Domains of data and domains of terms in AXIOM

Ronald Brown, Winfried Dreckmann
School of Mathematics
University of Wales, Bangor
Gwynedd LL57 1UT, U.K.

March 14, 1995

Introduction

The symbolic computation package AXIOM provides a high level programming language with several advantages for the coding of new mathematical structures and for manipulating with them. In this sense it could be said to make a contribution to "structural computation". This term reflects how the development of computation has paralleled that of mathematics, with, in turn, numbers, symbols, and now structures.

The advantages which AXIOM has in developing the coding of mathematical structures are:

- The code is nearer than in other programs to standard mathematical practice.
- The language is strongly typed and object oriented.
- The internal representation of objects is chosen by the user.
- The output form is chosen by the user.
- There is a large range of inbuilt mathematical packages and structures which are available to link in with new ones.
- The code is compiled, which makes for efficiency, speed and power.
- There is a convenient Hyperdoc system which gives relational information on existing code and is therefore helpful for the construction of new code.

On the other hand, it is not easy to get started by oneself in writing AXIOM code. We hope therefore that this account will be helpful to those interested in using the system.

Structures in AXIOM are formed using what AXIOM calls "categories" and "domains". It is in the construction of domains that the real work of programming AXIOM takes place. A domain is a type which defines a collection of exported symbols, which may denote types, constants and functions.

Categories in AXIOM are not categories in the sense now usual in mathematics, and which we also require in this paper. So we will call the former "AXIOM categories". Formally, these are types which specify information about domains. They include for the domain the specification of the public interface, which consists of a collection of declarations for those

operations which may be used by clients of the domain. A domain belongs to an AXIOM category by assertion. For our purposes, an AXIOM category may be thought of as a collection of signatures which the AXIOM domains asserted to belong to, or have, this category must export.

We shall explain the overall structure of such programs, and illustrate this with our own examples.

The main new concept we wish to illustrate in this paper is a distinction between "domains of data" and "domains of terms", and its use in the programming of certain mathematical structures. Although this distinction is implicit in much of the programming work that has gone into the construction of AXIOM categories and domains, we believe that a formalisation of this is new, that standards and conventions are necessary and will be useful in various other contexts. We shall show how this concept may be used for the coding of free categories and groupoids on directed graphs.

The code for handling finite directed graph data (see A.2) allows for the input of a finite set of objects, a finite set of arrows, and information on the source and target for each arrow. It is important that the export from this code is a domain which has AXIOM category SetCategory. This means that the signatures allowed for the objects produced are principally "equality". That is, in the programming for this data, we have to specify when two finite directed graph data are equal.

To proceed to the free category and free groupoid on a given finite directed graph, G, say, we need to construct, from the data of G, an object with signatures of a more complicated kind, namely

$$\mathit{object?}, \quad \mathit{arrow?}, \quad \mathit{source}, \quad \mathit{target}.$$

These are the signatures belonging to the new AXIOM category DirectedGraphCategory (see A.1).

This change of view can be seen as moving from a view external to the graph to a view internal to the graph. Alternatively, we see the "terms" of the new object to be precisely the objects and arrows of the graph G. It is these "terms" which are used as the starting point in the construction of free categories and groupoids.

We thus develop another program which turns an object with the data of a finite directed graph to a "domain of terms" of that finite directed graph, which has the signatures mentioned above.

The notion of free groupoid $F(G)$ on a directed graph G is important for various manipulations of graphs, since the elements of $F(G)$ are essentially the paths of G. The distinction between domains of data and terms is necessary because we wish to construct the free category and the free groupoid on any domain (of terms) which has AXIOM category DirectedGraphCategory, where the "terms" in the domain may or may not come from a "domain of data". One advantage of this procedure is iteration: any groupoid, and in particular the free groupoid, is of AXIOM category DirectedGraphCategory, and so the FreeGroupoid construction can be applied to that, and so on.

The implementation of the free category on a directed graph is expected to be useful because its terms are what is needed for the basic compositions of category theory. In any case, it is a basic construction in mathematics, generalising the free monoid on a set.

Acknowledgements

We are very grateful to Larry Lambe for the explanations and examples in his publications, and for much encouragement, time and help, in a number of visits to Bangor. We would

also like to thank: the other members of the Bangor team, including, Andy Tonks, who first exploited AXIOM at Bangor, and, more recently, Tim Porter, Chris Wensley and A. Razak Salleh for continuous help and discussions; Themos Tsikas at NAG for help; and Prof J H Davenport for allowing remote use of the AXIOM system at Bath. The equipment and software for this work on AXIOM is supported by an EPSRC grant GR/J 63552 "Non abelian homological algebra". Winfried Dreckmann is grateful to a University of Wales Fellowship funded by the Validation Board for support for his work.

1 What the programs do

1.1 ccat.as: Some new AXIOM categories

This file defines the new AXIOM categories that are used in hierarchical order:

- DirectedGraphCategory
- ReflexiveDirectedGraphCategory
- CatCategory
- GroupoidCategory

For each of these, there is a set of signatures which defines the operations allowable on objects of this type. Some of these signatures are basic and some are deduced. For example, CatCategory allows for a composition of a list of arrows, which is deduced in the standard way from the composition of two arrows.

