STUDIA UNIV. BABES-BOLYAI, INFORMATICA, VOLUME II, NUMBER 1, MARCH 1997

A POSSIBILITY TO DESCRIBE AN ALGEBRAIC HIERARCHY

ALINA ANDREICA

Abstract We propose a simple way of describing in Mathematica the semigroup - monoid - group - abelian group - ring - (field, abelian ring - abelian field) hierarchy by specifying the properties of each structure and a few operations which characterize them. These ideas could be the starting point for constructing a symbolic computation module to manipulate abstract domains.

1. Introduction

The symbolic computation systems (SCS) have been successfully used during the last decades in order to obtain quickly the desired results in symbolic computation sequences which involve series, limits, differentiations, integrations, etc. There were two directions that marked SCS evolution: designing complex, general-purpose systems or designing specialized systems, meant to deal with calculations characteristic to certain domains, such as: astronomy, quantum mechanics, relativity, etc. [1]. The most popular systems (Reduce, Maple, Mathematica, whose performances are studied in [2], together with Macsyma and Derive) contain complex algorithms for a large variety of calculations but they deal only with common algebraic domains: integer, rational, real and complex domains and do not provide the possibility of working with other types of algebraic domains. Host languages vary from Lisp in Reduce (A. C. Hearn, early '80s) to C in Maple (created at the Waterloo University, Canada in the late '80s, early '90s).

Therefore the direction consacrated to the definition of abstract types of domains (which would be particularized by the ones mentioned above) appeared in the theory of symbolic computation as a natural phenomenon. The most well-known SCS based on these ideas is AXIOM (R. Jenks, R. Sutor, 1992), but it was preceded by Scratchpad, based on the Lisp language (R. Jenks, early '80s). Although AXIOM is an outstanding accomplishment, some authors criticize it for being too rigid, therefore not very convenient for some cases. The proposed alternative would be to create more flexible modules, capable of describing abstract domains, which should be attached to the commonly used SCS [3]. As a research direction, we intend to build such an extension for Mathematica.

The study of implementing abstract types of structures is based on the theory of domains and categories. We present here the essence of these concepts according to [3].

Domains are parametrized types; the parameters can be values or other domains. They can be created in a SCS by a function/procedure. Categories are abstract

Received by the editors: October 5, 1997.
1991 Mathematics Subject Classification. 68Q40.
1991 CR Categories and Descriptors. I.1.2 [**Algebraic Manipulation**]: Algorithms - *algebraic algorithms.*

parametrized types, the parameters can also be values or domains. They can be shared by many domains and provide a code which is independent of data representation.

For example, if we consider the *Set* as a category, we can write a function/procedure to create a set domain M. We must also provide the possibility of introducing some operations on M, the most common ones being the tests for equality and non-equality.

Developing our example, an Euclidean domain E would be a new category, created by a function / procedure and for which at least the division remainder and gcd operations would be defined. Describing the Euclidean algorithm within this context is an example of enriching the abstract structures with specific algorithms. It is generally recommendable that the corresponding algorithm of the host SCS would be the one implicitly used, the newly defined algorithm being only its extension for more abstract cases.

The algebraic semigroup - monoid - group - abelian group - ring - (field, abelian ring - abelian field) hierarchy, which is the object of this paper, was described using only representation algorithms, since the above mentioned domains do not have specific abstract algorithms. The technique of its defining is further described; the purpose of the paper is to propose elementary means of building this hierarchy using a Mathematica package.

2. Implementing the Mathematica package

The semigroup - monoid - group - abelian group - ring - (field, abelian ring - abelian field) hierarchy is well-known [4]. We intend to implement it using a Mathematica package which will be able to build one ore more groups, one or more rings, etc., with different names. Moreover, on any of these domains we must have the possibility to perform elementary operations, such as applying an operator upon two not necessarily elementary operands (the result is to be obtained as a concatenation of the operands as strings), testing equality and non-equality of two operands, verifying that the operands of an expression belong to the domain, explicitly applying some properties. All operations performed within a domain will regard the properties of the corresponding operator: associativity, commutativity or distributivity.

