

## Finite Structure Software, Application to Algebra

**Introduction:** the software described in this paper is an interpreter for a language that allows to create an arbitrary finite many-sorted structure and evaluate monadic second order expressions in the structure. The paper describes an application of the software to algebra - a library of commands that allow:

1. adding to a structure of algebraic objects: groups, binary operations, elements of groups. In particular, groups  $\mathbb{Z}/n$ ,  $S_n$  and product groups will be discussed.
2. extending the language with functions on the algebraic objects.
3. defining algebraic notions such as centralizer, normalizer, subgroup, identity, inverse, normal subgroup, etc.

## Using the Software

### Command Language

In the following description by a bracketed expression we will mean a set of characters, i.e.  $[a-z]$  is a set of letters  $a, b, c, \dots, z$ .

A sequence of expressions enclosed in parenthesis or an expression followed by  $*$  (star symbol) means repeated 0 or more times.

A symbol  $'|'$  will denote or. For instance  $+ | -$  means plus or minus.

A sequence of character in quotes is to be understood literally.

By  $'\backslash'$  we will mean a quote character.

By  $.$  (dot) we will mean any character. Hence  $.*$  means 0 or more characters.

expression = simpleExpression (operator simpleExpression)\*.

That is an expression is a simpleExpression followed by 0 or more clauses consisting of operator and simpleExpression.

simpleExpression = identifier | string | list | set | functionCall | "

operator = '+ | '-' | '\*' | '&&' | '||' | '!' | '=' | '- >' | '^' | '<' | '<='

These are plus, minus, multiply, logical and, logical or, logical not, assignment, equality, implication, power operators. The operator precedence is in the following order:

$\wedge, *, +, -, ==, <, <=, !, \&\&, ||, - >$ .

identifier=[a-z,A-Z][a-z,A-Z,0-9,\_]\*.

That is an identifier is a letter followed by 0 or more letters, digits or underscores.

string = "'\*.!' | '\'.\*!'

That is a string is a sequence of characters enclosed in double quotes, or a sequence of characters beginning with a single quote and ending with a double quote. To denote a

quote character or a double quote character within a string one may use `\'` or `\"`, that is slash-quote, slash-double-quote.

```
list='(' expression (',' expression)* ')'  
set = '{' expression (',' expression)* '}'
```

That is a list and a set is a sequence of expressions separated by a comma enclosed into parenthesis or curly brackets.

functionCall = identifier list.

That is functionCall is of the form: functionName(arg1, arg2, ..., argn).

### Data types, objects and identifiers.

The basic data type of the software are objects. Objects may be of various sorts: in particular strings, ordered sets (lists), sets, graphs, integers, groups.

The software starts with a set of integers from 0 to 63.

Each object has an id. Thus an id of integer 5 is a string "5".

When a name - an identifier or a string is encountered the software attempts to resolve it to an object:

by first checking a list of constants. If a constant with the specified name is not found, the software checks whether an object with the id matching the name exists.

For integers this means that only integer from 0 to 63 can be referred by their id. However, the software will generate larger integers if necessary in the computation. Thus, while 400 cannot be referenced by name the following is valid:

```
>20*20  
'400"
```

### Variable definition:

A variable definition, further referred to as varDef is a list consisting of two elements: variable name and sort definition. A sort definition may be a sort name, a set, a list, or a group.

```
varDef = '(' variableName ',' sortDefinition ')'
```

A list of variable definitions will be referred to as varDefList.

### FA command.

FA command implements "for all" functionality. Its syntax is:

```
FA(varDefList, expr).
```

The command generates a set of tuples as defined in the varDefList. For each tuple, the function assigns to each variable in the varDefList the corresponding elements of a tuple and then evaluates expr with this variable assignment. The command evaluates to TRUE if the expression expr evaluates to TRUE for every variable assignment. Otherwise the command evaluates to FALSE.

