Programming with Algebraic Structures: Design of the Magma Language

Wieb Bosma
John Cannon
Graham Matthews
School of Mathematics and Statistics, University of Sydney, Sydney, NSW 2006, Australia

Abstract

MAGMA is a new software system for computational algebra, number theory and geometry whose design is centred on the concept of algebraic structure (magma). The use of algebraic structure as a design paradigm provides a natural strong typing mechanism. Further, structures and their morphisms appear in the language as first class objects. Standard mathematical notions are used for the basic data types. The result is a powerful, clean language which deals with objects in a mathematically rigorous manner. The conceptual and implementation ideas behind MAGMA will be examined in this paper. This conceptual base differs significantly from those underlying other computer algebra systems.

1 Introduction

1.1 Structural Computation

It is instructive to classify algebraic computations roughly according to the degree of abstraction involved. Thus, a *first order* computation consists entirely of calculations with *elements* of some algebraic structure; the structure itself can be essentially ignored. A *second order* computation involves explicit calculation with *structures* and substructures. Finally, a *third order* computation is one in which the operations involve entire *categories* of algebraic structures. Although a structural computation will ultimately reduce to a series of element calculations, it will usually involve a great deal of effort unless support is provided by the programming language.

The paradigm that has driven most research in algebra over the past century has been the idea of precisely *classifying* all structures that satisfy some (interesting) set of axioms. Notable successes of this approach have been the classification of all simple Lie algebras over the field of complex numbers, the Wedderburn classification of associative algebras, and the very recent classification of finite simple groups. Classification problems typically concern themselves with categories of algebraic structures, and their solution usually requires detailed analysis of entire structures, rather then just consideration of their elements, that is, *second order* computation.

Examination of the major computer algebra systems reveals that most of them were designed around the notion of *first order* computation. Further, systems such as Macsyma, Reduce, Maple [6] and Mathematica [13] assume that all algebraic objects are elements of one of a small number of elementary structures. Indeed, most objects are assumed to live in a fixed differential ring. Calculation in a non-elementary structure is often fudged by setting a flag. For example, the global MOD flag in Macsyma is used to simulate arithmetic in $\mathbf{Z}/m\mathbf{Z}$ (often with strange results since the current value of MOD affects every integer calculation). In Maple, calculation in a residue class ring is achieved by setting a parameter on each function invocation. For the purpose of doing calculus in a fixed structure this may be a valid approach. However, the power of modern algebra comes from the interaction between (elements of) several algebraic structures together with the precise identification of the axiom system to which each object belongs. A major advance was the introduction of *domains* and *categories* in Axiom [9], which provided the basis for a strongly typed system. (See also Santas [11]). However, Axiom does not support algebraic structures as first class objects.

Two fundamental prerequisites for serious computation in the more advanced parts of algebra are:

1. A strong notion of type.
2. First class status for algebraic structures and their homomorphisms.

ISAAC 94 - 7/94 Oxford England UK

1.2 Cayley and Magma

Cayley [4] was the first widely used Computer Algebra language to provide an environment for computing with certain structures (primarily *groups*) and their homomorphisms. The success of Cayley stemmed, in large part, from its exploitation of (1) and (2) above, albeit for a limited class of structures.

Many of the ideas, initially developed in Cayley for computation with groups, were seen to be generally applicable to algebraic and geometric structures. We therefore undertook the design of a general language for algebraic computation based on a 'theory of structural computation' which, in contrast to traditional systems, emphasised computation with algebraic structures. Because of limitations in the original design of Cayley, the new language was conceived not as an extension of Cayley, but rather as a new language ([2], [3]).

The use of algebraic structure as an organizing principle for algebraic computation led us to develop the outlines of what might be called 'algebraic programming', whereby the algebraic structure plays a role analogous to, say, the role of the function in functional programming. These ideas also provided a model for the implementation of MAGMA, just as the theory of categorical combinators provides both a semantics for the CAML system and a basis for the CAM implementation of the system [7].

1.3 Summary of Objectives

- The system would be designed for those areas of mathematics that are algebraic in nature. Since a given mathematical object is often **viewed** quite differently by workers from distinct branches of mathematics, there is no reason to expect that the view of, say polynomials, presented in a successful system for calculus is likely to prove satisfactory for algebraists. We proposed to take an explicit view of algebra, and develop a system in accordance with it. Our view asserts the primacy of the structure and structure-preserving mappings – we shall refer to this as the **structural view**.

