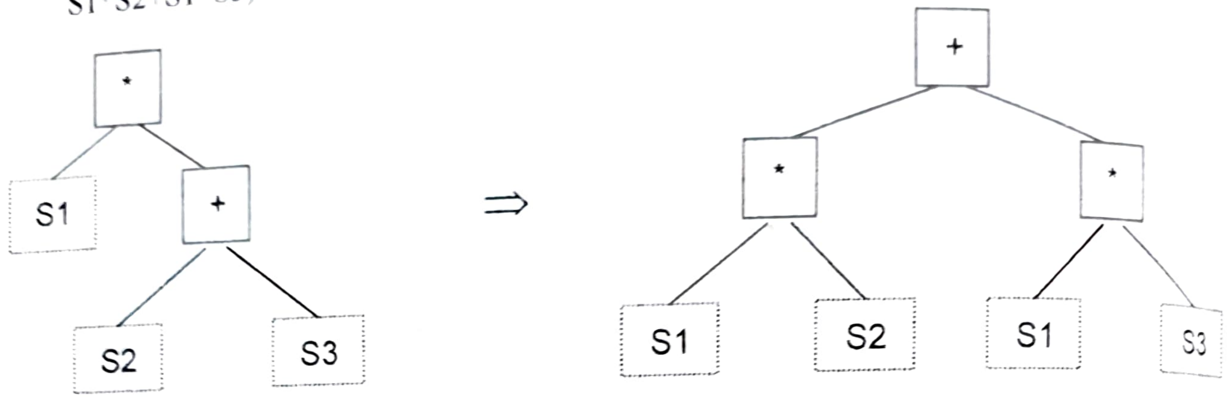
If the algebraic structure in which we operate possesses a commutative and invertible operator, it is necessary to generate a canonical form in which the operands from a sub-expression containing the same commutative operator are sorted lexicographically and, moreover, the properties of the symmetrical and neutral operands are applied. The problem of lexicographically ordering the operands is not a trivial one, as these can further contain sub-expressions. The sub-trees from the expression representation which correspond to operands will be sorted in such a way that those containing operators in their root will be placed at the end and those which contain identifiers defined as symmetrical elements will be placed immediately after their corresponding elements, such that the symmetrical elimination algorithm can be applied.

The property of distributivity in structures of ring type can be applied easiest on the binary tree representation, which can subsequently be converted to the *n*-ary tree representation, as shown earlier. Denoting by "+" the additive operator of the ring and by "*" the multiplicative one, the following cases need to be considered, recursively:

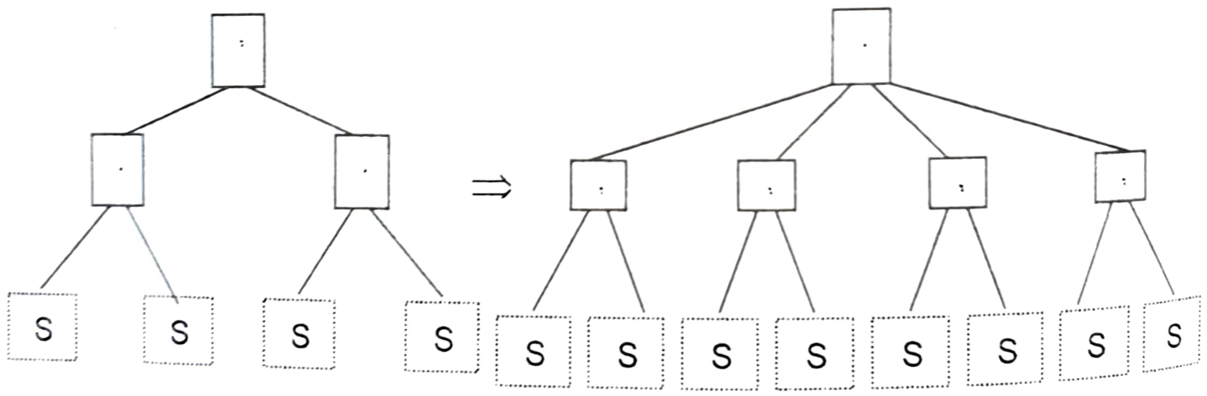
- if the root of the tree contains the multiplicative operator and the right subtree contains the additive one (in the form S1*(S2+S3)), then the resulting tree will contain in its root the additive operator, the multiplicative operator in the roots of each of its two subtrees, and the initial left subtree will appear as an operand in both of them (in the form S1*S2+S1*S3).