Example:

$\forall x \in \text{integer } x + 0 = x.$

```
>FA((x, integer), x+0==x)
'TRUE' [object]
```

#### TE command

TE command implements "there exists" functionality. Its syntax is:

TE (varDefList, expr).

The command evaluation is similar to that of FA command, except that TE evaluates to TRUE if expr evaluates to TRUE for at least one variable assignment. If expr evaluates to FALSE for all the variable assignments, then TE evaluates to FALSE.

Example:

$\exists x \in \text{integer } \forall y \in \text{integer } x + y = y$

```
>TE((x, integer), FA((y, integer), x+y==y))
'TRUE' [object]
```

#### Q command

The syntax is:

Q(varDefList, expr).

The command is similar to FA and TE commands. It returns a list of tuples if varDefList contains more than one varDef, and a list of values if varDefList contains one varDef. The tuples or values returned are those for which expression expr evaluates to TRUE.

Example:

$v_1 \in \text{integer} : 1 < v_1 \wedge \forall v_2 \in \text{integer } \forall v_3 \in \text{integer } (v_1 = v_2 \cdot v_3 \rightarrow v_2 = 1 \vee v_3 = 1)$ , which should produce a list of primes contained in the set "integer" = [0, 63].

```
>Q((v1, integer), 1<v1 && FA((v2, integer), FA((v3, integer), v1==v2*v3 -> v2==1
|| v3==1)))
('2", '3", '5", '7", '11", '13", '17", '19", '23", '29", '31", '37", '41", '43",
'47", '53", '59", '61")
```

#### DO command

The syntax is:

DO(varDefList, expr1, expr2).

The command is similar to the Q command. For each tuple as described by the varDefList, if expr1 evaluates to TRUE, expr2 is evaluated and the result of evaluation is added to the result list. We will use a previous example to generate a list of integers from 0 to 63 which are one greater than primes:

```
>DO((v1, integer), 1<v1 && FA((v2, integer), FA((v3, integer), v1==v2*v3 -> v2==1
|| v3==1)), v1+1)
('3", '4", '6", '8", '12", '14", '18", '20", '24", '30", '32", '38", '42", '44",
'48", '54", '60", '62")
```

#### fdef command

fdef command allows to define functions. Its syntax is:

`fdef(fname, varDefList, str).`

The command creates a function with the name `fname`, which inputs are described in the `varDefList`. The value of the function for arguments  $(x_1, \dots, x_n)$  is determined by assigning the arguments to the corresponding variables in the `varDefList` and evaluating the expression contained in the string `str`.

Example: we will define a function that will check whether an integer is a prime:

```
>fdef(prime,((n,integer)), "1<n && FA((x,integer),(y,integer)),x*y==n -> x==1 ||
y==1)")
'TRUE"[object]
>prime(12)
'FALSE"[object]
>prime(11)
'TRUE"[object]
```

powerSet function.

Power set function generates a power set of a set. Its argument may be a set, a list, a group or a graph.

Example: the power set of a set  $\{1, 2, 3\}$

```
>powerSet({1,2,3})
{{}, {'1'}, {'2'}, {'1', '2'}, {'3'}, {'1', '3'}, {'2', '3'}, {'1', '2', '3'}}[s
et]
```

Algebra

The following functions allow to generate and examine algebraic objects:

`fadd(n)` - creates an addition modulo  $n$  function. The function is given id: *addn* where  $n$  is as specified in the argument.

`group(groupName, set, fname)` - given a set and a binary function generates a group object with id: *groupName*.

Example:

```
>fadd(4)
'add4"
>group(Z4,{0,1,2,3},add4)
group(Z4)({'0', '1', '2', '3'}, add4)[group]
```

`zmodn(k)` - generates a group  $\mathbb{Z}/k$ . The group is given id: *Zmodk*, where  $k$  is an integer specified as the argument.