- A high-level programming language (the MAGMA language), designed both as an implementation language for large programs and as the interactive interface for the system, would be provided. The system would be implemented partly in C and partly in the MAGMA language.

- The MAGMA language would adopt a strong notion of type and would provide user-defined types.

- Algebraic structures and their morphisms would be first-class objects in the language.

- The MAGMA language would be based on fundamental concepts from algebra, using notation as close as possible to standard mathematical practice. For example, we chose to support 'mathematical' data structures such as sets, sequences and mappings, rather than adopting more typical computer science data structures such as lists, trees, or even pointers. Our view is that the most natural way of specifying an algebraic algorithm is in terms of these mathematical notions.

- The system would allow complete integration of three key types of knowledge, namely, algorithmic knowledge (the library of algorithms), static knowledge (data bases of useful knowledge), and dynamic knowledge (properties and relationships remembered in the course of computation).

- Efficiency was to be a paramount concern since the system was intended to be capable of tackling hard problems. The basic machinery for computationally important classes of algebraic structures (mainly particular families of groups, rings, fields, modules and algebras) would be hand-coded in the C kernel.

- The system kernel would be constructed on top of a low-level 'standardized platform' thereby allowing us to 'plug in' C code written by people outside the Magma group. By importing code written by the experts in a particular field, we could include within the system optimal implementations of many important algorithms at the cost of a small amount of effort on our part.

2 Algebra of Structures

2.1 Magmas

As the design of MAGMA is intended to reflect the structural view of algebra, we take as fundamental the notion of an algebraic structure. For brevity, we use the term **magma** when referring to an algebraic structure. The name magma was introduced by Bourbaki[1] to denote a set with a law of composition.

How does one represent a magma? A *concrete* magma is one defined over an explicit carrier set, whereas an *abstract* magma is one in which the carrier set is not specified. It is clear that actual computation must take place in a concrete magma. We now assume that our (concrete) magmas are finitely generated. Since most interesting infinite algebraic structures having a developed theory are finitely generated, this restriction is not

[1] N. Bourbaki, *Algèbre 1*, p. 1: "Un ensemble muni d' une loi de composition est appelé un magma."

particularly onerous. A concrete magma will be represented by a finite set of generators.
The following organizational principles underlie the design of MAGMA:

- Every object x that may be defined in MAGMA belongs to a unique magma (called the parent of x).
- A magma will be represented in terms of a finite set of generating elements.
- Magmas are organized into **categories** where a category corresponds to a family of magmas sharing a common representation (such as the class of all polynomial rings).
- A collection of categories of magmas whose laws of composition all satisfy a common set of identical relations form a **variety**.

Every object definable in MAGMA, is either a magma or is defined in terms of a magma. Thus, objects such as matrices, vectors and polynomials may only exist (indeed are only definable) in the context of a magma of which they are considered elements. For example, a polynomial in the indeterminate x with integer coefficients, is regarded as an element of $\mathbf{Z}[x]$.

2.2 Categories

A collection of magmas belonging to the same variety and which share a common representation, form a 'category'. While MAGMA categories usually correspond either to categories or indexed families of categories in the sense of Category Theory, it will sometimes be convenient to apply the term to a family of magmas which do not form a category in the strict sense. The category to which a magma belongs determines each of the following:

- The representation of the elements of the magma.
- The representation of the carrier set of the magma.
- The operations which may be performed on elements of the magma.
- The operations which may be performed on the magma.

Every object has associated with it a pointer to its parent. Thus, an element refers to its parent magma, while a magma refers to its parent category. The data structure representing a magma A contains the information needed to completely specify A, and possibly a representation of its carrier set. The data structure representing a category C contains all of the functions and operators (i.e., functions written in infix format) that may be applied to magmas belonging to C.

The notion of a variety provides us with a framework for designing a generic method for defining a magma. Because of space limitations, we can only present a very simplified version here. Consider a class of algebras forming a variety. Now these algebras are closed under the formation of subalgebras, homomorphic images and Cartesian products. The class of magmas that constitute a particular category will be parameterized in some way. For example, a polynomial ring is parameterized by its coefficient ring and number of indeterminates. For a given choice of parameters, there always exists a unique 'free' algebra having those parameters. When a category is installed, a function is provided to create this free algebra, given particular values for the parameters. Now theory tells us that any algebra belonging to the category and having the given parameters can be obtained from the free algebra by application of one or more of the three variety operations. In the case of categories of magmas which do not form a variety, some minor variation of the above ideas will usually work.