- if the tree root contains the multiplicative operator and the left subtree contains the additive one (in the form (S1+S2)*S3), then the resulting tree will have in the root the additive operator, the multiplicative one in the roots of each of its two subtrees and the initial right subtree will appear as an operand in both of them (in the form S1*S2+S1*S3).



- if the root of the tree contains the multiplicative operator and each of his two subtrees contain the additive one, then the resulting tree will correspond to a sum of four subtrees / subexpressions corresponding to the product formed from each two initial subtrees (in the form S1*S3+(S1*S4+(S2*S3+S2*S4))).



It is very important that the expressions represented within the system be easily extracted in an appropriate external format. For this purpose, the tree form corresponding to the internal representation of the expression is transformed into the equivalent parenthesis form, taking into account operators priorities (an important issue within ring type structures). Parentheses are introduced for subexpressions whose root contains an operator of a lower priority than the operator from the root of the entire tree.

In certain situations the user will find it useful to consult the internal representation of an expression – obviously, in an easily understandable form. Such a form can be obtained by representing the tree associated with the expression in a parenthesis form, of string type, which can be displayed at any moment. This feature for viewing the internal form [1] is also present in the classic symbolic calculus systems (*FullForm* in

Mathematica). Practically, the construction of such a form can be done by simply visiting the corresponding tree in preorder.

The *validation module* contains functions that verify the syntactical correctness of a command. The tests are done in a modular fashion, the base level containing functions for validating user identifiers, operators and reserved keywords that designate algebraic structures. On the next level there are functions which parse command's elements:

- strings representing declarations of domains with name, operators and possibly neutral elements, separated by commas (and delimited by parentheses)
- strings representing expressions – the corresponding function retains the expression's arrays of variables (this feature will be used for keeping track of all variables of a given domain) and operators (thus it is possible to verify whether undefined operators appear)

These verification operations are used by the function which validates a command (see the types describes earlier). The argument string is processed in accordance with its form, thus determining whether the command is syntactically correct and, if so, which is its type. Depending on the command's type, the following elements will be stored for later use:

- name, operators and possibly neutral elements – for domain declarations
- the variable which is assigned a value to, the variable list and the expression's operators – for variable assignment
- variables and operators lists, for expressions, equality and difference tests
- the two variables and possibly the operator with respect to which they are symmetrical (in the case of domains with more than one invertible operator) – for symmetrical element declaration.

The hierarchy of algebraic structures implements the categories of semigroup, monoid, group, abelian group, ring, field, abelian ring, abelian field as structures in which symbolic computation operations can be performed, without particularizing them into a given domain. These structures will appear in the hierarchy as special classes, used to describe the domain's name, operators, attributes, variables and generally available methods. For the classes corresponding to structures that have an invertible operator there are also defined lists of corresponding symmetrical elements. In this paper (and in the source text), the classes that describe the above-mentioned algebraic structures are named TipSemigrup, TipMonoid, TipGrup, TipGrupCom, TipInel, TipCorp, TipInelCom, and TipCorpCom, respectively. Obviously, between these classes there are inheritance relationships. Moreover, each class will be identified by a code (0..7) which creates a one-to-one mapping between the types of domains and the information stored for each domain within a special structure of array type (Tip). Pursuing the goal of processing the domains in a similar manner, a special (abstract type) class TipAbstract has been defined, which enables conversions from / to algebraic domain types (TipSemigrup, TipMonoid, TipGrup, TipGrupCom, TipInel, TipCorp, TipInelCom, respectiv TipCorpCom). Keeping a history of all defined domains is of special importance if the user wishes to return to a previously declared domain. For this purpose we use an array of pointers to abstract types (Tip); these pointers will later be converted into pointers to algebraic domains (of the types described above). The newly created objects will be stored in this array when executing a domain declaration command.

