Modelling algebraic structures with Prolog
extended abstract of full paper for CANT'95

Richard Buckland and Michael Johnson *

School of Mathematics and Computing, Macquarie University
AUSTRALIA

Abstract

This paper presents a novel technique of using Prolog with never instantiated variables to manipulate a range of algebraic structures. The paper argues that Prolog is a powerful and underrated tool for use in computational number theory. A detailed example is presented in this extended abstract, and several in the full paper, showing the advantages of using this technique. The detailed example is an application of higher dimensional category theory which has been used for solving problems in this area.

1 Introduction

Among the many problems dealt with in computational algebra two important classes of problems deal with enumerating algebraic structures with certain properties and manipulating or calculating with the elements of such structures. For particular algebraic structures very efficient solutions to these problems are known and such solutions are typically made available as parts of one or more of the large computational packages now available (which include for example Cayley [?], Maple [?] and Mathematica [?]).

Rather than adding to the list of specific solutions this paper sets out a methodology for using Prolog (a declarative logic programming language) to solve a range of enumeration and calculation problems. The rationale for describing such a methodology is that in developing computational software two major obstacles are the time required to code the solutions and the need to

*The authors gratefully acknowledge a Microsoft Fellowship (awarded to the first author) and the support of the Australian Research Council (ARC).

develop confidence in that code. By using the techniques described here one is unlikely to obtain an optimally efficient solution to the problem but coding times are frequently very short and most of the detailed manipulations required are carried out by Prolog's internal mechanisms which have been well validated. In addition the remaining code is typically short and clear making it easy to argue for the correctness of the solution. Furthermore, for very specific problems and for exploratory investigations it may not be prudent to invest a great deal of time in developing solutions in imperative programming languages. Rather one needs quick prototyping techniques such as the methodology outlined here.

The techniques described below are most useful for constructing or enumerating algebraic structures, perhaps those satisfying various extra constraints, and for checking for isomorphism. They have already been used in several applications and in what follows we will trace a particular application to higher dimensional computational category theory where the technique has been particularly useful.

In Section ?? we give a brief overview of the methodology and outline the advantages of using Prolog. Probably the most distinctive aspect of the methodology is the use of never-instantiated variables which allow us to check easily for isomorphism by using Prolog's internal unification mechanism while providing sufficient information to permit the identification of inappropriate bindings. In Section ?? we then present an example which illustrates the effectiveness of this technique. Never-instantiated variables are discussed in detail in Section ??.

2 The methodology

When representing sets with algebraic structure in computations it is often convenient to use terms. Such terms typically contain several arguments all of which denote a common element of the set. For example, if we represent a four element monoid by constructing a term for its multiplication table then this term will contain 16 arguments, each of which will represent one of the four elements of the monoid and several arguments will represent the same element. The equivalence relation 'represents the same element' must therefore be captured by the term structure in some manner in order to have a true representation.

In a typical programming solution this relation would be implicitly specified by having variables for the arguments and setting them equal to the same atom if and only if they represent the same element. In logic programming terminology we would instantiate them to atoms corresponding to the elements they represented.

This paper however proposes an alternative approach. Instead of instantiating variables we leave them uninstantiated and use bindings to provide the relation (note that 'are bound' is an equivalence relation). In the case of the above monoid for example the term representation would comprise 16 uninstan-

tiated variables (probably arranged in some meaningful manner) and any two variables X and Y would be bound if and only if they represented the same element of the monoid, and the bindings would never be grounded. The use of such *never instantiated* variables is a somewhat unorthodox use of Prolog's binding mechanism but provides significant computational advantages in detecting isomorphism.

Where the structural information is such that a unique normal form exists for each term then terms will be isomorphic when these normal forms can be unified. If a single normalised term does not exist then two structures are isomorphic if the term for one is unifiable with any of the term forms of the second. A four element monoid will have 3! term forms since non-identity elements can appear in any order. Obviously the test will be simplest when there is a single normal form. In other cases the unification test may well be expensive, but no more than any other generalised test.

Using never instantiated variables also provides a simple and elegant way of constructing algebraic structures. This is elaborated in the example application described in the following section.

The use of Prolog in processing algebraic structures has certain other advantages. In particular backtracking is ideal for enumerations and searches of solution spaces. Furthermore the logical basis of the language permits, in a natural way, the constraining of solutions or the finding of counter examples to ad hoc conjectures on the fly. Finally, as mentioned above, its declarative nature leads to short, straightforward programs which tend to map closely to the problem statement and this facilitates correctness. In our opinion Prolog is an ideal and underrated language for performing many algebraic tasks.

3 An example: working with pasting schemes

Intuition often outstrips definition. This has certainly been the case in the theory of n-dimensional categories (n-categories). *Pasting diagrams* were used in 2-categorical work nearly three decades ago but it was not until work of Street [?], and Johnson [?] that a formal theory of *pasting schemes* emerged. Once the theory was in place a number of new applications appeared including models of concurrency (Pratt [?] and Buckland [?]) and a new approach to the homotopy classification problem (Kapranov and Voevodsky [?]).

