The GAP 4 Type System
Organising Algebraic Algorithms

Thomas Breuer and Stephen Linton

Lehrstuhl D für Mathematik, RWTH Aachen, Templergraben 64, Aachen, Germany

Division of Computer Science, University of St Andrews, North Haugh, St Andrews, Fife, KY16 9SS, UK

`sam@math.rwth-aachen.de`
`sal@dcs.st-and.ac.uk`

Abstract

Version 4 of the GAP (Groups, Algorithms, Programming) system for computational discrete mathematics has a number of novel features. In this paper, we describe the *type system*, and the way in which it is used for *method selection*. This system is central to the organisation of the library which is the main part of the GAP system. Unlike simpler object-oriented systems, GAP allows method selection based on the types of all arguments and on certain aspects of the relationship between the arguments. In addition, the type of an object can change, in a controlled way, during its life. This reflects information about the object which has been computed and stored. Individual methods can be written and installed independently. Furthermore most checking of the arguments is done in a uniform way by the method selection system, making individual methods simpler and less prone to error. The methods are combined automatically to produce a powerful and usable system for interactive use or programming.

1 Introduction

1.1 Acknowledgement

The ideas described in this paper, and their implementation, are joint work of the authors, Frank Celler and, especially, Martin Schönert. Frank and Martin have since left academia.

1.2 The GAP System

GAP (Groups, Algorithms, Programming) [6] is a system for computational discrete mathematics. It was developed from 1986 until 1997 at Lehrstuhl D für Mathematik, RWTH Aachen, under the leadership of Prof. Joachim Neubüser. The system includes the work of many people: students and staff at Lehrstuhl D, visitors who traveled to Aachen wholly or partly to work on the system and users who extended it for their own purposes then made their extensions and enhancements available as share packages for others to use.

Following Prof. Neubüser's retirement in 1997, coordination of the development has been taken over by the School of Mathematical and Computational Sciences at the University of St Andrews, although much work continues to be done in Aachen and elsewhere and development is very much seen as an international cooperation.

The system was originally specialized in group theory, but developments in group theoretic algorithms required calculation with other mathematical objects, such as vectors, algebras and polynomials. Users interested in other areas of algebra and discrete mathematics, such as graph theory, coding theory and semigroup theory, also found the system useful, and so GAP is now described and developed as a system for computational discrete mathematics.

The GAP system consists of the following parts:

- The *kernel*, written in C and containing:
  - the interpreter and compiler for the GAP language;
  - the user and system interfaces;
  - memory management
  - basic functions for such built-in objects as integers of any size, (exact) rational numbers, permutations, finite field elements, elements of free groups (words), lists and records;
  - certain critical routines implemented in the kernel for speed
- The *library*, written in GAP, and containing most of the algebraic algorithms
- *Databases* of groups, character tables and tables of marks, accessed via functions in the library
- *Share Packages* which are refereed user contributed extensions
- *Documentation* which can be accessed using LaTeX, as HTML or from GAP, as on-line help

The system is free software, distributed subject to a "copyleft" which allows redistribution, but protects our rights. It can be obtained via the World Wide Web [5].

2 Method Selection – Issues and Alternatives

2.1 The Problem

Evaluating a GAP expression such as `x * y`, or `DerivedSubgroup( g )` can trigger the execution of many different algorithms, depending on the values of the variables `x`, `y` and `g`. This *overloading* of `*` and `DerivedSubgroup` is needed for at least two reasons. Firstly, we want the system to provide natural answers to natural mathematical questions, without requiring the user to know about the computational aspects of group theory. Consequently, we try to allow the programming language to follow mathematical terminology, in which symbols and words are heavily overloaded. Secondly, we wish to support *polymorphic* programming, whereby the same subroutine might be applicable to many different kinds of object, relying on overloading of more elementary functions. A simple example is the standard algorithm for powering monoid elements by repeated squaring. The same implementation of this algorithm can be applied to elements of any monoid, provided that, in each case, the `*` operator implements the monoid operation.

A method selection system must provide a way of organizing all the functions (called *methods*) which might be used to multiply two objects, or compute a derived subgroup (in general to carry out some *operation*) and selecting a suitable one each time the operation is invoked.

Furthermore, extensibility is a major design goal of GAP, and so the method selection system must be capable of extensions by new methods and operations without modification (or even complete understanding) of the existing code.

2.2 Explicit Method Selection

One possible approach to method selection is to implement, for every operation, a single function (a *dispatcher*) which "knows about" every method for that operation and makes a decision based on the actual arguments.