In order to represent objects belonging to algebraic structures, i. e. expressions declared within these structures (possibly assigned a user identifier), newly derived classes will be defined. These classes are named **Semigrup**, **Monoid**, **Grup**, **GrupCom**, **Inel**, **Corp**, **InelCom**, and **CorpCom**, respectively. Besides direct inheritance relations introduced between these classes, there are also inheritance relations (for member variables and methods) from superior order classes that describe the domains to which expression-objects belong. The classes for objects/expressions will describe the name of the variable associated to the expression, the string and the tree representing the expression (in canonical form). In order to be able to process objects of different types as abstract objects, a new class **ObiectAbstract** is defined; this class enables conversions from / to various expression types (**Semigrup**, **Monoid**, **Grup**, **GrupCom**, **Inel**, **Corp**, **InelCom**, and **CorpCom**, respectively). The object-oriented programming philosophy facilitates the process of freeing objects that correspond to expressions without an associated identifier and the storage of all others. This memorization process uses an array (**Obiecte**) of pointers to those objects (referred to as abstract objects from now on). This array will be updated whenever an assignment command is issued by the user. Moreover, the variables from any expression entered by the user (assigned or not) will be retained in the variable list associated to the corresponding domain (within the superior class), to be used later if the user wants to display the identifiers declared within a given domain. Among the methods specific to expression classes, besides constructors, destructors and assignment operators, there are also functions for displaying the result of the current expression (the canonical form processed according to the current's domain properties, in which assigned variables are replaced accordingly) and functions for processing the list of created objects (variables with an assigned value) from all the domains defined by the user. The process of initializing the pointer to each such instance of assigned expressions is performed whenever an assignment command appears.

The design principles presented above induce the implementation of the algebraic structures and their associated objects as a multiple inheritance hierarchy presented in fig. 1.

The problem of eliminating duplicate inheritances has been solved by using virtual inheritance, which is implemented using pointers.

Computation operations in structures of semigroup type are implemented using two classes, which model:

- general structures of semigroup type – the **TipSemigrup** class. This contains the name and the operator of the structure, its axioms (coded), the type code of the variables, as well as specific implementation methods. These include initialization and copy constructors (very important for parameters and results transfer), overloaded assignment operators, as well as display functions (for a given domain, there will be displayed its name, operator and type in a manner similar to a domain declaration), functions for displaying the list of variables belonging to the domain. One can observe that it is sufficient for the domain's type and properties to be stored in a single instance, as they are the same for every instance representing the same type of algebraic structure. We can anticipate a little by revealing that the **TipSemigrup** class will be the base of the entire multiple inheritance hierarchy that is described in what follows.