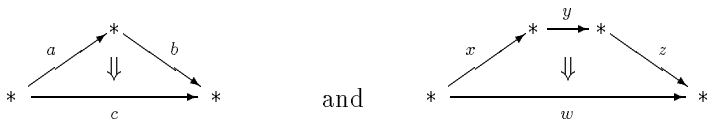
Unfortunately there are two major limitations when working with pasting schemes. Firstly the manipulation of pasting schemes, except in very low dimensions, is notoriously difficult. More fundamentally, the current definitions of pasting schemes are complicated and unwieldy. It seems that computer assistance is needed both to work with the schemes and to give information to aid in the development of new and elegant definitions. We report below a Prolog representation of pasting schemes which is being used for both these purposes.

To give the reader some feeling for pasting schemes, the following is an

informal description of them. Precise details can be found in [?] where the schemes we are dealing with are called *well-formed loop-free pasting schemes*. There are also a large number of pictures of pasting schemes in [?].

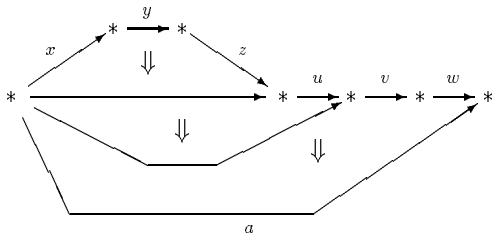
A *pasting scheme* is essentially a uniformly directed piece of space of some dimension, say n.

An n-dimensional pasting scheme is made up of a number of *n-cells*. Each cell is an 'atomic' piece of directed space of dimension n which is bounded by two uniformly directed pieces of $(n-1)$-dimensional space. For example, when $n = 2$,



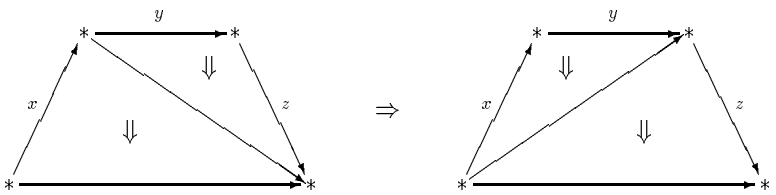
are cells. The $(n-1)$-dimensional boundary at the beginning of the n-cell is called its *domain* (a, b in the first case and x, y, z in the second) and the boundary at the end of the n-cell is called its *codomain* (c and w respectively).

Cells may be pasted together to form a pasting scheme, provided that their orientations agree and that the resultant scheme has global domain and codomain which are schemes of one lower dimension. For example, three 2-cells can be pasted together to form the scheme



whose global domain is x, y, z, u, v, w and whose global codomain is a.

An example of a 3-dimensional scheme which is made of a single 3-cell is



which is actually a tetrahedron which has been broken in half for ease of representation. The horizontal double arrow represents the directed 3-dimensional space inside the tetrahedron.

3.1 The Prolog representation

Representing and manipulating pasting schemes of arbitrary size and dimension previously posed computational difficulties because of the volume of information required to define complex objects of high dimension. However the possibility of representing such objects as recursive terms led us to consider the use of Prolog which was found to tackle the problem with ease and with a surprisingly small amount of code and manageable data structures. The declarative nature of Prolog also helped in establishing confidence in the code, which was essential in working with high dimensional shapes which are difficult to visualise.

The recursive structure of cells (each n-cell ($n > 0$) is bounded by cells of dimension $n-1$), suggests a Prolog representation by terms. The chosen form of a cell term is

`cell(Name, Dim, Domain, Codomain)`

where `Name` is an identifier for the cell, `Dim` is a positive integer and `Domain` and `Codomain` are lists of cell terms of dimension `Dim - 1` (or '`nil`' if `Dim = 0`). The lists are ordered in a standard way so that unification of cell terms is efficient.

A cell is represented as a Prolog compound term with structure as set out above. An n-dimensional pasting scheme is an ordered list of the terms of its n-cells. The `Domain` and `Codomain` terms of an n-cell ($n > 0$) are themselves pasting schemes of dimension $n-1$.

Simply being able to represent cells is useful, but the real power derived from using Prolog is the fact that Prolog itself is able to look after some of the major tasks which arise in dealing with these high dimensional objects. These are

1. Keeping track of which cells in a pasting scheme are distinct and which are the same
2. Testing objects or segments of objects for isomorphism with linear efficiency
3. Generating schemes by pasting n-cells together across common k-dimensional faces.

In a procedural language these tasks would be dauntingly difficult to implement, yet they are provided almost for free by Prolog — and with linear efficiency in terms of Prolog primitives.

These tasks are made possible by the inclusion of the `Name` argument for each cell. *The* `Name` *argument is never instantiated.* This enables us to consider a cell term as modelling an *unlabelled* n-dimensional object. Thus a term represents an equivalence class of isomorphic cells.

