A Database of Categories *

Michael Fleming
Department of Computer Science
University of Waterloo
Waterloo, Ont, Canada

Ryan Gunther
Department of Computer Science
University of Waterloo
Waterloo, Ont, Canada

Robert Rosebrugh
Department of Mathematics and Computer Science
Mount Allison University
Sackville, NB, Canada

ABSTRACT

We describe a program which facilitates storage and manipulation of finitely-presented categories and finite-set valued functors. It allows storage, editing and recall of finitely-presented categories and functors. Several tools for testing properties of objects and arrows, and the computation of right and left kan extensions are included. The program is written in ANSI C and is menu-based. Use of the program requires a basic knowledge of category theory.

*Support of the first two authors by the NSERC Canada USRA program is acknowledged.

1 Introduction

Computation with categories and functors using digital computers has not been widely implemented. The authors believe that a system which allows storage, retrieval, queries and computations on categorical objects will be valuable for research and instruction. For a recent description of much of the work which has been done, for example, to implement categories as a data type, see the posting of D. Rydeheard [4]. Also, R. Brown and collaborators [1] have recently implemented tools for specification of categories, functors and natural transformations using the AXIOM computer algebra system. AXIOM is certainly better adapted to describing categorical algebra than other computer algebra systems and their work is of considerable interest.

Our project began with a prototype using a small relational database system (PARADOX). That provided a suitable environment for storage of categories and functors. We found that all of our queries required programming – the select, project, join and set-theoretic operations of relational algebra were of little help. Thus, while recognizing that in doing so we gave up the set-at-a-time navigation tools of relational systems, we have preferred to implement our queries in a procedural language more adapted to responding to our more complex queries. An object-oriented database system with a strong procedural component might provide another suitable environment for the sort of system we are interested in.

Finitely presented categories, functors between them, and (finite-)set valued functors can be stored in and manipulated by a digital computer. We have written an interactive menu-based ANSI C program which provides methods for storage, retrieval and updating of a database of finitely-presented categories. Tools for queries on properties of the stored categories are also available. The computation of kan extensions of finite-set valued functors (and hence of limits and colimits in finite sets) is also part of the package.

The next section provides a description of the mathematical structures which are implemented and the theoretical tools used. In Section 3 we give a brief overview of usage of the program, displaying some of the menus and their usage.

For the User Guide, source code and DOS executables see the project page available from:
`http://www.mta.ca/~rrosebru`

2 Mathematical structures

We assume that the reader is familiar with the basic definitions of category, functor and so on as found in, e. g., Mac Lane [3]. In this section we define the mathematical objects to be stored and manipulated, and briefly describe the mathematical background for the tools which are available to the user.

The ASCII file format used by our program `category` for storing categories and functors, and some of its data structures are adapted from those of Carmody, Leeming and Walters' `kan` program [2]. Thus files created for use with their program may be used as data files for `category`.

One of the three sorts of objects dealt with by our system is the finitely-presented (FP) category.

A *finite presentation* of a category $\mathbf{C}$ is specified by the following data:

- a finite set C_0 of *objects* denoted $A, B, C, ...$
- a finite set C_1 of *generating arrows* denoted $f, g, ...$ with a domain function δ_0 and codomain function δ_1 to C_0 (so the objects, generating arrows and δ_0, δ_1 determine a finite directed graph whose set of paths is denoted C_1^*)
- a finite set E of relations, or *equations*, between pairs of paths in the directed graph of objects and arrows (and such that equated paths both have both the same domain and codomain objects)

A *finitely presented* category $\mathbf{C}$ is determined by a finite presentation. The category has objects C_0 and arrows given by equivalence classes of C_1^* under the equivalence relation on C_1^* generated by the equations E. For details see Mac Lane [3]. Of course $\mathbf{C}$ has many *equivalent* finite presentations if it has one. It should also be noted immediately that a finitely presented category need not have finitely many arrows. For example, if C_0 and C_1 each have one element and there are no equations then $\mathbf{C}$ has infinitely many distinct arrows.