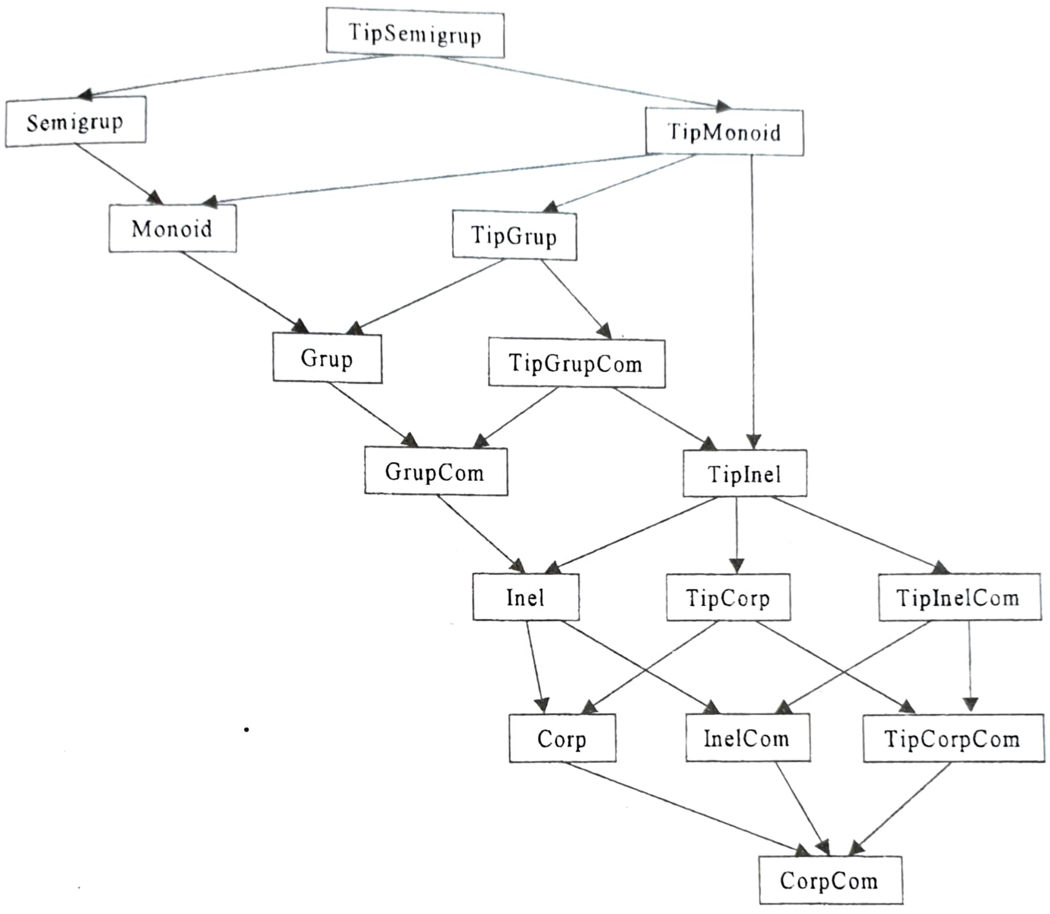


Figure 1

- Expression-objects belonging to a domain of semigroup type – the **Semigrup** class. This inherits the **TipSemigrup** class and contains as specific data elements describing an expression: the identifier associated with the expression (if the expression has been assigned a value to), the expression as a string and as a tree (as shown earlier, considering that the operator of the domain is associative, the expression can be represented as an *n*-ary tree, representation used for the internal operations), the position of the current object's pointer in the array (**Obiecte**) which retains the objects having an assigned value from all domains in a similar form, as shown in what follows, and specific methods for expression processing. These include, besides overloaded constructors, destructors and assignment/stream operators, operators used for the equality/difference tests (these tests are performed on the canonical form representations) and a function for displaying the result of the expression entered (the system works as an interpreter). To obtain the result of an expression we analyse its identifiers by visiting the associated tree, verifying if there are identifiers having assigned a value to. In this case, they are replaced with the corresponding sub-expression (by a function from the tree module that performs a pointer swapping), after which the tree form is transformed into a external one by visiting its nodes. In fact, this conversion function from the internal tree representation into the string representation is more complex, as it takes into account that priorities might be associated to the operators, in the case of multiple operator domains. One can observe that from the Semigrup class all other classes describing objects of expression type will be derived, directly or indirectly.

Computation operations in structures of monoid type are implemented using two similar classes, which model:

- general structures of monoid type – the TipMonoid class. This class inherits the TipSemigrup class and supplementarily contains the neutral element as well as specific constructors and destructor (which call those of the base class TipSemigrup).
- expression-objects belonging to a monoid type domain– the Monoid class. This class inherits TipMonoid and Semigrup classes and contains specific constructors / destructor. It also redefines the function that returns the result, eliminating neutral elements from expressions (this operation is performed on the tree form, which is later converted to the external form).