This approach provides the desired external behaviour – a single entry point for all multiplications, for example, and supports polymorphic programming. The main disadvantage is that it makes the system hard to extend or maintain: a new kind of object requires changes to the dispatchers of all the operations that might be applied to it. In addition, when the conditions of applicability of the methods become complex, subtle errors in the logic of a dispatcher may lead to incorrect or unnecessarily slow methods being used, or to an applicable method not being detected.

There is also a possibly serious performance problem, caused by the testing required to select a method for every operation invocation.

2.3 The Object-Oriented Approach

The most widely used approach to method selection is the "object-oriented" approach (see, for example, [9]). In this approach, the methods applicable to an object are actually or implicitly stored as part of the object. When an operation is invoked, one of the arguments (the first, or one indicated by special syntax) is examined to find the appropriate method.

Typically, objects are gathered into *classes* and the class of an object actually determines the methods applicable to it. Classes are connected in a tree or DAG by "inheritance", and polymorphic methods may be associated with suitable non-leaf nodes of this structure.

Variations of this approach are used in the C++ [12], Java [4], Smalltalk [7] and Eiffel [9] programming languages, in GAP version 3 [11], (hereafter simply GAP 3), in Magma [2] and in Axiom [8] *inter alia.*

2.4 Limitations of the Object-Oriented Approach

Our experience with GAP 3 revealed two main limitations of this approach.

Firstly, operations may have more than one argument, and there may be no natural way of deciding which one should supply the method. This is a very common situation when dealing with algebraic structures. A simple example is the multiplication operation. A user implementing elements of algebraic number fields would naturally wish to be able to multiply them from both sides by rational numbers. One side, however, will inevitably require changes to the multiplication method of the existing rational number class. It is clearly undesirable for an extension to the system to require modification of the existing system in this way. A second problem is that the method selected may have to perform a further level of method selection based on the type of the other argument(s).

The second major limitation of the object-oriented approach is, perhaps, even more peculiar to algebraic computation. In this type of work, we usually begin with a highly implicit description, such as generating permutations or a group presentation, of an object and apply powerful algorithms to uncover, step by step, more and more explicit information about it. We may, for example, discover after substantial computation, that a certain group is abelian. Once this is known, it is possible to compute further using algorithms that are more efficient, but which are only correct for abelian groups. We would like our method selection system to be able to make use of this type of information.

In GAP 3, faced with, for example, a method for abelian groups much more efficient than the corresponding method for general groups, we had a number of choices:

1. Force the system to determine in advance whether or not the argument is abelian and consequently select a method. This may be unacceptably slow, or even impossible.
2. Use the slower general method unless the group was known from its original construction to be abelian.
3. Look inside the data structure representing the group to see whether it is known to be abelian. This introduces undesirable interdependencies between the representation of groups and the method selection.
4. Change the class of the group when it is discovered to be abelian. This was done in some cases, but proved to be a difficult and unsafe operation.

Some other object-oriented systems such as Cecil [3] support limited change of class during the lifetime of an object, and would support option 4. Fixing certain data representations would have made it possible to use option 3 systematically, at some cost in flexibility. However, no solution within the conventional object-oriented paradigm would have allowed a satisfactory level of flexibility in selecting methods for operations of more than one argument. These considerations led us to design a completely new method selection system for GAP 4.

3 Basic Assumptions of GAP 4

3.1 Elements

Two different GAP 4 objects may represent the same mathematical entity for technical reasons, such as being stored in different representations, or having different amounts of stored knowledge, but also for mathematical reasons, since many algebraic structures are defined as quotient sets. Finitely presented (f.p.) groups and algebras, or quotients of groups or vector spaces are perhaps the most common examples. Accordingly, `=` is a definable operator in GAP 4, and it is understood that objects may differ in various ways, but still be equal. Equivalence classes of objects under `=` are referred to as *elements*. In all but a few cases, operations are defined in terms of elements, so that equal arguments will produce equal results.

3.2 Domains

A design decision in GAP is that elements are created first, arithmetic is defined for elements, and then algebraic structures (called *domains*) are constructed from them, usually by giving generating elements. A consequence of this is that, for example, the addition and multiplication of an algebra are necessarily those already defined for its elements. This is occasionally restrictive, but has a number of advantages. One of these is that domains can gain certain knowledge from their generating elements. Every magma[1] consisting of permutations is associative since the multiplication of permutations is known to be associative, and every algebra generated by matrices whose product is the Lie bracket is automatically known to be a Lie algebra. An element can lie in many domains, for example an invertible matrix can lie in a group, an algebra, or a vector space.