The operator characters which can be used are defined as members of the list *Operatii* = {"|", "^", "+", "*", "$", "-", "/", "!", "#"}. As most of these characters have built-in meanings and properties in Mathematica, which are not convenient to our purposes, we adopted a technique inspired from Mathematica's internal representation in order to correctly manipulate abstract expressions. We wrote a procedure which generates an internal type form of the expression, similar to *FullForm* from Mathematica [5] and another procedure which obtains the external string-type form (of an internal type form).

Obviously, the equality "=" and non-equality "<>" tests will be performed upon the internal form because once the operators were associated their properties (*Flat, Orderless*), the built-in Mathematica functions *SameQ* and *UnsameQ* [5] can be applied.

To every operator character we associate a name used in the internal type form, for example "+" - "Oplus", "*" - "Ostea", etc. This association uses a list of names (corresponding to the list of operators) and two functions which provide the name of an

operator, respectively the character associated to a name. To allow automatic application of the properties corresponding to the attributes, the internal form heads will be symbols (results of *NumeSimb* function).

The internal form, built by *FForm* function, looks like *Operator_name[operands_list]*, similar to *FullForm* from Mathematica[5]. In our case, the abstract operators from the above-mentioned hierarchy are binary. *FForm* function follows the basic idea from constructing a binary tree associated to an arithmetic expression; in order to make the transformation more general, valid for expressions containing parenthesis, this case has also been dealed with, as well as the situation when operators have different priorities.

First, *CrSir* function, whose input is a string expression, retains its operands and operators into a vector *Sir* with *dim* elements, eliminates the parenthesis and introduces the priorities of each element from *Sir* into a vector *Pr*.

```
CrSir[s_String]:=Module[{i=1,j=0,pr=0,n=0,x,c},
    n=StringLength[s];
    While[i<=n,c=StringTake[s,{i,i}];
        Which[c==" ",i++,
            c=="(",pr+=10;i++,
            c==")",pr-=10;i++,
            ELitera[c],
                x="";While[ELitera[c]&&i<=n,x=x<>c;If[i<n,i++;
                    c=StringTake[s,{i,i}],i++]];
                Sir[++j]=x;Pr[j]=pr+100  ,
            Operator[c],Sir[++j]=Nume[c];Pr[j]=pr+prio[Nume[c]];i++]];
                dim=j]
```

Afterwards, *CrForm* recursive function looks for the smallest priority element between two indexes in *Sir* and introduces it as head of expression with the arguments obtained from the two remaining subintervals.

```
CrForm[i_Integer,j_Integer]:=Module[{mn,k,ind},
    Which[
        i>j,Return[],
        i==j,If[MemberQ[Nm,StringDrop[Sir[i],1]],
            Return[NumeSimb[Sir[i]]],Return[Sir[i]]],
        i<j,For[mn=Pr[i];k=i;ind=i, k<=j, k++,
            If[Pr[k]<mn,mn=Pr[k];ind=k,]];
            Return[
        Apply[NumeSimb[Sir[ind]],List[CrForm[i,ind-1],CrForm[ind+1,j]]]]]]
```

The internal form is obtained by combined application of the two above described functions upon the expression string in which the blanks have been removed (the function that eliminates the blanks distinguishes non-significant blanks from blanks representing multiplication). Previously, attributes specific to *Plus* and *Times* are removed in order to avoid implicit application of commutativity or associativity.

```
FForm[expr__]:=Block[{Attributes},ElimAtrib;
```

```
Return[CrForm[1,CrSir[ElimBlank[ToString[expr]]]]]]
```