Magma provides both generic and category-specific constructors for creating magmas. The generic constructions include the following:

- A free magma constructor.
- A submagma constructor that takes an existing magma M together with a set X of elements of M and creates the submagma generated by X.
- A quotient magma constructor that takes an existing magma M together with a set X of elements of M and creates the quotient of M by the ideal generated by X.
- A constructor that forms an extension of a magma by some other magma (the form of this is rather dependent upon the variety to which the magmas belong).

2.3 Mappings

Of equal importance to the concept of a magma, is the concept of a mapping between magmas, especially structure-preserving mappings (morphisms). Mappings are used to represent the following types of associations:

- A natural relationship holding between two magmas (e.g. inclusion).
- A general homomorphism between two magmas.
- An endomorphism of a magma.
- An action of magma A on magma B.

- An association between two sets.

How do we represent mappings? A theorem from Universal Algebra states that in the case of a class of magmas forming a variety V, a homomorphism of any magma A belonging to V is uniquely determined by the images of a generating set for A.

During the execution of a MAGMA program, the runtime system automatically creates and stores most natural homomorphisms that arise. For example, when a submagma is created, the inclusion mapping is stored. Similarly, when a quotient magma is created, the natural epimorphism is stored. When operations are attempted on objects belonging to different magmas related by such natural homomorphisms, the evaluation mechanism will automatically apply these homomorphisms iteratively so as to coerce the operands into a common magma.

3 The Language

The MAGMA language is an imperative programming language with standard imperative-style statements and procedures. In addition, it supports both the set-theoretic [12] and functional programming paradigms. In particular, the language has a functional subset providing functions as first class objects, higher order functions, partial evaluation, etc.

A novel and central aspect of the MAGMA language is the provision of general *constructors*. These are used to define magmas and mappings as well as instances of the basic data types: *set*, *sequence*, and *tuple*. We give the reader a brief introduction to some of these constructors by means of simple examples.

Magmas are typically created by applying a varietal operation to some existing magma. The process begins with the creation of a 'free' magma. Consider the following MAGMA statements where MAGMA input is preceded by a > prompt sign.

```
> Q := RationalField();
> P<x> := PolynomialRing(Q);
> I := ideal< P | x^2+1 >;
> F<i> := quo< P | x^2+1 >;
```

The function `RationalField` returns the rational field $\mathbf{Q}$ (a magma). The function `PolynomialRing` creates a *free* magma, the 'free' (commutative) ring in one indeterminate over $\mathbf{Q}$. The `<x>` construction assigns the generator of the ring P to the variable x. In addition, x will be used as the name of the indeterminate when elements of P are printed. The third statement uses the sub-constructor to create the ideal of P generated by $x^2 + 1$. Finally, using the quo-constructor, the field $F = \mathbf{Q}(\sqrt{-1})$ is constructed as the quotient of P by the ideal generated by $x^2 + 1$.

Homomorphisms are defined either by giving the image of a general element of the domain magma or by specifying the images for the defining generators of the domain magma.

```
> P<x, y> := PolynomialRing(Integers(), 2);
> K := FiniteField(101);
> T<u> := PolynomialRing(K);
> tau := hom<P -> T | x -> K ! 23, y -> u>;
> print tau(x^4*y^5 - 24*x^2*y + x^2 - y + 7);
71*u^5 + 29*u + 31
```

We create a polynomial ring P in two indeterminates x and y over the ring of integers $\mathbf{Z}$. We next create the finite field $\mathbf{F}_{101}$ and a polynomial ring T in one indeterminate u over K. The mapping ρ is the natural embedding of $\mathbf{Z}$ in K. Finally, we define a homomorphism τ from P to T which evaluates x at 23. The notation `K!23` indicates that the integer is to be coerced into the corresponding finite field element.

A MAGMA set consists of a collection of elements belonging to a single magma. A similar convention holds for sequences. A tuple is an element of a general Cartesian product. A set is specified either by explicitly listing its elements or by using a predicate to define the set as a subset of a larger set. Sets and sequences are defined by similar syntax and are distinguished through the use of different delimiters: braces for sets and square brackets for sequences. The following statement creates the set of primes less than or equal to 100:

```
> P := { x : x in [1..100] | IsPrime(x) };
```

The constructor uses the Boolean-valued intrinsic function `IsPrime` to select the subset of the integers in the range $[1, 100]$ that are prime. Note the use of a special construction for arithmetic progressions to designate the set of integers lying between 1 and 100. The notation mimics the mathematical notation $\{x : 1 \leq x \leq 100 \mid x \text{ is prime}\}$.