To store a category externally the program `category` uses a simple ASCII file format that lists the objects, arrows and equations. Users are able to create and store category files, and retrieve stored categories. During creation or after retrieval a category may be modified interactively by changing some of its defining data. Data entry is guided by prompts and entries are validated. For example, domains and codomains of generating arrows are required

to be already stored, and the paths in equations are checked to be valid in the underlying graph.

A *functor* $F : \mathbf{C} \longrightarrow \mathbf{D}$ between finitely presented categories $\mathbf{C}$ and $\mathbf{D}$ is specified by functions $F_0 : C_0 \longrightarrow D_1$ and $F_1 : C_1 \longrightarrow D_1^*$ where D_1^* is the set of paths in the graph of $\mathbf{D}$. The function F_1 is subject to the requirements (1) that the domain and codomain of the image of an arrow must be the image of the domain and codomain of the arrow, and (2) that F_1 is compatible with composition. The second property can be finitely checked by comparing images of the pairs of paths appearing in the equations defining $\mathbf{C}$.

The data for a functor are stored in an ASCII file. Users are able to create, store and retrieve functor files. The creation of functors is guided by prompts to the user and the responses are validated.

Finite-set valued functors from FP categories to the category of finite sets $\mathbf{set}_0$ may be created, stored and retrieved. We represent finite sets by their finite cardinals e. g. $\mathbf{n} = \{1, 2, ..., n\}$, so strictly speaking our functors take values in the finite cardinal skeleton of finite sets. Once again, the creation of these functors is guided by prompts to the user and the requirements of functoriality are validated.

The storage and retrieval tools outlined above are intended to provide a user with the capability of building a database of categories of interest. We have also developed several demonstration tools for appropriate queries on the stored categories, and for working with finite-set valued functors. We describe these now.

One of the simplest questions one might ask about a FP category is whether two paths in the underlying graph represent equal arrows. To answer this query it is convenient to have a normal form for paths available and an algorithm to reduce paths to the normal form. The Knuth-Bendix procedure (described in detail by Walters in [5]) often provides such an algorithm, and we give a brief summary.

The objective of the Knuth-Bendix procedure is to replace the equations for a FP category with a confluent set of reductions which presents the same category. Reductions are defined with respect to an order specified on paths in the underlying graph (this can be done by ordering generating arrows and extending lexicographically.) A *reduction* is simply an equation viewed as a replacement rule of a larger path (in the order) by a smaller one. It is *applied* to a path which contains the larger path by rewriting it using the smaller path from the reduction. A path is *irreducible* with respect to a set

of reductions if no reduction is applicable. A set of reductions is *confluent* if every path reduces to a unique irreducible *normal form*.

For an FP category presented with a confluent set of reductions the question of equality for paths is settled by comparison of normal forms. Confluent sets of reductions are characterized by satisfaction of two easily checkable properties detailed in [5]. The Knuth-Bendix procedure applied to a set of reductions R halts when R is confluent. If R is not confluent, the failure of one of the properties mentioned generates an equivalent pair of irreducible paths. The reduction from the larger to the smaller of these is added to R and the new set of reductions is checked for confluence. When the procedure terminates it has produced an equivalent confluent set.

The Knuth-Bendix procedure is implemented in `category` and replaces the equations of an FP category with a confluent set of reductions. (The order used on generating arrows is that of their first entry.) The other tools available assume that this has been carried out.

The simplest query checks for equivalence of paths, that is equality of the represented arrows, in the underlying graph. Queries are also implemented to determine (1) if an object is an initial object of a stored category and (2) if a cospan of arrows, i. e. a diagram of the form $A \longrightarrow C \longleftarrow B$, in a stored category is a coproduct diagram. The construction of a finite presentation of the opposite of a FP category from a finite presentation has been implemented, so the latter two queries mentioned can also determine whether an object is terminal and whether a span is a product.

There are some points to make about these procedures. The first is that there is in fact an algorithm to determine whether an object in a FP category is initial even if the category is infinite. Indeed, it is only necessary to consider paths in the underlying graph which are loop-free. We have the following.