Example:

```
>zmodn(5)
group(Zmod5)({'0", '1", '2", '3", '4"}, add5)[group]
```

$S_n(k)$  - generates a symmetric group  $S_n$ . The group is given id: Sk, where  $k$  is an integer specified as the argument.

Example:

```
>Sn(3)
group(S3)({(), (('2", '3")), (('1", '2')), (('1", '2", '3')), (('1", '3", '2')), (('1", '3'))}, pcomposition3)[group]
```

The elements of the group are permutations. Each permutation is expressed as a list of cycles. Each cycle is a list of integers. Thus  $1 = ()$ , a cycle  $(1, 2)$  must be entered as  $((1, 2))$ .

$\text{cross}(\text{groupList})$  - given a list of groups, generates a cross product group.

Example:

```
>Sn(2)
group(S2)({(), (('1", '2'))}, pcomposition2)[group]
>cross((S2,S2,S2))
group(S2xS2xS2)({(), (), (), (), (), (('1", '2')), (), (('1", '2')), (), (), (('1", '2')), (('1", '2')), (('1", '2')), (('1", '2')), (), (), (('1", '2')), (), (('1", '2')), (('1", '2')), (('1", '2')), (('1", '2')), (('1", '2')), (('1", '2')), (('1", '2'))}, gcross0)[group]
```

$\text{gset}(G)$  - given a group  $G$  returns a set of  $G$ .

Example:

```
>gset(S3)
{(), (('2", '3')), (('1", '2')), (('1", '2", '3')), (('1", '3", '2')), (('1", '3"))}
```

$\text{gop}(G, x_1, x_2)$  - given a group  $G$  and elements  $x_1$  and  $x_2$ , performs  $x_1 \cdot x_2$  as defined in  $G$ .

Example:

```
>gop(S3, ((1,2)), ((2,3)))
(('1", '2", '3'))
```

$\text{gid}(G)$  - returns an identity of group  $G$ .

$\text{ginv}(G, x)$  - returns an inverse of  $x$  in group  $G$ .

$\text{gpow}(G, x, n)$  - return  $x^n$  in  $G$ .

Example:

```
>gpow(S3,((1,2)),2)
()
```

`ggen(G,S)` - given a group  $G$  and  $S \subseteq G$ , returns a subgroup generated by  $S$ .

Example:

```
>ggen(S3,{{(1,2,3)}})
{('1", '2", '3"), (('1", '3", '2")), ()}
```

Other commands:

A graph is created by using command: `addObject(name,graph)`.

Then vertices may be created by specifying a graph and some other object which is used to mark the vertex:

```
addVertex(g,1)
```

```
addVertex(g,2)
```

edges are entered by specifying the graph object, object marking source vertex, and finally an object marking the destination vertex.

```
addEdge(g,1,2)
```

Boolean objects: the software starts up with three built-in boolean objects: TRUE, FALSE, UNCERTAIN.

`id(object)` - returns string value of object's id. Example:

```
>id(addObject(g,graph))
'g"
```

`listFunctions()` - lists functions known to the software

`listConstants()` - lists constants

`listPredicates()` - lists predicates

`listSorts()` - lists object sorts

`listObjects()` - lists objects over which the quantification is defined

`add(x,y)` -  $x,y$  are integers. The result is  $x+y$

`sub(x,y)` -  $x,y$ , integers. The result is  $x-y$ .

`mult(x,y)` -  $x,y$ , integers. The result is  $x*y$ .

`pow(x,y)` -  $x,y$  integers. The result is  $x^y$ .

`addSort(sortId)` - creates a new sort with a specified id. If a sort with the specified id already exists, throws an exception.

`addObject(objectId, sortId)` - creates an object with `objectId` of the specified sort. If an object with the specified id already exists, throws an exception.