3.3 Inheritance of Algebraic Structure

In mathematics each type of algebraic structure implies weaker structures in the sense that a group is itself also a monoid, a monoid is also a semigroup, an F-algebra is also an F-vector space, and so on. In GAP 4, we support transparent access to these weaker structures, so that it is unnecessary to "convert" an algebra into a vector space in order to work with it as such (to construct a basis, say). One issue which arises is with the notion of "generation": we must distinguish group generators and monoid generators, algebra generators and vector space generators. Another possibly inconvenient consequence is that some variations in mathematical notation cannot be supported. For example, the operation `Centre` is defined for any magma in GAP as the subset of elements that commute with all elements in the domain. Since any Lie algebra is mathematically a magma, and we wish algorithms for magmas (which might call `Centre`) to be applicable to Lie algebras, `Centre` cannot return the annihilator of a Lie algebra, although this is the usual meaning of the term "centre" in the theory of Lie algebras.

[1] A magma [1, page 1] is a set equipped with a binary multiplication about which no properties are assumed

4 Method Selection in GAP 4

4.1 Basic Concepts

Hereafter, certain terms, such as "Operation", "Method" and "Object" will be used with an initial capital letter to indicate that they have a specific and technical meaning within GAP 4. Typewriter style will be used for GAP variables, such as `Centre`.

An *Operation* is a mathematically well-defined function taking argument(s) from some domain(s). A *Method* is a GAP function capable of computing the result of an Operation for some, but possibly not all, relevant sets of arguments. An Operation is a GAP function, and when it is called, the method selection system must choose and call an appropriate Method or signal an error if no suitable Method can be found.

Each Method is associated to an Operation and certain conditions on the arguments, including conditions on relationships between the arguments. It is essential that two Methods which may be applicable to the same arguments should produce the same results, or more precisely that they should produce results satisfying the same specification. This ensures that the selection among applicable Methods is purely a heuristic issue, and protects the user from unexpected behaviour.

An Operation is created by a call to `DeclareOperation`. Methods are installed by calls to `InstallMethod`. Each installation specifies an Operation and conditions on the arguments. These conditions can include "non-mathematical" criteria, such as the representation of an object or what is already known about it.

When an Operation is invoked with particular arguments, the most specialised applicable Method, if any, is determined and applied. It may happen that there are two or more applicable Methods, whose specialization is not comparable, for example a Method for nilpotent groups, and a Method for permutation groups, when the argument is known to be a nilpotent permutation group. In this case a heuristic ranking scheme is used, which is outlined in section 4.5.

More details of the method selection system, and practical instructions for its use, can be found in the GAP manual [6].

4.2 Example

The exponent of a group G is, by definition

$$e(G) = \text{l.c.m.}\{ord(g) : g \in G\}$$

where the order of g is $ord(g) = \min\{n > 0 : g^n = 1_G\}$. If $G = \langle g_1, \dots, g_r\rangle$ is abelian, then it is easy to show that $(gh)^k = g^k h^k$, whence

$$e(G) = \text{l.c.m.}\{ord(g_i) : 1 \le i \le r\} \qquad (1)$$

We use this to provide a simple example of the GAP 4 method selection system in action (figure 1). An Operation `MyExponent` is defined, and two Methods installed for it. The first, specialized, Method relies on equation (1) and is therefore applicable to abelian groups with a known generating set. The second is a generic Method applicable to any group.

The Operation is applied to the three groups of orders 7 and 9 from the library of small groups, and the system reports the Methods applied. The group of order 7 is cyclic

```
gap> DeclareOperation("MyExponent", [IsGroup]);;
gap>
gap> InstallMethod(MyExponent,
>      "Abelian method based on generators",
>      true,
>      [HasGeneratorsOfGroup and IsAbelian
>            and IsGroup],
>      0,
>      function(g)
>     return Lcm(List(GeneratorsOfGroup(g),Order));
>    end);
gap>
gap> InstallMethod(MyExponent, "Default method",
>        true, [IsGroup], 0,
>        function(g)
>    local e,x; e := 1;
>    for x in g do e := Lcm(e,Order(x)); od;
>    return e;
> end);
gap>
gap>
gap> gps := [SmallGroup(7,1),SmallGroup(9,1),
>                               SmallGroup(9,2)];
[ Group( [ f1 ], ... ), Group( [ f1, f2 ], ... ),
Group( [ f1, f2 ], ... ) ]
gap> TraceMethods(MyExponent);
gap> List(gps,MyExponent);
#I  MyExponent: Abelian method based on generators
#I  MyExponent: Default method
#I  MyExponent: Default method
[ 7, 9, 3 ]
gap> HasIsAbelian(gps[2]);
false
gap> IsAbelian(gps[2]);
true
gap> MyExponent(gps[2]);;
#I  MyExponent: Abelian method based on generators
```

Figure 1: GAP 4 Method Selection Example

and known to be abelian, and the abelian group Method is chosen. The two groups of order 9 (although in fact abelian) are not known at this stage to be so, and the generic method is used. The `HasIsAbelian` Operation confirms the state of our knowledge, while the call to `IsAbelian` actually forces the system to determine the answer. Once this has been done, the more specialized Method is chosen, since the group is now known to be abelian.