Structures of group type, as well as the expressions belonging to group type domains, are introduced pursuing the same principles, using the classes TipGrup and Grup, respectively. TipGrup class is directly derived from TipMonoid class and supplementarily introduces two arrays that store pairs of symmetrical elements for each domain of group type. In the Grup class, which is derived from TipGrup and Monoid, the method for returning the result is redefined. The new method reduces the pairs of symmetrical elements.

Computation operations in structures of abelian group type and expressions belonging to abelian groups are defined in a similar manner using TipGrupCom and GrupCom classes. The differences lie in the fact that the canonical representation of the expressions belonging to these domains takes into account the commutativity property of the domain's operator. This canonical representation is obtained by lexicographically sorting operands of each subexpression (within the associated tree representation, see tree module). This operation is performed within the function that redefines the external form from Grup class by using the canonical representation of abelian group domains.

Ring type structures are described within TipInel class, which is derived from TipGrupCom and TipMonoid classes; TipInel supplementarily introduces the multiplicative operator, together with its neutral element and properties. The constructors used to create ring type instances will obviously call the constructors from its base classes. Expressions belonging to ring type domains are created within Inel class, which is directly derived from TipInel and GrupCom classes, on the same principles as previous derivations. In order to generate the result of an expression belonging to a ring, there will be cancelled symmetrical elements in respect with the additive operator and neutral elements in respect with both operators. Obviously, we shall verify whether the expression contains previously assigned identifiers (therefore retained as pointers to abstract objects in the array of expression objects) and if so, they will be substituted in the tree form for the corresponding subexpressions and afterwards the external form will be generated (these operations are performed within the tree module). For expressions belonging to ring type domains and domains derived from these ones, the function that applies the distributivity of the operators is explicitly applied.

Computations in abelian ring type structures are performed similarly, using the classes that describe structures of this type – TipInelCom class and expressions over abelian rings –InelCom class. TipInelCom class inherits InelCom class and InelCom class is derived from TipInelCom and Inel. The difference in expression manipulation consists in the fact that the lexicographical sorting of operands is also applied for the multiplicative operator, therefore the function that generates results of expressions will be redefined accordingly.

The field type structure, described within TipCorp class, inherits TipInel class and supplementarily introduces the pairs of symmetrical variables in respect of the multiplicative operator. Expressions over field type domains are objects of Corp class, which is derived from TipCorp şi Inel classes and introduces specific manipulations of expressions, taking into account the invertibility of the multiplicative operator: in the tree form and within the function that generates the results of expressions there will also be cancelled the symmetrical elements in respect of the multiplicative operator.

Abelian fields are instances of TipCorpCom class, which inherits TipCorp class. Expressions over abelian field domains are instances of class CorpCom, derived from TipCorpCom and CorpCom classes. The manipulation of these expressions involves the lexicographical sorting of operands in respect of both operands (we consider that the multiplicative operator has a higher priority than the additive one). Other types of expression processings, such as the cancellation of neutral and symmetrical elements, are similar to the ones from the ascending classes.

In order to perform an efficient processing of the user's commands, it is necessary to be able to refer in a common manner to all the instances that represent abstract algebraic structures, respectively expressions over these domains.

The abstract type that all classes representing algebraic structures (TipSemigrup, TipMonoid, TipGrup, TipGrupCom, TipInel, TipCorp, TipInelCom, TipCorpCom) will be converted to, named TipAbstract, retains the encoded type of the structure (used in the conversion process), and an undefined (void) pointer, which enables type transformations. These conversions will be performed from the above mentioned classes to the new type, using conversion constructors, as well as from the abstract type class to algebraic domain classes, whenever an instance must be processed according to the type of the domain it represents. Such operations are necessary within the methods that process user's commands, which, on the one side, must be generally valid and, on the other side, must describe computations derived from the type of the current algebraic domain. Taking into account its implementation, it is necessary that TipAbstract redefine the specific methods of the algebraic domains by an explicit call of the appropriate ones, based on the type of the current domain. Obviously, the constructors, destructor and overloaded operators of **TipAbstract** class will use the appropriate methods defined within the hierarchy classes.