1.2 fdirgdata.as: The domain of data FiniteDirectedGraphData(O,A)

The file contains the code for constructing the domain of finite directed graph data. A domain is a special type which comes with its own operations, i. e. an object in the sense of object oriented programming. The definition of the domain is actually a so-called "domain constructor", that is a domain depending on parameters to be specified by the user at runtime. In this case there are two parameters, the type of the objects and the type of the arrows. AXIOM will construct the corresponding domain at runtime and then it is the turn of the user again to construct specific data objects and bring them into action by means of the operations exported by the new domain.

Here is some AXIOM printout illustrating this:

```
(1) ->FDGD := FiniteDirectedGraphData(Integer,Symbol)

   (1)  FiniteDirectedGraphData(Integer,Symbol)
                                                          Type: Domain
(2) ->g := makeGraphData(set[1,2,3], set[a,b,c,d],_
                         [[a,1,2],[b,2,3],[c,3,1],[d,3,3]])$FDGD

   (2)  [{1,2,3},{a,b,c},[a:1,2],[b:2,3],[c:3,1],[d:3,3]]
                              Type: FiniteDirectedGraphData(Integer,Symbol)
```

```
(3) ->setOfObjects(g)

   (3)  {1,2,3}
                                                    Type: Set Integer
```

The chosen types for objects and arrows are Integer and Symbol, respectively. AXIOM constructs *FDGD*, and the user produces g by calling *makeGraphData* from *FDGD*. Note how AXIOM keeps the user informed about types. The output form in (2), for example the notation [a:1,2], is determined by the user in setting up the domain resp. domain constructor, see A.2 lines 97 – 113.

1.3 fdirg.as: The domain of terms FiniteDirectedGraph(O,A,g)

This domain constructor depends on three parameters, a type O for objects, a type A for arrows and an object g of type *FiniteDirectedGraphData(O,A)*. The domain G := *FiniteDirectedGraph(O,A,g)* which is constructed out of O, A, and g, is a domain of terms, of AXIOM category DirectedGraphCategory. This means that if something has type G, then we can apply to it the operations *object?*, *arrow?*, *source*, and *target?*, and for two arrows, we can determine if they are composable (cf. A.1).

```
(4) ->G := FiniteDirectedGraph(Integer,Symbol,g)

   (4)
  FiniteDirectedGraph(Integer,Symbol,[BRACEAGGLST123,BRACEAGGLSTabc,BRACKETCO
  NCATa:1,2,BRACKETCONCATb:2,3,BRACKETCONCATc:3,1],BRACKETCONCATd:3,3])
                                                          Type: Domain

(5) ->a0 := makeArrow(a)$G

   (5)  [a:1,2]
Type: FiniteDirectedGraph(Integer,Symbol,[BRACE(AGGLST 1 2 3),BRACE(AGGLST a
b c),BRACKET(CONCAT a : 1 , 2),BRACKET(CONCAT b : 2 , 3),BRACKET(CONCAT c : 3
 , 1),BRACKET(CONCAT d : 3 , 3)])

(6) ->source(a0)

   (6)  1
Type: FiniteDirectedGraph(Integer,Symbol,[BRACE(AGGLST 1 2 3),BRACE(AGGLST a
b c),BRACKET(CONCAT a : 1 , 2),BRACKET(CONCAT b : 2 , 3),BRACKET(CONCAT c : 3
 , 1),BRACKET(CONCAT d : 3 , 3)])
```

Here the printout of the previous section is continued. Note that the output (4) contains an internal one line description of the output form of the domain of data *FiniteDirectedGraphData(O,A)*.

1.4 freec.as: The domain of terms FreeCategory(G)

The parameter of this domain constructor may be any domain G of AXIOM category DirectedGraphCategory. It constructs the free category on it. The terms of this domain are either

the terms of G which satisfy *object?* or reduced words of arrows (see lines 31 – 38 of freec.as for a description of reduced words).

One way of getting to a domain of AXIOM category DirectedGraphCategory is to start with an object g of type *FiniteDirectedGraphData(O,A)* and to turn it into the domain of terms G := *FiniteDirectedGraph(O,A,g)* as described above. Then we can form the free category *FreeCategory(FiniteDirectedGraph(O,A,g))* and perform computations in the free category as follows:

```
(7) -> F := FreeCategory(G)

   (7)
  FreeCategory FiniteDirectedGraph(Integer,Symbol,[BRACEAGGLST123,BRACEAGGLST
  abc,BRACKETCONCATa:1,2,BRACKETCONCATb:2,3,BRACKETCONCATc:3,1,BRACKETCONCATd
  :3,3])
                                                          Type: Domain

(8) ->a1 := a0 :: F

   (8)  [[a:1,2]]
Type: FreeCategory FiniteDirectedGraph(Integer,Symbol,[BRACE(AGGLST 1 2 3),BR
ACE(AGGLST a b c),BRACKET(CONCAT a : 1 , 2),BRACKET(CONCAT b : 2 , 3),BRACKET
(CONCAT c : 3 , 1),BRACKET(CONCAT d : 3 , 3)])
```