Proposition 1 *Let* $\mathbf{C}$ *be a finitely presented category. The object* I *is an initial object of* $\mathbf{C}$ *if and only if all of the following conditions hold:*

i) there is a loop-free path from I *to each object of* $\mathbf{C}$

ii) if π_1 *and* π_2 *are loop-free paths from* I *to an object* A *in the underlying graph, then* π_1 *is equivalent to* π_2

iii) whenever there is a loop λ *on an object* A *in the underlying graph such that* λ *itself is loop-free, and any loop-free path from* I *to any object of* λ *(except* A*) passes through* A*, then* $\lambda\pi \sim \pi$ *for any path* π *from* I *to* A*.*

Proof. The conditions are all clearly necessary, so we need only demonstrate sufficiency.

Our objective is to show that there is a unique arrow from I to any object. If the first condition is satisfied, we need only show that any two paths from I to the same object are equal. If the second condition is satisfied, we need only show that any path from I which contains one or more loops represents the same arrow as a loop-free path from I. We will show that the third condition accomplishes this under the assumption that the first two are satisfied.

Suppose that $\pi = \pi_2\lambda\pi_1$ is a path from I and λ is loop. If $\lambda\pi_1 \sim \pi_1$, then $\pi_2\lambda\pi_1 \sim \pi_2\pi_1$. Thus it is sufficient to consider paths from I containing only a single loop λ which includes the final object on the path. So λ is a loop-free loop. We can also restrict consideration to paths from I which are loop-free before they first encounter the final object.

Now suppose that $\lambda\pi$ is a path from I to A with λ a loop-free loop and π loop-free. If the third condition is satisfied by λ then $\lambda\pi \sim \pi$. If not, there is a loop-free path π' to an object B ($\neq A$) on λ. Thus λ decomposes at B as $\lambda = \lambda_2\lambda_1$ and we have $\lambda\pi = \lambda_2\lambda_1\pi \sim \lambda_2\pi' \sim \pi$ since $\lambda_1\pi$ and π' are loop-free paths from I to B, and $\lambda_2\pi'$ and π are loop-free paths from I to A. In either case, $\lambda\pi \sim \pi$ as required. ∎

The proposition allows the construction of a straightforward path traversal algorithm to determine whether an object is initial.

Our second comment concerns coproducts. A cospan $i : A \longrightarrow C \longleftarrow B : j$ is a coproduct diagram if there is a bijection between arrows f from C to an arbitrary object T, and pairs (fi, fj) of composites of f with i and j. If any of the hom-sets from C is infinite there is no method to determine whether such a bijection exists. However there is no algorithm to determine that all of these hom-sets are finite for a general FP category. Nevertheless, if we know that all of the endomorphisms of an FP category are of finite order then the FP category is finite. Thus we have provided a program parameter which restricts consideration to endomorphisms of a fixed finite order, and the user is responsible to ensure that this order is sufficient to capture all of the arrows of the FP category under consideration. The tree which results from 'unfolding' paths from a fixed object in a directed graph is the data structure used in enumerating the hom-sets from the fixed object. This data structure has proved to be efficient and is also used in our right kan extension computation as described below.

Kan extensions are a fundamental construction in category theory. Given

a functor $F : \mathbf{A} \longrightarrow \mathbf{B}$, its left and right kan extensions exist along any **set** valued functor $X : \mathbf{A} \longrightarrow \mathbf{set}$ (or indeed along any functor from $\mathbf{A}$ to a complete and cocomplete category.) They are the functors $L, R : \mathbf{B} \longrightarrow \mathbf{set}$ satisfying

$$\frac{L \longrightarrow K}{X \longrightarrow KF} \qquad \frac{S \longrightarrow R}{SF \longrightarrow X}$$

where the arrows between functors are *natural transformations.* Thus, corresponding to the identity natural transformation, there are also transformations $\lambda : X \longrightarrow LF$ and $\rho : RF \longrightarrow X$. On objects B of $\mathbf{B}$, L and R can be computed by the formulae:

$$LB = \mathrm{colim}_{f:FA \to B} X(A) \qquad RB = \lim_{g:B \to FA} X(A)$$

in which the (co)limits are taken over the comma categories F/B and B/F respectively.