The following paragraphs describe how each of these main tasks was implemented taking advantage of Prolog's inbuilt mechanisms.

Keeping track of which cells are distinct and which are the same

This problem arises because the component k-cells of an n-cell ($k < n-2$) typically occur many times in the n-cell term in various roles. For example, the zero dimensional cells (points) in the examples above occur as the domain or codomain of more than one of the 1-cells. It is important to be able to know whether two k-cell terms occurring in an n-cell term represent two distinct k-cells. As discussed in section ?? above we capture this information by binding the `Name`s of two k-cell terms if they represent the same k-cell. If the `Name`s are not bound then they represent distinct k-cells.

Testing for isomorphism

Keeping `Name`s uninstantiated makes it straightforward to compare two objects to see if they are isomorphic. This is because each scheme has a canonical term representation so isomorphism testing is a straightforward syntactic operation — checking for the existence of a most general unifier. So we can simply attempt to unify the two terms and if the unification succeeds they are isomorphic (subject to the comments in Section ??).

However if cell `Name`s were instantiated then this simple approach could not be used as the terms would only be unifiable if the corresponding objects have both the same structure and the same cell `Name`s, and comparing shapes would be a complex operation.

Pasting cells together

Dealing with pasting schemes presents some specific computational problems. In particular it is often useful to 'paste' two such schemes together to form a new scheme. The representation methodology described above transforms this to a trivial operation in Prolog. To paste pairs of schemes together to form composite schemes or indeed to use other constructors and form higher dimensional cells we unify the k-cell terms of the common k-dimensional interface. This binds the `Name`s of the i-cells (for all $i \leq k$) common to both schemes wherever they occur.

This means that the task of pasting n-dimensional objects together across k-dimensional faces which was previously considered computationally complex is implemented simply by unifying the terms of the common k-dimensional faces. Furthermore the necessary precondition that the faces across which pasting is to occur be isomorphic is simultaneously and elegantly checked as it is the precondition for the unification to succeed. It turns out that the entire suite of procedures for manipulating pasting schemes is a single page of code!

4 Notes on never-instantiated-variables

Binding provides a mechanism to specify that two variables are standing for the same entity (even if that entity is never instantiated). However there is no equivalent mechanism to assert that two variables are different. In traditional Prolog programs this is handled by the backtracking mechanism, for, if essentially distinct variables become bound at some stage the later attempt to instantiate the variables to non-unifiable terms will fail and the bindings will be undone. However, as the cell `Name`s are never instantiated, distinct `Name`s can unify, and thus erroneously become and remain bound. In the case of pasting schemes this amounts to identifying two distinct parts of one of the n-dimensional shapes and so it is called *distortion*

One possible way of overcoming this problem is to explicitly specify that certain variables should never be bound. Many modern implementations of Prolog allow such constraints to be specified using built in predicates, for example the `diff` predicate of SICStus Prolog. This approach is computationally expensive however, as 'being distinct' is not an equivalence relation, and maintaining such relationships is an exponential task. This approach could also make it difficult to legitimately identify previously distinct cells when constructing quotients of structures, or constructing composite structures such as the pasting constructions of the above example.

Alternatively we can overcome the problem by simply being careful in our treatment of never instantiated variables. There are two issues which then must be considered. Firstly one must use some sort of meta-predicate when determining if two cell `Name`s are the same. That is, one must test to see if the variables are bound rather than simply attempting unification which will succeed for any two uninstantiated variables, bound or not.

Secondly, any unification must be attempted with care due to the possibility of introducing distortion. Provided that one remembers to guard against it, distortion is easily avoided. In our pasting scheme application we count the number of distinct cell `Name`s in the cell term both before and after unification. If the number has reduced then distortion has been introduced and the call which required unification is failed.

5 Future work

Finally, the development of the representation has implications for an alternative approach to defining pasting schemes, the *constructive pasting schemes* [?].

References

[1] W. Bosma and J.J. Cannon, *A language for group theory*, Springer Verlag (1992).

[2] R. Buckland, Modeling concurrency with dimension, in preparation, (1994).

[3] R. Buckland and M. Johnson, Constructed Pasting Schemes, in preparation, (1994).

[4] B.W. Char, K.O. Geddes, et al, *Maple V*, Springer Verlag (1991).

[5] M.S.J. Johnson, The combinatorics of n-categorical pasting, *Journal of Pure and Applied Algebra*, **62** (1989) 211–225.

[6] Kapranov and Voevodsky, The homotopy classification problem, preprint, 1991.

[7] V. Pratt, Modelling concurrency with geometry, preprint, (1991).

[8] R.H. Street, The algebra of oriented simplexes, *Journal of Pure and Applied Algebra*, **49** (1987) 283–335.

[9] J.Wolfram, *Mathematica: A system for doing mathematics by computer*, Addison Wesley (1991).

email: mike@macadam.mpce.mq.edu.au