(From now on the type information will be omitted unless a new type occurs.)

```
(9) -> b1 := (makeArrow(b)$G) :: F

   (9)  [[b:2,3]]

(10) ->a1 * b1

   (10) [[a:1,2][b:2,3]]

(11) ->composable?(b1,a1)

   (11)  false
                                                          Type: Boolean

(12) ->c1 := (makeArrow(c)$G) :: F

   (12)  [[c:3,1]]

(13) ->d1 := (makeArrow(d)$G) :: F

   (13)  [[d:3,3]]

(14) ->abd := a1 * b1 * d1
```

```
   (14)  [[a:1,2][b:2,3][d:3,3]]

(15) -> dddd := d1 * d1 * d1 * d1

                 4
   (15)  [[d:3,3] ]

(16) ->comp[abd,dddd,d1,c1,abd,d1]

                                6                                  2
   (16)  [[a:1,2][b:2,3][d:3,3] [c:3,1][a:1,2][b:2,3][d:3,3] ]
```

Note that words in the free category are always surrounded by square brackets. The reason for this will appear in the next section.

1.5 freegpd.as: The domain of terms FreeGroupoid(G)

This domain constructor acts on a domain of AXIOM category DirectedGraphCategory in the same way as *FreeCategory(G)* does, but this time produces the free groupoid on$G$. The file freegpd.as is very similar to freec.as, although this time a more sophisticated algorithm is needed to reduce words. The only difference in the representation of terms is that negative exponents are allowed now. Doing constructions in the same way as in the previous section we this time get free groupoids:

```
(17) ->FG := FreeGroupoid(G)

   (17)
  FreeGroupoid FiniteDirectedGraph(Integer,Symbol,[BRACEAGGLST123,BRACEAGGLST
  abc,BRACKETCONCATa:1,2,BRACKETCONCATb:2,3,BRACKETCONCATc:3,1,BRACKETCONCATd
  :3,3])
                                                          Type: Domain

(18) ->ag1 := a0 :: FG

   (18)  [[a:1,2]]
Type: FreeGroupoid FiniteDirectedGraph(Integer,Symbol,[BRACE(AGGLST 1 2 3),BR
ACE(AGGLST a b c),BRACKET(CONCAT a : 1 , 2),BRACKET(CONCAT b : 2 , 3),BRACKET
(CONCAT c : 3 , 1),BRACKET(CONCAT d : 3 , 3)])

(19) ->bg1 := (makeArrow(b)$G) :: FG

   (19)  [[b:2,3]]

(20) ->cg1 := (makeArrow(c)$G) :: FG

   (20)  [[b:2,3]]

(21) ->u := comp[ag1,bg1,cg1]
```

```
   (21) [[a:1,2][b:2,3][c:3,1]]

(22) ->inv(u)

                -1       -1       -1
  (22) [[c:3,1]  [b:2,3]  [a:1,2]  ]

(23) ->u * inv(u)

   (23) 1
        1
```

Now there is a surprise. The types *FreeCategory(G)* and *FreeGroupoid(G)* have AXIOM category DirectedGraphCategory (because the set of signatures of DirectedGraphCategory is a subset of the set of signatures of both CatCategory and GroupoidCategory). So the construction can be iterated. That is, we can form:

```
(24) ->FG2 := FreeGroupoid(FG)

   (24)
  FreeGroupoid FreeGroupoid FiniteDirectedGraph(Integer,Symbol,[BRACEAGGLST12
  3,BRACEAGGLSTabc,BRACKETCONCATa:1,2,BRACKETCONCATb:2,3,BRACKETCONCATc:3,1,B
  RACKETCONCATd:3,3])
                                                                  Type: Domain
(25) ->ag2 := ag1 :: FG2

   (25)   [[[a:1,2]]]
Type: FreeGroupoid FreeGroupoid FiniteDirectedGraph(Integer,Symbol,[BRACE(AGG
LST 1 2 3),BRACE(AGGLST a b c),BRACKET(CONCAT a : 1 , 2),BRACKET(CONCAT b : 2
 , 3),BRACKET(CONCAT c : 3 , 1),BRACKET(CONCAT d : 3 , 3)])

(26) ->(u :: FG2) * (inv(u) :: FG2)

                                            -1       -1       -1
   (26)   [[[a:1,2][b:2,3][c:3,1]][[c:3,1]  [b:2,3]  [a:1,2]  ]]

(27) ->(u :: FG2) * inv(u :: FG2)

   (27)  1
         1
```

We now see the reason for the brackets round a word in the free constructions, since they enable one to distinguish between various levels.

2 The structure of the definitions of AXIOM categories and domains

The three levels of objects, domains, categories are fundamental to AXIOM. For a detailed account see [2]. Further explanations and examples are given in [1, 3, 4, 5, 6, 7, 8]. The following remarks are meant as a guide through the code listed in the appendix.

A domain is a programming environment with a collection of exported constants which can be types, functions or other things. It typically exports one type (denoted %) and a collection of related operations. Other typical features are parameters (already discussed in sections 1.2 – 1.5 above), the internal representation type, and the output form which allows for user controled output. A convenient but not the only way of writing the code for a domain is the following:

```
<domain name> (parameters) : Exports == Implementation where
{
   Exports ==> <category> with { ... }

   Implementation ==> <parent domain> add { ...}
}
```

Here "Exports" and "Implementation" are names of two macros which are defined in the immediately following where-part. The names "Exports" and "Implementation" don't matter, they are immediately replaced in the process of compilation by their macro expansions. This form is convenient because the Exports and Implementation parts can become very large and complicated. The Exports part is a type declaration — actually the definition of an Axiom category — while the Implementation part contains the code needed for the exported operations.