OForm function brings back an expression given in the internal form, as an argument, to the external string-type form. The function is recursive, based on the principle that *FForm* forms are binary and takes into account the case of operators with different priorities.

```
OForm[e_]:=Module[{o1,o2,x,y},
    If[AtomQ[e], Return[ToString[e]],
        o1=Level[e,1][[1]]; o2=Level[e,1][[2]];
    If[prio[Head[e]]<=prio[Head[o1]] || AtomQ[o1],
        x=OForm[o1], x="("<>OForm[o1]<>")"];
    If[prio[Head[e]]<=prio[Head[o2]] || AtomQ[o2],
        y=OForm[o2], y="("<>OForm[o2]<>")"];
    Return[x<>ToString[Car[Head[e]]]<>y]]]
```

In order to obtain a correct form for all operators, we also use the auxiliary function *Tr* which transforms a *FullForm* argument into the new internal type *FForm* expression. This function is necessary because in Mathematica there are more than one built-in operators with the same *FullForm* head, for example /, !, *, # have the same head *Times and* +, - have the head *Plus* [5]. This problem is not trivial, since a *FullForm* form can have an arbitrary number of arguments, whereas *FForm* has only two and there are internal forms which use heads specific to other operations. For example, *FullForm[a-b-c]* is *Plus[a, Times[-1,b], Times[-1,c]]* but has the *FForm* form *Ominus[a, Ominus[b,c]], FullForm[a/b/c]* is *Times[a, Power[b,-1], Times[c,-1]]* but *FForm* is *Oslash[a, Oslash[b,c]], FullForm[a#b#c]* is *Times[a, b, c, Power[Slot[1], 2]* but *FForm* is *Odiez[a, Odiez[b,c]]*.

The significant part of internal form arguments was retained in the elements of an array *T*, which will be used by *Aux* function to generate the form *Head[T[1], Head[T[2], ... , Head[T[n-1],T[n]]...]]*, using *TrAux* auxiliary function. *TrAux* recursively comutes the two arguments in the form generated by *Fold* [5] built-in function TrAux[Fold[f,T[n],Table[T[i],{i,n-1,1,-1}]]].

```
Tr[e__]:=Module[{c,x,y,p=0,n=0,T,i=0,h},
    If[AtomQ[e],e,
    c=ToString[Head[e]]; x=e[[1]];
    h=NumedinFF[c];
    n=Length[e];
    If[n>=2, y=e[[2]],];
    Which[c=="Power",
        Return[Apply[Ocaciula,List[Tr[x],Tr[y]]]],
            c=="Times",
        Which[
            MatchQ[y,Power[__,-1]],
                T[1]=Tr[x];
                Do[If[MatchQ[e[[i]],Power[__,-1]],T[i]=Tr[e[[i]][[1]]],],
                    {i,2,n}];
                Return[Aux[Oslash,T,n]],
```

```
        MatchQ[e[[n]],Slot[_]] || MatchQ[e[[n]],Power[Slot[_],_]],
            T[1]=Tr[x];
        Do[T[i]=Tr[e[[i]]],{i,2,n-1}];
            Return[Aux[Odiez,T,n-1]],
        MatchQ[y,Factorial[__]],
            Do[If[MatchQ[e[[i]],Factorial[__]],T[i-1]=Tr[e[[i]][[1]]],],
                {i,2,n}];
            T[n]=Tr[x];
            Return[Aux[Oexclam,T,n]],
        MatchQ[y,_],
            T[1]=Tr[x];
            Do[T[i]=Tr[e[[i]]],{i,2,n}];
            Return[Aux[Ostea,T,n]]],
            c=="Plus" || c=="Alternatives",
    Which[
        MatchQ[y,Times[-1,__]] && c=="Plus",
            T[1]=Tr[x];
            Do[If[MatchQ[e[[i]],Times[__,-1]],T[i]=Tr[e[[i]][[2]]],],
                {i,2,n}];
            Return[Aux[Ominus,T,n]],
        MatchQ[y,_],
            T[1]=Tr[x];
            Do[T[i]=Tr[e[[i]]],{i,2,n}];
            Return[Aux[NumeSimb[h],T,n]]]]]]
```

The procedure which defines a semigroup contains the definition of applying the operator upon two not necessarily elementary operands, the equality and non-equality tests, an operation which allows the explicit applying of associativity upon *FForm* form and *"corect"* operation, which verifies whether all symbols which appear in the argument expression belong to the domain. Moreover, to the corresponding operator we associate the attribute *Flat* (which specifies associativity).