`relateSorts(parent, child)` - creates an inheritance relation between two sorts, where 'child' is a subsort of 'parent'.

loadFile(fileName, flag) - loads a file with the name fileName (it is suggested that fileName is a string). The flag is a boolean value TRUE or FALSE. If TRUE is specified, the progress of loading a file is output onto the display.

saveFile(fileName) - save the structure into the specified file. The contents are saved as a text file with instructions as they might be entered via a command line interface.

addFunction(functionId, arity) - functionId is identifier or a string which is to be function's id, arity is integer specifying the number of arguments for a function.

addPredicate(predicateId, arity) - predicateId is identifier or a string specifying predicate's id, arity is integer specifying the number of arguments for a predicate

addConstant(constantId) - adds a constant with the specified id.

setF(functionId, (arg1,arg2,...,argk),v) - sets a function extension entry so that f(arg1,arg2,...,argk) evaluates to v.

setP(predicateId, (arg1,arg2,...,argk),v) - sets a predicate extension entry so that p(arg1,...,argk) evaluates to v, where v is TRUE or FALSE.

setC(constantId, value) - equivalent to: constantId=value.

showExtension(id) - lists extension entries for a specified function or predicate. The list will contain only the modifications made to an extension by a user using setF(), setP() commands. This will not list entries for built in function, or functions defined using fdef command.

make(id) - returns an object with the specified id if it already exists, otherwise creates a string with the specified id.

appendList(listId, element) - append the specified list with the specified element. Example:

```
>appendList((1,2,3),4+5)
(0:'1", 1:'2", 2:'3", 3:'9") [object]
```

appendSet(setId, element) - append a set with the specified element. Example:

```
>appendSet(set(s,{1,2,3}),4+5)
{'1", '2", '3", '9"} [set]
```

removeFromList(listId, index) - removes the element with the specified index from the list. The index of the element is the position of the element in the list starting with 0. Example:

```
>c=(1,2,3,4)
(0:'1", 1:'2", 2:'3", 3:'4") [object]
>removeFromList(c,2)
'3" [integer]
>c
(0:'1", 1:'2", 2:'4") [object]
```

getFromList(listId, index) - returns the elements of the list with the specified index. Example:

```
>getFromList((1,2,(1,2,3)),2)
(0:'1", 1:'2", 2:'3") [object]
```

getFromSet(setId, index) - returns the element of the set with the specified index. The index of the element is the position starting with 0 of the element in the string representation of the set.

addToList(listId, index, element) - inserts the specified element in the position with the specified index in the list.

fdef (functionId, (varDef1, ...varDefk), "expr") - defines a function with the functionId. The function is to accept k argument which are defined in the vardef list. The value of the function is computed by evaluating the expression. Example: we will define a tertiary function  $f(x, y, z) \equiv (x + y)z$ .

```
>fdef(f, ((x, integer), (y, integer), (z, integer)), "(x+y)*z")
'TRUE'[object]
>f(1,2,3)
'9'[integer]
>f(2,3,1)
'5'[integer]
>f(2,3,2)
'10'[integer]
```

cat (x, y) - concatenates two strings, or lists. Examples:

```
>cat('this is a ', 'string')
'this is a string'[string]
>cat((1,2), (3,4))
(0:'1', 1:'2', 2:'3', 3:'4')[object]
```

and(a,b) - logical function. Evaluates to  $a \wedge b$ . Same as  $a \& \& b$ .

or(a,b) - logical function. Evaluates to  $a \vee b$ . Same as  $a || b$ .

not(a) - logical function. Evaluates to  $\neg a$ . Same as  $!a$ .

then (a,b) - logical function. Evaluates to  $a \rightarrow b$  or equivalently  $b \vee (\neg a)$ . Same as  $a - > b$ .