AXIOM categories form an hierarchy. This is compatible with the inclusion of their collections of signatures, but the actual relationship of the categories is determined by assertion. Our hierarchy (see A.1) is: DirectedGraphCategory, ReflexiveDirectedGraphCategory, CatCategory, GroupoidCategory. AXIOM categories are also designed to include assumptions about the operations.

A domain asserted to belong to one of these categories must export the operations declared in the category, and these operations must satisfy the assumptions specified in the category. The domain will usually have more operations than those of the category. For example, the domain FiniteDirectedGraph (see A.3), which belongs to DirectedGraphCategory, has four additional operations (see lines 17-21).

The code in A.2 – A.5 is all written in the way indicated above. The with- and add-forms reflect the hierarchical nature of the whole AXIOM system. Categories are usually defined as extensions of existing categories and the Implementation part can make use of existing code in a "parent domain".

A typical Exports part is given by lines 9 – 18 in A.2. SetCategory is one of the standard AXIOM categories containing signatures for equality and output form. But this doesn't mean that *FiniteDirectedGraphData(O,A)* has AXIOM category SetCategory only. It in fact means that the domain has any AXIOM category given by any subset of the signatures in SetCategory and in the with-part.

Our code has no examples of parent domains. If the code in the Implementation part wants

to make use of other AXIOM domains, the exports of these domains have to be imported. An easy way of doing this is via explicit imports as in lines 22 – 29.

Line 31 contains the definition of the internal representation of the data as a macro. The name "Rep" of the macro matters because it is used by other macros including "rep" and "per" which form the bridge between internal representation and exported type. Here Rep is a Record. There is no access to the internal structure of this Record outside the domain environment, so, typically, controlled access is provided by exported operations. These are the operations *setOfObjects*, *setOfArrows*, *source* and *target* in this example.

Typically, there are also operations for constructing instances of the domain, for example, in this case, the operation *makeGraphData* (see lines 11 and 69 – 77). The function "checkAnchorlist" in lines 39 – 67 is an example of an internal function which is not exported. Beside these functions there has to be code which corresponds to the signatures in SetCategory. The code for equality is in lines 35 – 37, and the code for the operation which coerces to the output form is listed in lines 97 – 113.

The new AXIOM categories in A.1, which all extend SetCategory, give a good impression of the nature of the AXIOM hierachy of categories. One should mention here the possibility of "default definitions" for the signatures of the category. A default definition is code related to a signature which AXIOM uses if there is no code in a given domain for the signuture. An example is the default part for *composable?* in lines 13 – 17 and for *comp* in lines 33 – 38 of A.1.

A The AXIOM files

The following code is written in the new programming language "AXIOM-XL, the AXIOM Extension Language", which is part of the new release AXIOM 2.0.

A.1 Axiom category definitions: ccat.as

```
 1 #include "axiom.as"
 2
 3
 4 define DirectedGraphCategory: Category == SetCategory with
 5 {
 6     object?     : % -> Boolean;
 7     arrow?      : % -> Boolean;
 8     source      : % -> %;
 9     target      : % -> %;
10
11     composable? : (%,%) -> Boolean;
12
13     default composable?(x:%,y:%):Boolean ==
14             {
15                 (arrow? x) and (arrow? y) => target(x) = source(y);
16                 error "arrows required";
17             }
18 }
19
```

```
20
21 define ReflexiveDirectedGraphCategory: Category == DirectedGraphCategory
    with
22 {
23     identity? : % -> Boolean;
24     identity  : % -> %;
25 }
26
27
28 define CatCategory: Category == ReflexiveDirectedGraphCategory with
29 {
30     *    : (%,%)   -> %;
31     comp : List(%) -> %;
32
33     default comp(u:List(%)):% ==
34             {
35                 empty? u => error "Argument must be a non-empty list.";
36                 empty? rest(u) => first(u);
37                 first(u) * comp(rest(u));
38             };
39 }
40
41
42 define GroupoidCategory: Category == CatCategory with
43 {
44     inv : % -> %;
45 }
```