In order to perform *"corect"* operation, we introduced all the symbols that were used in a domain *(D1,*)* or *(D2,+,*)* into lists of symbols refered by *Simb[D1,"*"]*, respectively *Simb[D2, "+", "*"]*. We also retained the names of the defined domains into the *LD* list, as well as neutral elements and corresponding type for each domain. The verification whether the symbols which belong to an expression were already used in the domain is done, depending on the number of operations in the domain (1 or 2), by:

```
ApDom1[e__,D_,op_]:=Module[{f,x,i},
    f=FForm[e];x=Union[Table[Sir[i],{i,dim}]];
    Return[Equal[x,Intersection[x,Append[Simb[D,op],Nume[op]]]]]]
ApDom2[e__,D_,o1_,o2]:=Module[{f,x,i},
    f=FForm[e];x=Union[Table[Sir[i],{i,dim}]];
    Return[Equal[x,Intersection[x,Join[Simb[D,o1,o2],
        {Nume[o1],Nume[o2]}]]]]]
```

We give below the body of the function which defines a semigroup.

```
Semigrup[S_,op_]:=Module[{f,ffa,ffb,i,x},
If[Operator[op],
    LD=Append[LD,S];
    TipDom[S,op]="Semigrup";
    Oper[S]=op; Simb[S,op]={};
    f=NumeSimb[Nume[op]]; prio[f]=1;
    If[Length[Attributes[f]]==0,
        r=Attributes[f]={Flat, Listable};Print[1],
        r=Attributes[f]=Union[Attributes[f],{Flat, Listable}]];
    S[op,a__,b__]:=Module[{x,y},
        ffa=FForm[a];
        Simb[S,op]=Union[Simb[S,op],Table[Sir[i],{i,dim}]];
        ffb=FForm[b];
        Simb[S,op]=Union[Simb[S,op],Table[Sir[i],{i,dim}]];
        x=Tr[FullForm[a][[1]]];
        y=Tr[FullForm[b][[1]]]; Print["=",Attributes[f]];
        If[op=="$",OForm[x]<>" $ "<>OForm[y],
            OForm[Apply[Nume[op],List[x,y]]]]];
    S["=",a__,b__]:=SameQ[FForm[a],FForm[b]];
    S["<>",a__,b__]:=UnsameQ[FForm[a],FForm[b]];
    S["corect",ex__]:=ApDom1[ex,S,op];
    S["asoc",expr__]:=Module[{f,x},
        f=FForm[expr]; x=Head[f];
    If[Depth[expr]<=2, Return[expr],
        Which[MatchQ[f,x[x[a__,b__],c__]],
                Return[x[S["asoc",a],x[S["asoc",b],S["asoc",c]]]],
            MatchQ[f,x[a__,x[b__,c__]]],
            Return[x[x[S["asoc",a],S["asoc",b]],S["asoc",c]]]]],
                Fail]]
```

Thus, in order to define a semigroup X, with the operator *"*"*, we call *Semigrup[X,"*"]*, while a sequence of operations on X could be, for example:

m=X["", a, b]*

n=X["", b, c]*

X["", m*c, a*n]* will generate *True*.

For a group, the basic operations from a semigroup are "inherited" by the explicit call of the corresponding procedure; this method is applied repeatedly for all other domains. For monoids and groups, the neutral element is stated as an argument in the call of the procedure which creates the domain we need; for a ring or field, both operators and both neutral elements are specified.