=(c,v) - constant assignment function. Assigns value of  $v$  to a constant  $c$ .

count(x) - x is a set or a list. Computes number of elements in the set or a list.

addVertex(g,x) - adds a vertex to a graph g marked by object x.

vertex(g,x) - returns a vertex from graph g which is marked by an object x.

addEdge(g,x1,x2) - adds an edge to a graph g, from vertex marked by x1 to a vertex marked by x2.

indexOf(list, element) - returns an index of a specified element in the list.

inDegree(g,x) - returns in-degree of a vertex marked by x in a graph g.

outDegree(g,x) - returns out-degree of a vertex marked by x in a graph g.

edgeSet(g) - returns a set of edges of the specified graph. The set is a set of lists of two elements, where each element is of the form (s,d) where s is a source vertex, d is destination vertex.

vertexSet(g) - returns a set of vertices of the specified graph.

powerSet(s) - generates a set of all subsets of the specified set. The set can then be referenced by id  $P_s$  where  $s$  is the id of the argument set. Example:

```

>set(S,{1,2,3})
{'1", '2", '3"}[set]
>powerSet(S)
{{}, {'1"}, {'2"}, {'1", '2"}, {'3"}, {'1", '3"}, {'2", '3"}, {'1", '2", '3'}}[set]
>PS
{{}, {'1"}, {'2"}, {'1", '2"}, {'3"}, {'1", '3"}, {'2", '3"}, {'1", '2", '3'}}[set]

```

set(setId, expr) - identifies a set specified by expr with the id: setId.

graphSet(vertexNum) - generates all the directed graphs with the specified number of vertices. Please note that due to the memory limitations the command works for vertexNum one of the following {0,1,2,3}.

integerSet(l,u) - generates a set of integers  $\{x \mid l \leq x \leq u\}$ . Example:

```

>integerSet(0,7)
{'0", '1", '2", '3", '4", '5", '6", '7"}

```

integer(stringValue) - creates an integer object with the specified value. Example:

```

>integer('1024")+1
'1025"

```

true() - built-in predicate that always evaluates to true. This is useful in listing objects of a certain sort. Example:

```

>Q((x, integerSet(0,7)), true())
(0:'0", 1:'1", 2:'2", 3:'3", 4:'4", 5:'5", 6:'6", 7:'7")

```

equals(x,y) - two place predicate. Return TRUE if x equals to y. The command is equivalent to  $x == y$ .

contains(x,y) - x is a list, set, string or a graph. Returns true if whenever x is a list or a set, x contains y as one of its elements. If x is a string, returns TRUE if y is a substring of x. If x is a graph then contains(x,y) returns true if a vertex marked with y is in the vertex set of x.

contains2(x,y) - if x is a string, contains2(x,y) is the same as contains(x,y). However if x is a list of a set, returns true if y is an element of x, or x is an element of one of the elements z of x whenever z is a list or a set. Example:

```

>contains2((1,2,{3,4}),4)
'TRUE"[object]
>contains((1,2,{3,4}),4)
'FALSE"[object]

```

less(x,y) - two place predicate for x,y integer. Returns TRUE if  $x < y$ .  
 leq(x,y) - two place predicate for x,y integer. Returns TRUE if  $x \leq y$ .  
 hasPath(g,x,y) - returns TRUE if there is a path from a vertex marked by x to a vertex marked by y in an undirected graph derived from g.  
 hasEdge(g,x,y) - returns TRUE if there is an edge from a vertex marked by x to a vertex marked by y in a graph g.  
 verbose() - toggles the display of a list from verbose to compact. Returns a new setting of the verbose flag. A verbose setting will make the list  $(x_1, x_2, \dots, x_n)$  as follows:  $(0 : x_1, 1 : x_2, \dots, n : x_n)$ . Example:

```
>verbose()
'TRUE"[object]
>(1,2,3)
(0:'1", 1:'2", 2:'3") [object]
>verbose()
'FALSE"[object]
>(1,2,3)
('1", '2", '3") [object]
```

### Algebra Examples:

Further by the textbook we will mean: Abstract Algebra, Third Edition by David S. Dummit and Richard M. Foote.

Example 1. A group.