A.2 Finite directed graph data: fdirgdata.as

```
 1 #include "axiom.as"
 2
 3 FiniteDirectedGraphData(O:SetCategory, A:SetCategory): Exports == Imple
   mentation where
 4 {
 5   KST ==> Record(key: A, src: O, tgt: O);
 6   KO  ==> Record(key: A, ob: O);
 7   OF  ==> OutputForm;
 8
 9   Exports ==> SetCategory with
10   {
11     makeGraphData : (Set(O), Set(A), List(KST)) -> %;
12
13     setOfObjects  : % -> Set(O);
14     setOfArrows   : % -> Set(A);
15
16     source        : (%,A) -> O;
```

```
17      target        : (%,A) -> O;
18    }
19
20    Implementation ==> add
21    {
22      import from Set(O);
23      import from Set(A);
24      import from KST;
25      import from KO;
26      import from OF;
27      import from Character;
28      import from String;
29      import from Boolean;
30
31      Rep ==> Record(ob: Set(O), arr: Set(A), src: Table(A,O), tgt: Table
    (A,O));
32
33      import from Rep;
34
35      ((x:%) = (y:%)): Boolean ==
36        (rep(x).ob = rep(y).ob) and (rep(x).arr = rep(y).arr) and
37        (rep(x).src = rep(y).src) and (rep(x).tgt = rep(y).tgt);
38
39      checkAnchorList(s: Set(O), a: Set(A), anchorList: List(KST)):() ==
40      {
41        for x in anchorList repeat
42        {
43          not member?(x.key, a) =>
44            error "Anchor list contains arrows not in the set of arrows."
    ;
45          not member?(x.src, s) =>
46            error "Anchor list contains sources not in the set of objects
    .";
47          not member?(x.tgt, s) =>
48            error "Anchor list contains targets not in the set of objects
    .";
49        }
50        anchorRest := anchorList;
51        while (not empty? anchorRest) repeat
52        {
53          firstKey := (first anchorRest).key;
54          for b in rest(anchorRest) repeat
55          {
56            if firstKey = b.key then
57              error "Anchor list contains duplicate arrows.";
58          };
59          anchorRest := rest(anchorRest);
```

```
60        }
61        keyList : List(A) := [x.key for x in anchorList];
62        for y in members(a) repeat
63        {
64          not member?(y, keyList) =>
65            error "Anchor list does not contain all the arrows.";
66        }
67      }
68
69      makeGraphData(s: Set(O), a: Set(A), anchorList: List(KST)):% ==
70      {
71        checkAnchorList(s,a,anchorList);
72        srcList := [[x.key, x.src] for x in anchorList]$List(KO);
73        tgtList := [[x.key, x.tgt] for x in anchorList]$List(KO);
74        srcTable := table(srcList)$Table(A,O);
75        tgtTable := table(tgtList)$Table(A,O);
76        per([s, a, srcTable, tgtTable]$Rep);
77      }
78
79      setOfObjects(x: %): Set(O) == rep(x).ob;
80
81      setOfArrows(x: %):Set(A) == rep(x).arr;
82
83      source(x: %, a: A): O ==
84      {
85        not member?(a,rep(x).arr) =>
86          error "Second argument is not a member of the set of arrows.";
87        (rep(x).src).a;
88      }
89
90      target(x: %, a: A): O ==
91      {
92        not member?(a,rep(x).arr) =>
93          error "Second argument is not a member of the set of arrows.";
94        (rep(x).tgt).a;
95      }
96
97      coerceEntry(x: %, a: A): OF ==
98      {
99        s : O := (rep(x).src).a;
100       t : O := (rep(x).tgt).a;
101       outputList : List(OF) :=
102         [coerce(a)$A, char(":")::OF, coerce(s)$O, char(",")::OF, coerce
    (t)$O];
103       bracket(hconcat(outputList));
104     }
105
```

```
106     coerce(x: %): OF ==
107     {
108       keyList : List(A) := members(rep(x).arr)$Set(A);
109       itemList : List(OF) := [coerceEntry(x,a) for a in keyList];
110       itemList := cons(coerce(rep(x).arr)$Set(A),itemList);
111       itemList := cons(coerce(rep(x).ob)$Set(O),itemList);
112       bracket(commaSeparate(itemList));
113     }
114   }
115 }
```

A.3 Finite directed graphs: fdirg.as

```
 1 #include "axiom.as";
 2
 3 #library CCATLIB "ccat.ao";
 4 #library FDIRGDATALIB "fdirgdata.ao";
 5
 6 import from CCATLIB;
 7 import from FDIRGDATALIB;
 8
 9
10 FiniteDirectedGraph(O: SetCategory, A: SetCategory, G: FiniteDirectedGr
   aphData(O,A)):Exports == Implementation where
11 {
12   OF  ==> OutputForm;
13   FDG ==> FiniteDirectedGraphData(O,A);
14
15   Exports ==> DirectedGraphCategory with
16   {
17      makeObject    : O -> %;
18      makeArrow     : A -> %;
19
20      retractObject : % -> O;
21      retractArrow  : % -> A;
22   }
23
24   Implementation ==> add
25   {
26     import from Set(O);
27     import from Set(A);
28     import from FDG;
29     import from Boolean;
30     import from Character;
31     import from String;
32
33     Rep ==> Union(object:O, arrow:A);
```

```
34
35      import from Rep;
36
37      ((x:%) = (y:%)):Boolean ==
38      {
39        (rep(x) case object) and (rep(y) case object) =>
40          (rep(x).object = rep(y).object)$O;
41        (rep(x) case arrow) and (rep(y) case arrow) =>
42          (rep(x).arrow = rep(y).arrow)$A;
43        false;
44      }
45
46      object?(x:%):Boolean ==
47        (rep(x) case object) and member?(rep(x).object,setOfObjects(G));
48
49      arrow?(x:%):Boolean ==
50        (rep(x) case arrow) and member?(rep(x).arrow,setOfArrows(G));
51
52      source(x:%):% ==
53      {
54        (rep(x) case object) or ((rep(x) case arrow) and
55        (not member?(rep(x).arrow,setOfArrows(G)))) =>
56          error "Argument is not in the set of arrows.";
57        per([source(G,rep(x).arrow)$FDG]$Rep);
58      }
59
60      target(x:%):% ==
61      {
62        (rep(x) case object) or ((rep(x) case arrow) and
63        (not member?(rep(x).arrow,setOfArrows(G)))) =>
64          error "Argument is not in the set of arrows.";
65        per([target(G,rep(x).arrow)$FDG]$Rep);
66      }
67
68      makeObject(x:O):% ==
69      {
70        not member?(x,setOfObjects(G)) =>
71          error "Argument is not in the set of objects.";
72        result : Rep := [x];
73        result.object := x;
74        per(result);
75      }
76
77      makeArrow(x:A):% ==
78      {
79        not member?(x,setOfArrows(G)) =>
80          error "Argument is not in the set of arrows.";
```

```
81        result : Rep := [x];
82        result.arrow := x;
83        per(result);
84      }
85
86      retractObject(x:%):O ==
87      {
88        rep(x) case arrow => error "Argument is not an object.";
89        rep(x).object;
90      }
91
92      retractArrow(x:%):A ==
93      {
94        rep(x) case object => error "Argument is not an arrow.";
95        rep(x).arrow;
96      }
97
98      coerce(x:%):OF ==
99      {
100       rep(x) case object => (rep(x).object)::OF;
101       rep(x) case arrow =>
102       {
103         a : A := rep(x).arrow;
104         s : O := source(G,rep(x).arrow)$FDG;
105         t : O := target(G,rep(x).arrow)$FDG;
106         outputList : List(OF) :=
107           [char("[")::OF, coerce(a)$A, char(":")::OF, coerce(s)$O,
108            char(",")::OF, coerce(t)$O, char("]")::OF];
109         hconcat(outputList);
110       }
111       never;
112     }
113   }
114 }
115
```