4.3 Method Installation in More Detail

The general format of `DeclareOperation` is

`DeclareOperation(` $\langle \textit{name} \rangle$`,` $\langle \textit{requirements} \rangle$`)`

Here $\langle \textit{name} \rangle$ is simply an arbitrary string label, while $\langle \textit{requirements} \rangle$ defines the full set of possible arguments for which the Operation makes sense. It is normally forbidden to install a Method for the Operation which requires less of its arguments than this.

The general format of `InstallMethod` is

`InstallMethod(` $\langle \textit{operation} \rangle$`, [`$\langle \textit{description} \rangle$`,]`
$\langle \textit{fampred} \rangle$`,` $\langle \textit{requirements} \rangle$`,` $\langle \textit{rank} \rangle$`,` $\langle \textit{method} \rangle$`)`

The optional argument $\langle \textit{description} \rangle$ is a string that describes the Method $\langle \textit{method} \rangle$, the meaning of the $\langle \textit{fampred} \rangle$ and $\langle \textit{rank} \rangle$ arguments will be explained later.

Requirements for a method and an operation are expressed in terms of a *Filter* for each argument.

4.4 Filters

A *Filter* is a Boolean quality of an Object such as:

IsMagmaWithInverses The Object represents a set whose elements can be combined with `*` and which support `^0` and `^-1` providing an identity and inverse.

HasSize The result of the `Size` Operation for the Object is stored (or otherwise very quickly available).

IsListHashTable The Object is a hash table represented in a particular way.

HasIsAbelian The Object knows whether or not it is abelian.

IsAbelian The Object knows that it is abelian.

It is often convenient to identify a Filter with the set of Objects which possess it.

The meet of two Filters may be constructed using `and`. For example, `IsGroup` is constructed as `IsMagmaWithInverses and IsAssociative`. A Filter which is not produced in this way is called a *simple filter*.

An implication may be established between Filters, asserting that any object in one is automatically in the other. For example `IsGroup and IsCyclic` implies `IsAbelian`, or `IsMagmaWithInverses` implies `IsMagma`.

Every Object carries with it (sometimes implicitly) the smallest filter to which it is *known* to belong.

4.5 Ranks of Filters and Methods

A simple filter, when it is created, is given a nonnegative *incremental rank* (nearly always 1) and the *rank* of a filter is simply the sum of the incremental ranks of the simple filters that compose it. For filters closed under implication, if filter F_1 is contained properly in filter F_2 then F_1 will have a higher rank than F_2.

Whenever a Method is installed, a *rank* is computed for it. This is basically the total of the ranks of its requirements filters added to the $\langle rank \rangle$ parameter of the installation (see section 4.3). The idea behind this is that a Method requiring more than another Method is expected to be a more efficient one.

As soon as the requirements of two Methods are incomparable, this ranking becomes no more than a heuristic, and in such situations it is usually highly dependent on the actual arguments which of the Methods is the best. In several cases, a Method is seen to be very efficient not because of the number of its requirements but because of a particular requirement. Then this value of the Method can be expressed by a high $\langle rank \rangle$ parameter. For example, consider the Operation to compute the concatenation of two lists. In the special case that one of the lists is empty one can simply return the other list (if it is immutable, see section 7.1), and this very efficient Method depends only on the requirement `IsEmpty`. So it makes sense to give it a high rank, although there might be Methods installed with more requirements, cf. section 5.2.

4.6 Method Selection

When an Operation is invoked on actual arguments, the Filters associated with those Objects are compared to the requirements of the installed Methods for the Operation, and the applicable Method with the highest rank is invoked. Again it should be emphasized that the aim is to select a "good" Method, not necessarily the "optimal" Method for the given arguments if such a Method exists at all.

4.7 Relations between Objects – Families

Filters are essentially static; the complete set of filters is created when the libraries (including user extensions) are read. They are sufficient to capture the possible qualities of individual objects, but do not allow relationships between objects to be captured.