Moreover, the abstract type that denotes algebraic structures enables the memorization of user defined domains by using pointers to their instances (**Tip** array). This fact allows the user to "reload" a previous working context by redeclaring its domain. Domain memorization is performed by specific methods of **TipAbstract** class, which verify whether the current domain instance was already created and if not, retain it in the abstract pointer array.

Pursuing the same representation principles regarding expression-objects from various types of algebraic structures, these instances will be created by an abstract class (**ObiectAbstract**), which implements conversions from / to classes that describe expression over defined domains (**Semigrup, Monoid, Grup, GrupCom, Inel, Corp, InelCom, CorpCom**). Conversions are performed based on the type of domain the expression belongs to, using an undefined (void) pointer both for conversion constructors and explicitly, whenever an abstract object must be processed according to particular properties of an algebraic domain. The constructors, destructor, overloaded operators and the other methods specific to abstract expressions (for example, the function that returns the result of an expression or the equality and difference tests) explicitly call similar methods from the expression classes (**Semigrup, Monoid, Grup, GrupCom, Inel, Corp, InelCom, CorpCom**). It can be noticed that the elements of the domain a certain expression belongs to are also available using **ObiectAbstract** class, since all expression-object classes are derived from the ones that model corresponding algebraic structures.

The abstract object class induces a common manner of processing expressions from various algebraic domains within the methods that manipulate user commands (particular manipulations will be performed consequent to the appropriate type conversion of the expression), as well as the memorization of all user-defined subexpressions by using pointers to abstract expression instances. This memorization is essential for correct computation of expressions that contain previously assigned identifiers (as shown above) and is performed within expression processing, by specific methods of **ObiectAbstract** class. Variables will not be memorized twice, but if a new symbolic value is introduced for an identifier, we shall obviously retain the latter. Further references to these identifiers will take into account: the type of the current domain, the position of its instance in the array of domains and the index of the variable (retained while creating the expression object).

User commands will be interpreted until the termination command (exit) is encountered. Command processing is initiated by a function that validates the command (see validation module) by parsing it and retaining its type and elements. Based on the type of command, there will be created an abstract instance representing an algebraic structure or an algebraic expression, which will be processed. The declarative manner of introducing domains and the inheritance relations implemented between domain classes and expression classes enable us to use the instance generated by a domain declaration command as a general context for creating the following expression instances, until a new domain declaration.