A.4 Free categories: freec.as

```
1 #include "axiom.as";
2
3 #library CCATLIB "ccat.ao";
4 import from CCATLIB;
5
6
7 FreeCategory(G:DirectedGraphCategory): Exports == Implementation where
8 {
9   NNI       ==> NonNegativeInteger;
```

```
10   OF        ==> OutputForm;
11   TERM      ==> Record(gen:G,exp:NNI);
12   GFAILED   ==> Union(value1:G, failed: 'failed');
13   NNIFAILED ==> Union(value1:NNI, failed: 'failed');
14
15   Exports ==> Join(CatCategory,RetractableTo G) with
16   {
17     termList : % -> List(TERM);
18   }
19
20   Implementation ==> add
21   {
22     import from Boolean;
23     import from NNI;
24     import from Character;
25     import from String;
26     import from TERM;
27     import from GFAILED;
28     import from NNIFAILED;
29     import from OF;
30
31     Rep ==> Union(object:G, arrow:List(TERM));
32                 -- case G: only object of the graph
33                 -- case List TERM: only reduced "words" either given
34                 -- by a non-empty list of "arrows of G" (satisfying
35                 -- arrow?) with positive exponents or by a list of
36                 -- length 1 consisting of an "object of G" with
37                 -- exponent 0 (in which case the identity arrow of
38                 -- the category is represented).
39
40     import from Rep;
41
42     ((x:%) = (y:%)):Boolean ==
43     {
44       (rep(x) case object) and (rep(y) case object) => rep(x).object =
   rep(y).object;
45       (rep(x) case arrow) and (rep(y) case arrow) =>
46       {
47         xx := rep(x).arrow;
48         yy := rep(y).arrow;
49         not (#xx = #yy) => false;
50         for xterm in xx for yterm in yy repeat
51         {
52           not (xterm.gen = yterm.gen) => return false;
53           not (xterm.exp = yterm.exp) => return false;
54         }
55         true;
```

```
56        }
57        false;
58      }
59
60      coerce(x:G):% ==
61      {
62        object?(x)$G => per([x]$Rep);
63        per([[[x,1]$TERM]$List(TERM)]$Rep);
64      }
65
66      retractIfCan(x:%): GFAILED ==
67      {
68        rep(x) case object => [rep(x).object]$GFAILED;
69        xx := rep(x).arrow;
70        empty?(xx) => never;
71        not empty? rest(xx) => [failed]$GFAILED;
72        term : TERM := first(xx);
73        (zero? term.exp) and (object?(term.gen)$G) => [term.gen]$GFAILED;
74        (one? term.exp) and (arrow?(term.gen)$G) => [term.gen]$GFAILED;
75        never;
76      }
77
78      termList(x:%):List(TERM) ==
79      {
80        rep(x) case object => error "arrow required";
81        rep(x).arrow;
82      }
83
84      object?(x:%):Boolean == rep(x) case object;
85
86      arrow?(x:%):Boolean == rep(x) case arrow;
87
88      source(x:%):% ==
89      {
90        rep(x) case object => error "arrow required";
91        xx := rep(x).arrow;
92        object?(first(xx).gen)$G => per([first(xx).gen]$Rep);
93        per( [source(first(xx).gen)$G]$Rep);
94      }
95
96      target(x:%):% ==
97      {
98        rep(x) case object => error "arrow required";
99        xx := rep(x).arrow;
100       object?(first(xx).gen)$G => per([first(xx).gen]$Rep);
101       per([target(last(xx).gen)$G]$Rep);
102     }
```

```
103
104     identity?(x:%):Boolean ==
105     {
106       rep(x) case object => error "arrow required";
107       xx := rep(x).arrow;
108       object?(first(xx).gen)$G;
109     }
110
111     identity(x:%):% ==
112     {
113       rep(x) case arrow  => error "object required";
114       xx := rep(x).object;
115       not (object?(xx)$G) => never;
116       termList : List(TERM) := [[xx,0]$TERM];
117       per([termList]$Rep);
118     }
119
120    ((x:%) * (y:%)):% ==          -- this is where we are reducing words
121    {
122       rep(x) case object => error "arrow required";
123       rep(y) case object => error "arrow required";
124       not composable?(x,y) => error "not composable";
125       identity?(x) => y;
126       identity?(y) => x;
127       xx := rep(x).arrow;
128       yy := rep(y).arrow;
129       xxlast  := last(xx);
130       yyfirst := first(yy);
131       xxlast.gen =$G yyfirst.gen =>
132       {
133         xxx := first(xx,subtractIfCan(#xx,1).value1);
134         newterm := [last(xx).gen,last(xx).exp + first(yy).exp]$TERM;
135         per([append(xxx,cons(newterm,rest(yy)))]$Rep);
136       }
137       per([concat(xx,yy)]$Rep);
138     }
139
140     coerceTerm(term:TERM):OF ==
141     {
142       one? term.exp => coerce(term.gen)$G;
143       scripts(coerce(term.gen)$G,
144         [empty()$OF,coerce(term.exp)$NNI]$List(OF))$OF;
145     }
146
147     coerce(x:%):OF ==                      -- coerce to output form
148     {
149       rep(x) case object => coerce(rep(x).object)$G;
```

```
150       xx := rep(x).arrow;
151       identity?(x) =>
152         scripts(char("1")::OF,[coerce(first(xx).gen)$G]$List(OF));
153       bracket(hconcat([coerceTerm(y) for y in xx]$List(OF))$OF)$OF;
154     }
155   }
156 }
```