The next example creates the set of integers up to 100 that are sum of two squares; for this we use the *existential quantifier* predicate to answer the question: $\exists(y, z) : 1 \leq y, z \leq 10$ such that $x = y^2 + z^2$?

```
> Q := { x : x in [1..100] | exists{ <y, z> :
          y, z in [1..10] | x eq y^2+z^2 } };
```

Note that a *tuple* `<x, y>` appears in the existential quantifier.

We conclude this section by giving a MAGMA language implementation of the modular algorithm for computing the greatest common divisor (GCD) of two monic polynomials with integer coefficients. A description of the algorithm may be found in [8]. The GCD function, `ModGcd`, calls the function `Chinese` to perform

the Chinese Remainder Algorithm on the polynomials $a \bmod m$ and $b \bmod p$. The function `Chinese`, in turn, calls the Magma intrinsic `Solution(u, v, m)`, where u, v and m are pairs of integers, to solve the congruences $u_1 x = v_1 \bmod m_1$, $u_2 x = v_2 \bmod m_2$, where m_1 and m_2 are coprime. The function `Parent`, applied to any object returns its parent magma.

```
Chinese := function(a, m, b, p)
   C := func<p, i | i gt Degree(p)
            select 0 else Coefficient(p, i)>;
   M := Max(Degree(a), Degree(b));
   X := [ Solution([1, 1], [C(a, i), C(b, i)],
            [m, p]) : i in [0..M] ];
   return Parent(a) ! [ x gt (p*m) div 2
            select x-p*m else x : x in X ];
end function;
```

For each successive prime, the function `ModGcd` creates $\mathbf{F}_p[u]$ together with the natural homomorphisms $\phi : \mathbf{Z}[x] \to \mathbf{F}_p[u]$ and $\rho : \mathbf{F}_p[u] \to \mathbf{Z}[x]$. The GCD of the images of polynomials f and g in $\mathbf{F}_p[u]$ is found by calling the intrinsic function `Gcd`.

```
ModGcd := function(f, g)
   R<x> := Parent(f);
   p := 2;
   d := R ! 1;
   m := 1;
   repeat
      p := NextPrime(p);
      S<u> := PolynomialRing(FiniteField(p));
      phi  := hom< R -> S | x -> u >;
      rho  := hom< S -> R | u -> x >;
      e := Gcd( phi(f), phi(g));
      if Degree(e) lt Degree(d) then
         d := Chinese(R!1, 1, rho(e), p);
         m := p;
      else
         d := Chinese(d, m, rho(e), p);
         m *:= p;
      end if;
   until (f mod d eq 0) and (g mod d eq 0);
   return d;
end function;
```

We apply our program to polynomials f and g:

```
> R<x> := PolynomialRing(Integers());
> f := (x^3 - 1)*(x^2 - 4*x + 4)^3*(x^2 + 1);
> g := (x^2 + x + 1)*(x - 2)^3*(x - 7);
> print ModGcd(f, g);
x^5 - 5*x^4 + 7*x^3 - 2*x^2 + 4*x - 8
```

A full description of the language may be found in [1] and [5].

4 Implementation Issues

4.1 Signatures and Coercion

While magmas and maps have an obvious expressive and organizational power, they are also very useful mechanisms internally in MAGMA. The use of parent structures in combination with the availability of relationships enables us to rapidly answer questions such as: Is it legal to apply operation f to objects x and y, and, if so, where is the algorithm for f?

The machinery to answer such questions is provided by MAGMA's signature matching mechanism which is a generic method for locating a function with a given name, and having a given argument specification. For those readers familiar with object oriented programming, MAGMA's signature mechanism is akin to multi-method dispatch in CLOS [10]. For example, the multiplication operator will have a signature entry indicating that two finite field elements are proper arguments for the operation. But this is only part of the story: in general, we do not want to multiply elements from different finite fields. So first we check equality of the parents of x and y. Even this is not the full story: we *do* want to permit multiplication of two elements belonging to subfields of a common finite field. In such a case, MAGMA uses its relationship machinery to determine whether there exists a common over-magma N for the parents of x and y. If so, x and y are each lifted into N using the transition maps stored with the relations, and the operation is performed in N.

Note that while this may seem complex, it is in fact just a series of simple decisions. It is also efficient since a match is usually found on the first attempt and the overhead of lifting using the relationship machinery is avoided. Observe also that the two problems of type checking (including type casting) and method selection are both handled by the one general mechanism, a mechanism which starts from the parent magma, and which uses remembered maps as required. Here then we see the practical benefits of using the magma and the map as the central objects in MAGMA.