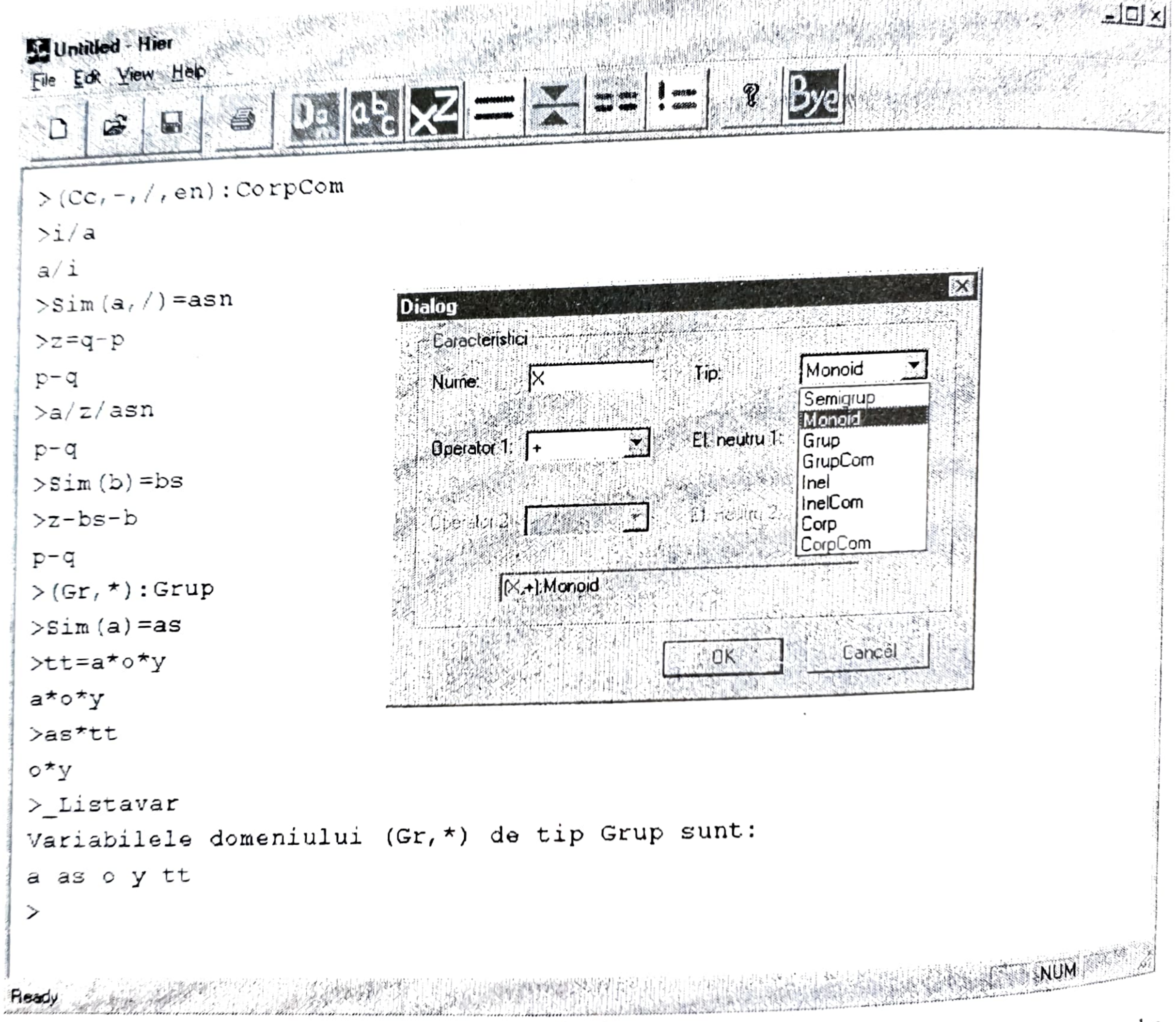
Command interpretation is performed as follows:

- for a domain declaration, an abstract (**TipAbstract**) instance will be created; this instance will retain, taking into account the domain type, a reference to an instance of the actual algebraic structure. The abstract instance will be retained in the domain array and will represent the current computing context (until the next domain declaration).
- for an assignment it is verified whether the operators belong to the current domain and if so, the expression is instantiated as an abstract object, which contains a pointer to the instance of the actual type expression, derived from **Semigrup** class. The abstract object is memorized in the expression array. An assignment command will generate the result of the symbolic expression evaluation by calling the evaluation method for the abstract object, which makes use of specific evaluations described in various expression classes (these ones operate upon the tree form of the expression).
- an expression command is processed similarly, except the memorization of the expression; therefore, after generating the result, the object may be destroyed.
- symmetrical element declarations involve the actualization of the current domain instance (retained as an abstract domain) by modifying its list of symmetrical variables. This list will be accessed in a specific manner for different types of domains since for domains with two operators the list of symmetrical elements in respect of the second operator is independent from the first one. If the specified operator (or implicitly assumed, for domains with one operator) is not invertible, an error message will be displayed.
- for equality and difference tests it is verified whether the operators are correct and if so, objects for each member of the equality / inequality an abstract expression is created. These two objects will be applied == and != overloaded operators, which use the similar operator defined in the basic expression (**Semigrup**) class and inherited by the other expression classes. Obviously, the tests take into account previously assigned identifiers, as well as the axioms of domain operators (commutativity, associativity), and generate a boolean result (**True** / **False**).
- the domain display command uses the domain array by means of a specific display method that generates an external form for each domain; this form includes its name, operator(s), neutral element(s) and type.
- the variable display command works on the current domain by displaying its variable list (the method defined for abstract expressions explicitly calls the corresponding ones from specific expression classes).
  - the **exit** command terminates the cycle that processes user commands.