We have implemented computation of both left and right kan extensions of functors $F : \mathbf{A} \longrightarrow \mathbf{B}$ between FP categories along finite set valued functors $X : \mathbf{A} \longrightarrow \mathbf{set}_0$. Our implementation of the left kan extension uses the elegant Todd-Coxeter procedure as described in [2] to provide an efficient construction of L and λ, so we will not comment further on that side. However we remind the reader that the left kan extension allows a computation of all of the arrows of (finite) FP category and of colimits of finite set valued functors (from FP categories,) and that right kan extensions allow computation of limits of the same functors.

To compute the right kan extension at an object involves computation of a limit of finite sets. We represent limits as subsets of products. Since enumeration of elements of a product is potentially costly, we have adopted two simple strategies to ameliorate this. First, instead of taking the product of all objects in B/F, we find an initial subcategory of B/F. This is done by enumerating objects of B/F (that is arrows of $\mathbf{B}$ from B to an object of the form FA) and adding an object to our initial subcategory only when there is no arrow (of B/F) to it from an already listed object. The required limit is a subset of the product of the values of X at the objects of the initial subcategory. Second, it is not necessary to enumerate all of the elements of the resulting product. For example, suppose that $x \in X(A), y \in X(A')$ and $B \xrightarrow{f} FA, B \xrightarrow{f'} FA'$ are objects in our initial subcategory. The only tuples of the form $(x, y, \ldots)$ which need be considered are those where $X(g)(x) = y$ for all $g : A \longrightarrow A'$ satisfying $F(g)f = f'$. Finally, we note that the required

natural transformation $\rho : SR \longrightarrow X$ is specified using projections from the values of R computed on objects.

This concludes our discussion of the tools implemented by `category`. We make some remarks about possible extensions in the Conclusion.

3 System description

The following is an abbreviated description of the interactive implementation of the capabilities described above. The system is written in ANSI C and a description of the data structures and algorithms used is available (see the third author's Web page noted in the Introduction).

3.1 The Main Menu

When the program `category` starts the Main Menu is displayed:

```
                Categories Database

        (1)  Category Menu
        (2)  Functor Menu
        (3)  Category Tools
        (4)  Right Kan Extension
        (5)  Left Kan Extension
        (6)  Change maximum order of endomorphisms

        (0)  Quit

Your choice...
```

The menu options call up another menu, or prompt the user to enter input. The first, third and fifth options are discussed below. Note that on choosing `(0) Quit` the program will ask the user if they wish to save each category or functor currently in memory, and the option `(6) Change maximum number of endomorphisms` allows the user to control the maximum number of times an endomorphism will be traversed by the tools in the program. The default value is 2. Most of the manipulation tools below require a finite category.

Selecting option `(1) Category Menu` from the Main Menu will display:

```
CATEGORY MENU

(1)  Create category
(2)  Load category
(3)  Edit category
(4)  Display category
(5)  List current categories
(6)  Save category
(7)  Remove category

(0)  Back to main menu
```

If option `(1) Create Category` is now chosen the following prompt will be displayed:
`Category name:`
After the user types in the name of the new category they will see:
`Enter @ to display all objects`
`Enter object name (type 'enter' when finished) :`
After object entry is complete, the following will be displayed:
`Enter arrow name (type 'enter' when finished) :`
After the name of an arrow has been entered, the program will ask the user to input the domain of the arrow and then the codomain. The symbol `1` is reserved for identity arrows.

Next the user enters the equations of the category starting from the prompt:
`Enter left side of equation:`
To enter an equation, the left side of the equation is entered followed by the right side. If some path of arrows in the category equals the identity, that path is entered for the left side and `1` for the right side. Once the equations have been entered, typing `Enter` for both the left and right sides of the equation will bring back the Category Menu.