For example, the generic commutator function might be installed as:

```
InstallMethod(Comm, "generic method", true,
        [IsMultiplicativeElementWithInverse,
         IsMultiplicativeElementWithInverse],
0, function(x,y) return x^-1*y^-1*x*y; end);
```

but is applicable to a permutation and a free group element, whose commutator is meaningless.

We could attempt to add checking to the method itself to deal with such problems, but this would have to be modified whenever a new type of multiplicative element was created. This is exactly the kind of tedious checking and problematic interdependence which we are trying to avoid.

We could install methods only for more specific pairs of argument types such as `[IsPerm, IsPerm]` or `[IsMatrix, IsMatrix]`. This would require repetitive code, and would still not, for instance, forbid attempting to compute the commutator of elements in different f. p. groups or matrices with entries in rings of different characteristics.

Instead, we capture these relationships using the notion of *Family*. Every Object lies in a Family (equal Objects are in the same Family) Families can be created dynamically, for example the elements of each f. p. group form a Family, which is created at the same time as the group itself.

Families play a role in method selection by way of the $\langle fampred \rangle$ argument passed with each method when it was installed. The value of this argument should be a function (or `true` which is treated as the constant function returning `true`), to which, when selecting a Method, the system will pass the Families of the actual arguments of the Operation. Only if this function returns `true` is the Method considered applicable.

Since this requires a GAP function to be called for every potentially applicable Method, it could present performance problems, but common cases such as Family equality are optimized and performance seems acceptable in practice.

4.8 Partial Methods

There are some algorithms whose applicability cannot be exactly described in terms of Filters and Families.

Such algorithms can be implemented as *Partial Methods*. They are installed as normal (in this case, requiring `IsGroup and HasSize`) and either return a value (in this case `true`) or call `TryNextMethod()`, which never returns, but instead causes the system to invoke the next highest ranked applicable Method if there is one. If no applicable Method remains, an error is signalled, just as if no applicable Method had been found in the first place.

A simple example is the trivial algorithm for determining solubility of groups of odd order (by returning `true`). Using partial Methods, this mathematical theorem can be formulated as one simple subroutine, and will be applied whenever it is needed. Without partial methods, every Method for solubility testing would need to take account of this theorem.

4.9 Immediate Methods and Implications

Some Methods are computationally very cheap, and may compute useful information about an Object that will allow faster Methods to be selected in the future. These may be installed as *Immediate Methods* (always taking just one argument) and are invoked *whenever* an Object is discovered to lie in the required Filters. It is important that Immediate Methods be extremely cheap to run, as they may be triggered without a direct user request, something that we usually avoid. For example, factoring the known size of a group for finding out whether it is a prime power is in general *not* extremely cheap, so there is no such Immediate Method for checking nilpotency of a group. Execution of Immediate Methods can be suspended without damaging the system.

A special case are *Implications*, where the Immediate Method is simply to return `true`. For example there is an Implication of `IsNilpotentGroup` triggered whenever an Object is found to have `IsAbelianGroup`. Implications are actually implemented differently, by direct manipulation of Filters without calling the Method as a function, and are very cheap to perform (but cannot be suspended safely).

4.10 Attributes and Properties

An *Attribute* is a unary Operation with no side-effects, for example `Exponent`. The results of such an Operation, applied to an immutable Object (see section 7.1), may safely be stored in the argument Object to be reused, although the memory used means that actually storing the result may not always be the best choice.

Associated with each Attribute is a simple filter, the so-called *attribute tester*, which is conventionally named `Has`⟨*attr*⟩ (e.g. `HasExponent`) but always accessible from ⟨*attr*⟩ by the built-in function `Tester`. The meaning of the tester is that the Attribute value is available without significant computation, typically because it is stored. The tester can then be used in method selection, as we saw above with `HasGeneratorsOfGroup`.

The storing of Attribute values is managed by another related Operation, the *attribute setter*, accessible by the built-in function `Setter` and conventionally given the name `Set`⟨*attr*⟩. When a Method for the Attribute returns a value, the setter Operation is invoked. The most general Method, installed for all setters, simply ignores the value. For many Objects a more specialized Method is installed which stores the value and sets the corresponding Tester. In these cases, a high rank Method is also installed for the Attribute itself, requiring the Tester, which simply recovers the stored value. In a very few cases, alternative Methods are installed, for example to store the Attribute value in a place where many related Objects can share it.