3. The user interface

The system's interface module is created by using the applications available for this purpose within Developer Studio for Visual C++ 5.0 [8] environment. The user will interact with the application by means of a Simple Document Interface (SDI) – see fig. 2, which was designed by using Application Wizard.

Within the document window (see fig. 2), the user can input commands from the keyboard after the prompter character; each command will be interpreted according to the above described rules. A more accessible possibility to introduce the commands is to use the corresponding buttons, which assist the user in specifying: domain declarations, variable / domain view requests, assignments, symmetrical declarations, equality / difference tests or finishing the computation (see fig. 2). For each button used to read elements of commands, a dialog window (Dialog Based Interface) was created; the latter contains specific controls, associated with the parameters of the command. The general frame for defining the controls was created by using Class Wizard, the most frequently used ones being edit boxes, drop down and drop lists. After specifying all the necessary elements, the command created within the dialog box is shown in the document window, validated and interpreted.



```
>(Cc,-,/,en):CorpCom
>i/a
a/i
>Sim(a,/)=asn
>z=q-p
p-q
>a/z/asn
p-q
>Sim(b)=bs
>z-bs-b
p-q
>(Gr,*):Grup
>Sim(a)=as
>tt=a*o*y
a*o*y
>as*tt
o*y
>_Listavar
Variabilele domeniului (Gr,*) de tip Grup sunt:
a as o y tt
>
```

Nume: X
Tip: Monoid
Operator 1: +
El. neutru 1:
Operator 2:
El. neutru 2:
(X,+):Monoid

Fig. 2

For domain declarations, dialog controls are activated / deactivated based on the domain type, so that a consistent declaration is obtained: for semigroups, the user may specify only the operator, for monoids and groups – the operator and, possibly, its neutral element, for rings – both operators and the first neutral element, for fields – both operators and both neutral elements.

The content of the document window (input commands and their results) is processed using a string array that ensures a correct display of the window content after resizing operations, as well as the possibility of saving the current working context into a text file, which can be later reloaded – for this purpose, the implicitly created buttons

New, Open, Save are used. The actions associated with these buttons were designed according to archiving and serializing principles that are implemented within Visual C++'s Document / View architecture.

4. Conclusions

The software system described in the paper belongs to the research area of symbolic computation systems based on type theory; it implements symbolic computations within the algebraic hierarchy containing semigroups, monoids, groups, abelian groups, ring, abelian rings, fields and abelian fields. The system's design principles rely on an object-oriented modeling of the relations between the above mentioned structures; in fact, the whole type theory was influenced not only by the algebraic category theory, but also by object-oriented programming principles [3]. The practical impact of the system specialized in symbolic computations within abstract algebraic structures is sustained by an accessible, visual-like user interface.

REFERENCES

1. Alina Andreica – Aplicaøii ale calculului simbolic, Referat de doctorat, 1996.
2. Alina Andreica – A Possibility to Describe an Algebraic Hierarchy, Studia Informatica, 1, 1997.
3. Alina Andreica – Type Systems in Symbolic Computation, Preprint Computer Science, forthcoming.
4. J. H. Davenport, B, Trager – Scratchpad's View of Algebra I : Basic Commutative Algebra, DISCO 1990, Springer Verlag 1990.
5. J. H. Davenport – The Axiom System, Axiom Technical Report TR5, 1992.
6. Liviu Negrescu – Limbajul C++, vol 2, Ed. Microinformatica, 1994
7. Bujor Silaghi – Din tainele programării în C++, Ed. Microinformatica, 1996.
8. Mickey Williams – Bazele Visual C++, Ed. Teora, 1998.

"Babes-Bolyai" University, Cluj-Napoca, Faculty of European Studies

Technical University Cluj-Napoca, Faculty of Automation and Computer Science