A.5 Free groupoids: freegpd.as

```
 1 #include "axiom.as";
 2
 3 #library CCATLIB "ccat.ao";
 4 import from CCATLIB;
 5
 6
 7 FreeGroupoid(G:DirectedGraphCategory): Exports == Implementation where
 8 {
 9   NNI       ==> NonNegativeInteger;
10   INT       ==> Integer;
11   OF        ==> OutputForm;
12   TERM      ==> Record(gen:G,exp:INT);
13   GFAILED   ==> Union(value1:G, failed: 'failed');
14   NNIFAILED ==> Union(value1:NNI, failed: 'failed');
15
16   Exports ==> Join(GroupoidCategory,RetractableTo G) with
17   {
18     termList : % -> List(TERM);
19   }
20
21   Implementation ==> add
22   {
23     import from Boolean;
24     import from NNI;
25     import from INT;
26     import from Character;
27     import from String;
28     import from TERM;
29     import from GFAILED;
30     import from NNIFAILED;
31     import from OF;
32
33     Rep ==> Union(object:G, arrow:List(TERM));
34                 -- case G: only object of the graph
35                 -- case List TERM: only reduced "words" either given
36                 -- by a non-empty list of "arrows of G" (satisfying
37                 -- arrow?) with non-zero exponents or by a list of
```

```
38                -- length 1 consisting of an "object of G" with
39                -- exponent 0 (in which case the identity arrow of
40                -- the category is represented).
41
42      import from Rep;
43
44      ((x:%) = (y:%)):Boolean ==
45      {
46        (rep(x) case object) and (rep(y) case object) => rep(x).object =
   rep(y).object;
47        (rep(x) case arrow) and (rep(y) case arrow) =>
48        {
49          xx := rep(x).arrow;
50          yy := rep(y).arrow;
51          not (#xx = #yy) => false;
52          for xterm in xx for yterm in yy repeat
53          {
54            not (xterm.gen = yterm.gen) => return false;
55            not (xterm.exp = yterm.exp) => return false;
56          }
57          true;
58        }
59        false;
60      }
61
62      coerce(x:G):% ==
63      {
64        object?(x)$G => per([x]$Rep);
65        per([[[x,1]$TERM]$List(TERM)]$Rep);
66      }
67
68      retractIfCan(x:%): GFAILED ==
69      {
70        rep(x) case object => [rep(x).object]$GFAILED;
71        xx := rep(x).arrow;
72        empty?(xx) => never;
73        not empty? rest(xx) => [failed]$GFAILED;
74        term : TERM := first(xx);
75        (zero? term.exp) and (object?(term.gen)$G) => [term.gen]$GFAILED;
76        (one? term.exp) and (arrow?(term.gen)$G) => [term.gen]$GFAILED;
77        never;
78      }
79
80      termList(x:%):List(TERM) ==
81      {
82        rep(x) case object => error "arrow required";
83        rep(x).arrow;
```

```
84      }
85
86      object?(x:%):Boolean == rep(x) case object;
87
88      arrow?(x:%):Boolean == rep(x) case arrow;
89
90      source(x:%):% ==
91      {
92        rep(x) case object => error "arrow required";
93        xx := rep(x).arrow;
94        object?(first(xx).gen)$G => per([first(xx).gen]$Rep);
95        xxx := first(xx);
96        positive? xxx.exp => per([source(xxx.gen)$G]$Rep);
97        per([target(xxx.gen)$G]$Rep);
98      }
99
100     target(x:%):% ==
101     {
102       rep(x) case object => error "arrow required";
103       xx := rep(x).arrow;
104       object?(first(xx).gen)$G => per([first(xx).gen]$Rep);
105       xxx := last(xx);
106       positive? xxx.exp =>per([target(xxx.gen)$G]$Rep);
107       per([source(xxx.gen)$G]$Rep);
108     }
109
110     identity?(x:%):Boolean ==
111     {
112       rep(x) case object => error "arrow required";
113       xx := rep(x).arrow;
114       object?(first(xx).gen)$G;
115     }
116
117     identity(x:%):% ==
118     {
119       rep(x) case arrow  => error "object required";
120       xx := rep(x).object;
121       not (object?(xx)$G) => never;
122       termList : List(TERM) := [[xx,0]$TERM];
123       per([termList]$Rep);
124     }
125
126     composeLists(xx:List(TERM), yy:List(TERM)):List(TERM) ==
127     {                                  -- xx and yy are not empty
128       xxlast := last(xx);
129       yyfirst := first(yy);
130       not (xxlast.gen =$G yyfirst.gen) => concat(xx,yy);
```

```
131       newExp := xxlast.exp + yyfirst.exp;
132       zero? newExp =>
133       {
134         xxx := first(xx, subtractIfCan(#xx,1).value1);
135         yyy := rest(yy);
136         empty? xxx => yyy;
137         empty? yyy => xxx;
138         composeLists(xxx,yyy);
139       }
140       xxx := first(xx, subtractIfCan(#xx,1).value1);
141       newterm := [last(xx).gen,last(xx).exp + first(yy).exp]$TERM;
142       append(xxx, cons(newterm,rest(yy)));
143     }
144
145     ((x:%) * (y:%)):% ==       -- this is where we are reducing words
146     {
147       rep(x) case object => error "arrow required";
148       rep(y) case object => error "arrow required";
149       not composable?(x,y) => error "not composable";
150       identity?(x) => y;
151       identity?(y) => x;
152       xx := rep(x).arrow;
153       yy := rep(y).arrow;
154       newList := composeLists(xx,yy);
155       empty? newList => identity(source(x));
156       per([newList]$Rep);
157     }
158
159     inv(x:%):% ==
160     {
161       rep(x) case object => error "arrow required";
162       xx := rep(x).arrow;
163       result := empty()$List(TERM);
164       for term in xx repeat
165       {
166         result := cons([term.gen, - (term.exp)]$TERM, result);
167       }
168       per([result]$Rep);
169     }
170
171     coerceTerm(term:TERM):OF ==
172     {
173       one? term.exp => coerce(term.gen)$G;
174       scripts(coerce(term.gen)$G,
175         [empty()$OF,coerce(term.exp)$INT]$List(OF))$OF;
176     }
177
```

```
178     coerce(x:%):OF ==                    -- coerce to output form
179     {
180       rep(x) case object => coerce(rep(x).object)$G;
181       xx := rep(x).arrow;
182       identity?(x) =>
183         scripts(char("1")::OF,[coerce(first(xx).gen)$G]$List(OF));
184       bracket(hconcat([coerceTerm(y) for y in xx]$List(OF))$OF)$OF;
185     }
186   }
187 }
```