As an example, consider a group `g`, whose size (Attribute `Size`) is initially unknown. It will therefore not lie in the Filter `HasSize` and the call `HasSize(g)` will return `false`. A call of `Size(g)` will trigger method selection and (if possible) computation of the size of the group. Since `Size` is an Attribute, the system will automatically call `Setter(Size)` with the result of the computation, and, for most groups, this will select a Method which stores the size in the group and sets the `HasSize` filter. Now `HasSize(g)` will return `true`. Subsequent calls of `Size(g)` will select the high rank Method that requires `HasSize` which will simply recover the stored value.

Attributes whose values are always either `true` or `false`, such as `IsAbelian`, are called *Properties.* Besides the Property tester, each Property is also a filter in its own right and can be used for Method selection. There are three possible values of the (Property Tester, Property) pair:

Tester false, Property false Nothing is known about whether the Object has the Property

Tester true, Property false The Object is known not to have the Property

Tester true, Property true The Object is known to have the Property

Since only one bit is required (see section 6.1), Property values are stored for reuse in almost all Objects.

Actually invoking the Property as an Operation will cause the system to attempt to determine whether the Object has the Property, but method selection will not trigger this (since it might be expensive or even impossible), it will make use only of existing knowledge. The example of figure 1 shows this behaviour.

4.11 Other Features of Method Selection

4.11.1 Constructors

A few Operations, such as `AbelianGroup`, actually take a Filter as an argument, which is treated specially. The Method selected is one that guarantees to return an Object which lies in the Filter.

Note that the direction of inheritance here is opposite to that for other Operations. For example, if a Constructor is invoked so as to require it to construct an Object in `IsPermGroup`, it is acceptable for it to choose a Method which guarantees to produce an Object in `IsPermGroup and IsTransitive`, but not one that may produce a general Object in `IsGroup`. A general Operation, on the other hand, may invoke a Method that only requires `IsGroup` on an Object in `IsPermGroup`, but *not* one that requires `IsTransitive`.

4.11.2 Methods for Unusual Arguments

The ⟨*requirements*⟩ given when an Operation is created act as a useful check on Method installations for that Operation and avoid a lot of potential errors caused by Method implementors misunderstanding the intended scope or meaning of an Operation. Sometimes, however, it is useful to install Methods for "unusual" argument types. In this case `InstallOtherMethod` can be used. This facility should be used with care, and each use justified in comments or documentation.

Each Operation normally requires a fixed number of arguments, and the Method selection system makes no provision for Operations with variable numbers of arguments, although `InstallOtherMethod` can be used to install a Method for a different fixed number of arguments. For example, for the Operation `GroupByGenerators` there are Methods for one argument, namely a list of generators, and Methods for two arguments, namely a list of generators and the identity of the group. Many situations that might appear to call for a variable number of arguments, such as `Print`, are dealt with by a function which calls an Operation, in this case `PrintObj`, on each argument in turn.

4.11.3 Debugging Facilities

Sometimes one wants to know which Methods of an Operation are actually used during a computation. For that purpose, the function `TraceMethods` can be called with a list of Operations, with the effect that for each of their Methods that is executed, the information string ⟨*description*⟩ (see section 4.3) is printed. figure 1 shows an example.

Access to the Method itself that GAP would choose for an Operation and given arguments is provided by the function `ApplicableMethod`, which can be used also to get the applicable Method of second highest rank etc., or a list of all applicable Methods, ordered by their rank.

5 The Type of an Object in GAP 4

The filter and family of an object together make up its *Type*, which determines the methods that will be applied to it.

All filters are treated equally in method selection. However, with a small number of exceptions, they can be divided into a number of groups which describe different aspects of an Object, which is reflected by the different ways in which they are created and used.

5.1 Categories

The Categories to which an Object belongs determine the Operations that can sensibly be applied to it. They are fixed when an Object is created, and never change. They can be likened to abstract classes in C++.

A typical Category of elements is `IsAdditiveElement`. An Object belongs to `IsAdditiveElement` if it can be added to (at least some) other Objects of its Family using the `+` Operation.

5.2 Representations

The Representations to which an Object belongs determine how it is stored. They are used to select low-level access and arithmetic Methods. Most Methods are completely independent of Representation, relying on Representation dependent Methods for a few basic access Operations.

For example, basic Operations for a list (see also section 7.2) are `Length`, access to an entry, the check whether a position is bound, assignment to a position, and unbinding a position. When a new Representation of lists, such as sparse lists, is introduced, then it is necessary to install Methods for these basic Operations that "understand" the way in which the data is stored. However, no such special Method is needed for the Operation `Add` which assigns an Object to the first position larger than the list length, since there is a generic Method that uses `Length` and the assignment Operation.