4.2 A Software Architecture

An important objective has been the development of a kernel design which allows the integration of C programs written independently of the Magma project into the system kernel. To this end we devised a software system architecture, later dubbed the 'the software bus', which isolated the language part of MAGMA from the part that implements the mathematical algorithms. This isolation takes several forms:

1. The memory management software provides both pointer and handle based memory allocation.

Thus, code written using the C library `malloc` and `free` interface can be added to MAGMA as MAGMA's memory manager provides equivalent services.

2. The relationship machinery provides a set of generic services for registering a relationship between two objects, for tracing relationship chains, for accessing a relationship and so on.

3. The run-time system assumes as little as possible about the objects it manipulates. All interaction with the mathematical part of the system is through six system functions and a set of mathematical functions. The system functions define low level facilities such as print, copy and delete. They are obligatory for any code module that implements a magma. A set of fundamental mathematical functions implement such tasks as the creation of a free magma and evaluation of the standard constructors. All modules added through this mechanism have the same status as code written especially for the system, and, in particular, they are accessible both to code written in the MAGMA language, and to kernel C code. Most importantly, all kernel modules adhere to the interface protocols. The latter point is critical since when an implementor installs a new ring type by defining its interface, all recursively defined rings, such as matrix and polynomial rings, will work immediately if defined over a member of the new ring type.

5 Current Status

With the exception of user defined categories, the complete language has been implemented. The kernel contains very efficient machinery for groups, rings, fields, modules, graphs and linear codes. MAGMA V1 was released for general distribution in December 1993 and by April 1994 was installed at 100 sites. In 1994, mechanisms will be implemented to enable users to create their own categories of magmas in the MAGMA language.

6 Acknowledgements

A preliminary design for MAGMA (then known as Cayley V4) was developed by John Brownie, Greg Butler, John Cannon, Robin Deed and Jim Richardson in 1987. Jim Richardson implemented a prototype parser for the language in 1987, while Robin Deed implemented a small prototype of the run-time system in 1988. The final design and implementation was carried out with the participation of Mark Bofinger, John Brownie, Steve Collins, Bruce Cox, Andrew Solomon, and Allan Steel, in addition to the authors. The development of MAGMA was funded in part by the Australian Research Council.

References

[1] Bosma W. and Cannon J.J., *Handbook of Magma Functions*, First Edition, 1993, 690 pages.

[2] Butler G. and Cannon J.J., *Cayley version 4: The user language*, in: P. Gianni (ed), *Proceedings of the 1988 International Symposium on Symbolic and Algebraic Computation*, Rome, July 4–8, 1988, LNCS **358**, Springer, New York, 1989, 456–466.

[3] Butler G. and Cannon J.J., *The design of Cayley, a language for modern algebra*, in: A. Miola (ed), *Design and Implementation of Symbolic Computation Systems*, LNCS **429**, 1990, 10–19.

[4] Cannon J.J., *An introduction to the group theory language Cayley*, in: M.D. Atkinson (ed), *Computational Group Theory*, Academic Press, London, 1984, 145–183.

[5] Cannon J.J. and Playoust C.A., *An Introduction to Magma*, First Edition, 1993, 240 pages.

[6] Char B.W. *et al*, *Maple V Language Reference Manual*, Springer-Verlag, New York, 1991.

[7] G. Cousineau, *The categorical abstract machine*, in *Logical Foundations of Functional Programming*, Addison-Wesley, 1990.

[8] J.H. Davenport, Y. Siret and E. Tournier, *Computer Algebra*, Academic Press, London, 1988.

[9] Jenks R.D. and Sutor R.S. *AXIOM - The Scientific Computation System*, Springer-Verlag, New York, 1992.

[10] Paepcke A., *Object-Oriented Programming: The CLOS Perspective*, MIT Press, 1993.

[11] Santas. P.S., *A type system for computer algebra*, in: Alfonso Miola (ed), *Design and Implementation of Symbolic Computation Systems*. LNCS **722**, Springer-Verlag, Berlin, 1993, 177–191.

[12] Schwartz J.T., Dewar R.B.K., Dubinsky E., and Schonberg E. *Programming with Sets – An Introduction to SETL*, Springer-Verlag, New York, 1986.

[13] Wolfram S., *Mathematica - A System for Doing Mathematics by Computer*, Addison-Wesley Publishing Company, Second edition, 1991.