References

[1] R. Brown and A. P. Tonks, Calculations with simplicial and cubical groups in AXIOM, *J. Symbolic Comp.*, 17 (1994) 159-179.

[2] R. D. Jenks, R. S. Sutor, "AXIOM: The scientific computation system", Springer (1992).

[3] L. A. Lambe, Resolutions via homological perturbation, *J. Symbolic Comp.*, 12 (1991), 71-87.

[4] L. A. Lambe, Resolutions that split off of the bar construction, *J. Pure & Appl. Alg.*, 84 (1993), 311-329.

[5] L. A. Lambe, Object-Oriented Mathematical Programming and Symbolic and Numeric Interface, with R. Luczak, 3^{rd} *International Conf. on Expert Systems for Numerical Computing, May, 1993*, in MATCOM 981, Math. and Computers in Simulation 36 (1994), Elsevier Science B.V, The Netherlands.

[6] L. A. Lambe, The AXIOM System, *Notices Amer. Math. Soc.*, Jan., 1994.

[7] L. A. Lambe, Next Generation Computer Algebra Systems, AXIOM and the Scratchpad Concept: Applications to Research in Algebra, 21^{st} *Nordic Congress of Mathematicians, Luleå, Sweden, Summer, 1992* in Analysis, Algebra, and Computers in Mathematical Research, ed. Gyllenberg/Persson, Chapter 14, 201-222, Marcel Dekker, 1994.

[8] L. A. Lambe, Generic Programming: A Tutorial on Object Oriented Mathematical Programming Using AXIOM, Workshop on Symbolic Computation in Advanced Undergraduate Education, Rutgers Univ., June, 1994.

[9] S.M. Watts et al., "AXIOM Library Compiler User Guide", NAG Ltd., Oxford, (1994).