Of course it might be useful to install Methods also for some non-basic Operations, e. g., for computing the concatenation of two lists, which can then be constructed as an Object in the new representation of sparse lists. The generic Method to compute the concatenation is of course also applicable, but the new Method will be faster, and have higher rank.

5.3 Attribute Testers and Properties

These have been discussed in section 4.10. Unlike Categories and Representation, which are always fixed when an Object is created, an Object may be put into these filters during its lifetime.

5.4 Kernel Objects

Built-in Objects like permutations have an implicit type, which cannot change. For basic operations, such as `+`, `*`, `PrintObj`, etc., the kernel selects the method directly, avoiding any overhead from the Method selection (at the cost of removing the ability to override the standard behaviour). For other Operations, the normal method selection procedure is followed, using the implicit type.

6 Implementation of Types and Method Selection

Most of the Type and Method selection system and is actually implemented in the GAP language. In particular, Types, Families, Filters, Operations and so on are themselves usual GAP Objects, whose data one can access. Also the Methods of Operations are stored in plain lists, as well as the Immediate Methods and the implications of filters.

Only a few time critical parts get additional kernel support via small functions that are frequently called. This can be described as follows.

6.1 Flag Lists

The Filter part of a Type is stored as a bit list, one bit for each existing simple filter, and these *flag lists* are used throughout the Method selection system instead of their Filters. This implementation admits effective checking whether one Filter is contained in another, which is a frequent requirement when the applicability of Methods is tested.

6.2 Caching and Hashing

A hash table of Types is used to ensure that two equal Types or Families are always represented by pointers to the same memory location. This costs time when they are created but saves time and space later.

Each Operation caches the three most recent (sets of) Types with which it was called and the Method selected. We find that most calls hit this cache, so that a search through the list of all Methods is started only if the Types of the actual arguments do not match an entry of the cache. The cache is cleared whenever a new Method is installed.

The same idea is used to reduce the costs of computing the meet of two flags lists, the closure of a flags list under implications of Filters, and certain more technical calculations.

7 Other GAP 4 Features

7.1 Mutability

Attribute and Property information cannot always be kept for Objects with mutable subobjects, because they may change without notice.

Our philosophy in GAP 4 is to work with immutable objects as much as possible. Immutable Objects can acquire information or be reorganised internally, but the element which they represent is not permitted to change. Thus, a sorted list of immutable Objects may be relied upon to remain sorted. Attribute values are generally stored using immutable Objects, so that pointers to them can be returned without risk that the stored values will be altered. This avoids much unnecessary copying.

7.2 Virtual Lists and Records, Iterators

The list access and assignment operators `\[\]`, `\[\]\:\=` and related functions such as `Length` are Operations. An Object can be in the `IsList` Filter and have Methods installed for these Operations and be a "virtual" list (possibly infinite). In the same way, the `\.` and `\.\:\=` operators are Operations allowing virtual records.

Similarly, the `for` statement will call the `Iterator` Operation to obtain a way of running through the elements of an Object that describes a set, enabling one to write `for el in grp do` to run through the elements of any group to which a Method for `Iterator` is defined.

8 Comparison to Other Type Systems

The GAP 4 type system has a number of features in common with those of most languages that describe themselves as object-oriented, including the command and extension languages of many other symbolic computation systems, and a number of important differences.

In common with most of the other systems, we have *inheritance*, whereby more specialized objects inherit the behaviour of more general ones, and the closely related notion of *inheritance-based polymorphism*, where the same code can be applied to any concrete object which inherits from a suitable abstraction. In the terminology of [13], we use inheritance only for *conceptual specification*, and insist on *strict* or *behaviourally compatible* inheritance. This is sometimes referred to as subtyping, rather than inheritance, and that is compatible with our identification of Filters with the set of Objects lying in them.

As is usual in symbolic computation (Axiom [8] being the notable exception) but less so in general-purpose programming, the system is purely dynamic. Almost no useful information about the type of the result of evaluating an expression can be obtained simply by analysing the expression, and it is almost never possible to predict during parsing or compilation what Method will be selected for a particular Operation invocation. This is, perhaps, appropriate for a style of computation where we are exploring the structure of an implicitly defined object. Static typing could not be expected to deduce whether a subgroup arising at a certain stage of the calculation will be soluble or not, but this fact may lead to completely different choices of representations and methods.

Another very visible difference between our system and most object-oriented languages is that we have (again in the terminology of [13]) *multiple dispatch* or *multimethods*, and, as is usual for languages with this feature, use a procedural syntax, rather than a message-passing syntax. Two other languages with multimethods are Common Lisp (using the Common Lisp Object System CLOS) [15] and Cecil [3]. GAP 4 Operations are very similar to CLOS generic functions.