The following definition is from p.16:

A group is an ordered pair  $(G, *)$  where  $G$  is a set and  $*$  is a binary operation on  $G$  satisfying the following axioms:

- (i)  $(a * b) * c = a * (b * c)$ , for all  $a, b, c \in G$ , i.e.  $*$  is associative.
- (ii) there exists an element  $e$  in  $G$ , called an identity of  $G$ , such that for all  $a \in G$  we have  $a * e = e * a = a$ ,
- (iii) for each  $a \in G$  there is an element  $a^{-1}$  of  $G$ , called an inverse of  $a$ , such that  $a * a^{-1} = a^{-1} * a = e$ .

Using software syntax:

```
>fdef(isGroup, ((G, group)), "FA(((a,G), (b,G), (c,G)), gop(G, gop(G, a, b), c)) == gop(G, a, gop(G, b, c))) && TE(((e,G)), FA(((a,G)), gop(G, a, e)) == gop(G, e, a) && gop(G, e, a) == a) && FA(((a,G)), TE(((b,G)), gop(G, a, b) == gop(G, b, a) && gop(G, a, b) == gid(G)))")
'TRUE"[object]
```

To check that  $\mathbb{Z}/5$  and  $S_3$  satisfy group axioms:

```

>isGroup(zmodn(5))
'TRUE"[object]
>isGroup(Sn(3))
'TRUE"[object]
>group(G,{0,1,2,3},fadd(5))

```

However, consider the following pair:  $G = (\{0, 1, 2, 3\}, \text{addition modulo } 5)$ . This should fail the group axioms since  $1 + 3 \notin G$ .

```

>fadd(5)
'add5"
>group(G,{0,1,2,3},add5)
group(G)({'0", '1", '2", '3"}, add5)[group]
>isGroup(G)
'FALSE"[object]

```

#### Example 2. Subgroups.

Using the subgroup criterion (p.47 of the textbook): A subset  $H$  of a group  $G$  is a subgroup if and only if

- (1)  $H \neq \emptyset$  and
- (2) for all  $x, y \in H$ ,  $xy^{-1} \in H$ .

Using the software syntax:

```

>fdef(isSubgroup, ((H,set), (G,group)), "!empty(H) && FA(((x,H), (y,H)), contains(H,
gop(G,x,ginv(G,y))))")
'TRUE"[object]

```

We will now check that  $\{0, 2\} \leq \mathbb{Z}/4$ .

```

>zmodn(4)
group(Zmod4)({'0", '1", '2", '3"}, add4)[group]
>isSubgroup({0,2},Zmod4)
'TRUE"[object]

```

However  $\{1, 3\} \not\leq \mathbb{Z}/4$ :

```

>isSubgroup({1,3},Zmod4)
'FALSE"[object]

```

#### Example 3. Centralizer of a set.

Using definition on p.49 of the textbook:  $C_G(A) = \{g \in G \mid gag^{-1} = a \text{ for all } a \in A\}$ . This is equivalent to

$$C_G(A) = \{g \in G \mid ga = ag \text{ for all } a \in A\}.$$

Using the software syntax:

```
>fdef(centralizer, ((G,group), (A,set)), "set(s,Q((x,G)),FA((a,A)),gop(G,x,a)=
=gop(G,a,x)))")
'TRUE"[object]
```