The supplemental commutativity property (*Orderless*) is introduced in the body of the functions which describe domains with this property and the explicit application of distributivity appears as an operation in the ring procedure. Within the definition of a

ring or field domain, the monoid operator was chosen to have a bigger priority than the group operator.

```
Monoid[S_,op_,e_]:=Module[{},
    Semigrup[S,op];
    TipDom[S,op]="Monoid";ElN[S]=e;Simb[S,op]={e};
    S[op,e,a__,b__]:=If[a==e, b, If[b==e, a, S[op,a,b]]]]

Grup[S_,op_,e_]:=Monoid[S,op,e];TipDom[S,op]="Grup";
    S["esim",a__,b__]:=SameQ[FForm[S[op,a,b]],e] &&
        SameQ[FForm[S[op,b,a]],e]

GrupCom[S_,op_,e_]:=Block[{f,Attributes},
    f=NumeSimb[Nume[op]];
    Attributes[f]={Orderless};
    Grup[S,op,e];TipDom[S]="Grup comutativ"]

Inel[S_,plus_,stea_,e0_,e1_]:=Block[{},
    prio[NumeSimb[Nume[plus]]]=1;
    prio[NumeSimb[Nume[stea]]]=2;
    Grup[S,plus,e0];
    Monoid[S,stea,e1];
    Simb[S,plus,stea]=Union[Simb[S,plus],Simb[S,Stea]];
    TipDom[S,plus,stea]="Inel";
    S["corect",ex__]:=ApDom2[ex,S,plus,stea];
    S["distrib",expr]:=OForm[Distribute[FForm[expr], Nume[e1], Nume[e0]]]]

InelCom[S_,plus_,stea_,e0_,e1_]:=Block[{f,Attributes},
    f=NumeSimb[Nume[stea]];
    Attributes[f]={Orderless};
    Inel[S,plus,stea,e0,e1]]

Corp[S_,plus_,stea_,e0_,e1_]:=Block[{},
    prio[NumeSimb[Nume[plus]]]=1;
    prio[NumeSimb[Nume[stea]]]=2;
    Grup[S,plus,e0];
    Grup[S,stea,e1];
    Simb[S,plus,stea]=Union[Simb[S,plus],Simb[S,Stea]];
    TipDom[S,plus,stea]="Corp";
    S["corect",ex__]:=ApDom2[ex,S,plus,stea];
    S["distrib",expr]:=OForm[Distribute[FForm[expr], Nume[e1], Nume[e0]]]]

CorpCom[S_,plus_,stea_,e0_,e1_]:=Block[{f,Attributes},
    f=NumeSimb[Nume[stea]];
    Attributes[f]={Orderless};
    Corp[S,plus,stea,e0,e1]]
```

3. Conclusions

The subject of the paper belongs to the direction of applying symbolic computation principles for abstract domains; it provides an example of constructing a simple domain hierarchy using a Mathematica package.

By describing the semigroup - monoid - group - abelian group - ring - (field, abelian ring - abelian field) hierarchy we experimented a few techniques which might be used for a future extension of Mathematica that would allow manipulating abstract domains.

REFERENCES

[1] B. Buchberger, G. E. Collins, R. Loos, R. Albrecht (ed.), *Computer Algebra and Symbolic Computation*, Springer Verlag, 1982

[2] D. Harper, C. Wolf, D. Hodgkinson, *A Guide to CA Systems*, Wiley Ed., 1991

[3] A. Miola (ed.), *Design and Implementation of Computer Algebra Systems*, DISCO '93, p. 81-94, 122-133, 177-191.

[4] I. Purdea, Gh. Pic, *Modern Algebra Treaty*, 1st vol., 1977

[5] S. Wolfram, *Mathematica*, 1992

"Emil Racoviță" Theoretical High School, Cluj-Napoca, Romania.
E-mail address: Ghergari@hera.ubbcluj.ro