On the other hand, neither of these languages has anything like GAP 4 Families as part of the method selection scheme. In fact, the effects achieved with them resemble more closely the *parametric polymorphism* found mainly in functional programming languages such as Haskell [10]. In these languages one can construct a function of type `(T: Num) =>( T, T) -> T` where `T` is a type variable which ranges over numeric types. This is conceptually similar to a GAP 4 Method whose ⟨*fampred*⟩ ensures that the two arguments are from the same Family.

The final unusual feature is the use for method selection of information discovered during the life of an Object. This seems natural in algebraic computation, because one is exploring the structure of a complex object, but no general purpose programming language has taken this approach. Cecil has *predicate objects*, in effect classes to which an object is considered to belong if it belongs to the parent(s) and satisfies a run-time predicate. In that case, though, there is no impression of information about an object being discovered. Instead the predicates are viewed as simply testing the current state of mutable objects. Self [14] supports *dynamic inheritance*, in which the parent of an object can be explicitly changed. This could be used simply to record that it now belongs to a subclass, but seems to be envisaged as a way to explicitly change the behaviour of the object to reflect a change in the outside world.

9 Results and Experience

This type system has been used by a number of people to construct and organise the GAP 4 library, which defines over 2000 Operations, 7000 Methods and 1100 Filters and consists of over 200 000 lines of code.

Our experience is that the type system works well as a framework for GAP, and, once learnt, seems more natural and more powerful than the GAP 3 system.

A major advantage is that individual Methods are usually small, relatively independent of one another and free of tedious and error-prone checking of the arguments passed.

While perfect choice of Methods is not achievable, the default behaviour of the system does well in the vast majority of cases.

An additional advantage is that definition and specification (for example calls to `DeclareOperation` and `DeclareCategory` and their documentation) can be separated from implementation (such as calls to `InstallMethod` and `DeclareRepresentation`). By convention, in the GAP library, we use different file extensions, `.gd` and `.gi`, for these purposes.

References

[1] BOURBAKI, N. *Éléments de Mathématique, Algèbre I*, vol. 1. Hermann, Paris, 1970.

[2] CANNON, J., AND PLAYOUST, C. *An Introduction to MAGMA*. School of Mathematics and Statistics, University of Sydney, 1993.

[3] CHAMBERS, C. The Cecil language: Specification and rationale. Tech. Rep. 93-03-05, Department of Computer Science, University of Washington, March 1993.

[4] CORNELL, G. *Core Java 1.1.* Java Series. SunSoft Press, May 1997.

[5] `http://www-gap.dcs.st-and.ac.uk/~gap`.

[6] THE GAP GROUP. *GAP – Groups, Algorithms, and Programming, Version 4*. Lehrstuhl D für Mathematik, RWTH Aachen, Germany and School of Mathematical and Computational Sciences, U. St. Andrews, Scotland, 1997.

[7] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The language and its implementation*. Addison-Wesley Series in Computer Science. Addison-Wesley, 1983.

[8] JENKS, R. D., AND SUTOR, R. T. `axiom` *The Scientific Computation System*. Springer-Verlag, 1992.

[9] MEYER, B. *Object-oriented Software Construction*. International Series in Computer Science. Prentice-Hall, 1988.

[10] PETERSON, J. C., HAMMOND, K., AUGUSTSSON, L., BOUTEL, B., BURTON, F. W., FASEL, J., GORDON, A. D., HUGHES, R. J. M., HUDAK, P., JOHNSSON, T., JONES, M. P., JONES, S. L. P., REID, A., AND WADLER, P. L. Report on the non-strict functional language, Haskell, version 1.4. Tech. rep., University of Yale, 1997.

[11] SCHÖNERT, M., ET AL. *GAP – Groups, Algorithms, and Programming*, sixth ed. Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1997.

[12] STROUSTRUP, B. *The C++ Programming Language*, 2nd ed. Addison-Wesley, 1995.

[13] TAIVALSARI, A. On the notion of inheritance. *ACM Computing Surveys 28*, 3 (September 1996), 438–479.

[14] UNGAR, D., AND SMITH, R. B. SELF: The power of simplicity. In *OOPSLA '87 Conference Proceedings* (1987), pp. 227–241.

[15] WATT, S. N. K. Pride and prejudice: Four decades of Lisp. In *Programming Languages: Practice and Experience*, M. Woodman, Ed. Chapman and Hall, 1995.