Now we compute  $C_{\mathbb{Z}/7}(\{1,2\})$  and  $C_{S_4}(\{(1,2)\})$ :

```
>zmodn(7)
group(Zmod7)({'0", '1", '2", '3", '4", '5", '6"}, add7)[group]
>Sn(4)
group(S4)({(), (('3", '4")), (('2", '3")), (('2", '3", '4')), (('2", '4", '3')),
(('2", '4")), (('1", '2')), (('1", '2'), ('3", '4')), (('1", '2", '3')), (('1",
'2", '3", '4')), (('1", '2", '4', '3')), (('1", '2", '4')), (('1", '3", '2')),
(('1", '3", '4", '2')), (('1", '3')), (('1", '3", '4')), (('1", '3'), ('2", '4')
), (('1", '3", '2", '4')), (('1", '4", '3", '2')), (('1", '4", '2')), (('1", '4"
, '3')), (('1", '4')), (('1", '4", '2", '3')), (('1", '4'), ('2", '3'))}, pcompo
sition4)[group]
>centralizer(Zmod7, {1,2})
{'0", '1", '2", '3", '4", '5", '6"}[set]
>centralizer(S4, {(1,2)})
{(), (('3", '4')), (('1", '2')), (('1", '2'), ('3", '4'))}[set]
```

We can now check that indeed these centralizer is a subgroup:

```
>isSubgroup(centralizer(S4, {(1,2)}), S4)
'TRUE"[object]
>isSubgroup(centralizer(Zmod7, {1,2}), Zmod7)
'TRUE"[object]
```

Example 4. Subgroups of a group.

Let's determine all the subgroups of a given group:

```
>fdef(subgroups, ((G,group)), "Q((H,powerSet(G)), isSubgroup(H,G))")
'TRUE"[object]
```

Subgroups of  $\mathbb{Z}/8$  :

```
>subgroups(zmodn(8))
({'0"}, {'0", '4"}, {'0", '2", '4", '6"}, {'0", '1", '2", '3", '4", '5", '6", '7"
})
```

Subgroups of  $S_3$ :

```
>subgroups(Sn(3))
({()}, {()}, (('2", '3"))}, {()}, (('1", '2"))}, {()}, (('1", '2", '3")), (('1", '3", '2'))}, {()}, (('1", '3"))}, {()}, (('2", '3")), (('1", '2')), (('1", '2", '3')), (('1", '3", '2')), (('1", '3'))}
```

Example 5. Elements of order 3 in  $S_3 \times S_3$ :

```
>Q((x,S3xS3),count(ggen(S3xS3,{x}))==3)
(((), (('1", '2", '3'))), ((), (('1", '3", '2'))), ((('1", '2", '3')), ()), ((('1", '2", '3')), (('1", '2", '3'))), ((('1", '2", '3')), (('1", '3", '2'))), ((('1", '3", '2')), ()), ((('1", '3", '2')), (('1", '2", '3'))), ((('1", '3", '2')), (('1", '3", '2'))))
```

Hence the elements of order 3 in  $S_3 \times S_3$  are:

$(1, (1, 2, 3)), (1, (1, 3, 2)), ((1, 2, 3), 1), ((1, 2, 3), (1, 2, 3)), ((1, 2, 3), (1, 3, 2)), ((1, 3, 2), 1), ((1, 3, 2), (1, 2, 3)), ((1, 3, 2), (1, 3, 2)).$

```
>count(Q((x,S3xS3),count(ggen(S3xS3,{x}))==3))
'8"[integer]
```

That is there are 8 elements of order 3 in  $S_3 \times S_3$ .

Example 6. Normal subgroups of  $S_3$  and  $\mathbb{Z}/6$ .

```
>Q((H,subgroups(S3)),FA((x,S3),(y,H),contains(H,gop(S3,x,gop(S3,y,ginv(S3,x))))))
({()}, {()}, (('1", '2", '3')), (('1", '3", '2'))}, {()}, (('2", '3')), (('1", '2")), (('1", '2", '3')), (('1", '3", '2')), (('1", '3'))}
```

As you can see the only normal subgroups of  $S_3$  are  $\{1\}$  and  $S_3$ .