Choosing option `(2) Load Category` will display the following prompt:
`Enter name of category you wish to load>`
The name of the file containing the category to load is entered and the program will load the category, or display an error message.

The option `(3) Edit Category` calls up a menu which allows changing objects, arrows or equations for a category stored in memory.

The option `(4) Display Category` will prompt the user to select a category from memory to be shown on the screen.

Choosing the menu option `(5) List current categories` will display a list of the categories that are currently in memory.

The selection `(6) Save Category` prompts the user to select a category to be saved, and asks for a file name in which to store it.

The selection `(7)Remove Category` will prompt the user to select a category to be removed from memory.

3.2 The Category Tools Menu

Choosing option 3 from the Main Menu, `Category Tools`, will display a list of the categories currently in memory. After providing the number of a category the `Category Tools Menu` is displayed:

```
                    CATEGORY TOOLS

                (1) Make Confluent
                (2) Initial Object?
                (3) Equality of Composites
                (4) Make Dual
                (5) Sum?
                (6) Display Category

                (0) Exit to Main Menu

  Your choice...
```

The options are:

`(1) Make Confluent`

Choosing this option will make the set of equations in the current category confluent by adding new equations if necessary.

`(2) Initial Object?`

After displaying the current category *which must be made confluent,* this option allows either testing all objects in the category, or one specific object.

`(3) Equality of Composites`

This option will determine if two composable paths are equal. The user is prompted to enter two paths.

`(4) Make Dual`

This will create the dual (opposite) of the current category and store it.

`(5) Sum?`

This will determine if an object and two paths, the candidate injections, into the object are a sum diagram in the category.

`(6) Display Category`

This displays the active category.

3.3 Left Kan Extension

Option (5) from the Main Menu, `Left Kan Extension`, will allow the user to compute a left kan extension.

The user will first see a list of categories, and is asked to select a category **A** and a category **B** for the left kan extension. Next a list of functors is displayed and the user is asked to select a functor from the category **A** to the category **B**, followed by a functor from **A** to the category of finite sets. The user is then asked to enter a file name in which to store the output.

The output of the left kan extension begins with information about the natural transformation $\lambda : X \to LF$. For each object A in **A**, the function $\lambda_A : XA \to LFA$ is displayed. Next, the action of the left kan extension L on the objects and arrows of **B** is displayed. Each object B in **B** is displayed with all of the elements of $L(B)$ listed below it. To the right will appear all generating arrows out of B with their action under L tabulated.

4 Conclusion

We have described a system for storage and manipulation of FP categories and computation of kan extensions of finite-set valued functors. Our system is user-friendly and publicly available. It implements some important features, but there are many additional desirable features which could be added and we list a few of them:

- checking that an object is some other (co)limit than those already implemented;
- checking whether a category is actually a preorder;
- checking whether a pair of functors between FP categories are adjoint or an equivalence;

- checking whether a functor between FP categories is full, faithful or both;
- construction of (co)limits of FP categories
- a graphical display of stored categories and functors;
- a graphical interface for specification and manipulation of categories and functors.

We believe that the data structures adopted and the tools already implemented demonstrate that these features are feasible and look forward to future work in this area.
ADDED NOTE: A recent Java implementation of similar tools with a graphical interface is available from the GDCT Project pages (see the third author's Web page noted in the Introduction).

References

[1] R. Brown. The AXIOM computer algebra system applied to computational category theory. Lecture at the 2'nd IMACS Conference, RISC, Linz, 1996.

[2] S. Carmody, M. Leeming, and R.F.C. Walters. The Todd-Coxeter procedure and left kan extensions. *Journal of Symbolic Computation*, 19:459–488, 1995.

[3] S. Mac Lane. *Categories for the Working Mathematician*. Springer Verlag, 1971.

[4] David Rydeheard. Re: Mechanization of category theory, 1996
`http://www.mcs.anl.gov/qed/mail-archive/volume-3/0139.html`

[5] R. F. C. Walters. *Categories and Computer Science*. Cambridge University Press, 1991.