For  $\mathbb{Z}/6$ , the normal subgroups are:

```
>zmodn(6)
group(Zmod6)({'0", '1", '2", '3", '4", '5"}, add6)[group]
>Q((H,subgroups(Zmod6)),FA((x,Zmod6),(y,H),contains(H,gop(Zmod6,x,gop(Zmod6,y,ginv(Zmod6,x))))))
({'0"}, {'0", '3"}, {'0", '2", '4"}, {'0", '1", '2", '3", '4", '5'})
```

which are all the subgroups of  $\mathbb{Z}/6$ :

```
>set(s,Q((H,subgroups(Zmod6))),FA((x,Zmod6),(y,H)),contains(H,gop(Zmod6,x,gop(
Zmod6,y,ginv(Zmod6,x))))))=subgroups(Zmod6)
'TRUE"[object]
```

Example 7: how many conjugate classes are there in  $S_3$ ?

The question can be rephrased as: what is the maximal cardinality of a subset of  $S_3$  where no two distinct elements are conjugates?

We first generate all the subsets of  $S_3$  where no two distinct elements are conjugates:

```
>Sn(3)
group(S3)({(), (('2", '3')), (('1", '2')), (('1", '2", '3')), (('1", '3", '2')),
 (('1", '3'))}, pcomposition3)[group]
>set(A,Q((H,powerSet(S3))),FA((x,H),(y,H),!(x==y)->!TE((z,S3),gop(S3,z,gop(
S3,x,ginv(S3,z)))=y))))
{ {}, {()}, { (('2", '3')) }, {(), (('2", '3')) }, { (('1", '2')) }, {(), (('1", '2'))
}, { (('1", '2", '3')) }, {(), (('1", '2", '3')) }, { (('2", '3')), (('1", '2", '3'))
}, {(), (('2", '3')), (('1", '2", '3')) }, { (('1", '2')), (('1", '2", '3')) }, {()
}, (('1", '2')), (('1", '2", '3')) }, { (('1", '3", '2')) }, {(), (('1", '3", '2'))
}, { (('2", '3')), (('1", '3", '2')) }, {(), (('2", '3')), (('1", '3", '2')) }, { (
'1", '2')), (('1", '3", '2')) }, {(), (('1", '2')), (('1", '3", '2')) }, { (('1", '
3')) }, {(), (('1", '3')) }, { (('1", '2", '3')), (('1", '3')) }, {(), (('1", '2", '
3')), (('1", '3')) }, { (('1", '3", '2')), (('1", '3')) }, {(), (('1", '3", '2')),
 (('1", '3')) } }[set]
```

Now we want a set of maximal cardinality:

```
>Q((X,A),FA((B,A),count(B)<=count(X)))
{(), (('2", '3')), (('1", '2", '3'))}, {(), (('1", '2')), (('1", '2", '3'))}, {
(), (('2", '3')), (('1", '3", '2'))}, {(), (('1", '2')), (('1", '3", '2'))}, {()
, (('1", '2", '3')), (('1", '3'))}, {(), (('1", '3", '2')), (('1", '3'))}
```

As you can see there is more than one set. However, since all such sets have maximal cardinality, we can take any such set and count the number of elements:

```
>count(getFromList(Q((X,A),FA((B,A),count(B)<=count(X))),0))
'3"[integer]
```

Hence, there are 3 conjugate classes in  $S_3$ .

Note:

1. In the first attempt the goal of the software is to provide the language and convenient interface rather than fast performance. Since evaluating second order expressions in the

software is exponential in the number of operations performed, evaluating such expressions over the large sets may not be compositionally feasible.

2. Once a group or any object is created under an id, further attempts to create an object with the same id will generate an exception. So once a group is created, let's say by using  $z \bmod n(8)$  command, it should subsequently be referred to by its id: in this case  $Z \bmod 8$ .

### **Conclusion.**

The paper described a Java software package that provides a capability to define via command line interface arbitrary monadic second order language and an arbitrary finite structure which interprets the language. The package then evaluates monadic second order expressions and queries involving monadic second order expressions in the defined structure. The paper further describes an adaptation of the software to algebra.

Please send questions or comments to: [abrik@stanford.edu](mailto:abrik@stanford.edu).