THE GAP4 TYPE SYSTEM
ORGANISING ALGEBRAIC ALGORITHMS

Stephen Linton

St. Andrews University

January 9, 1998

About GAP

- GAP is a system for computational discrete algebra.
- Originally specialized in group theory
- Structured like Maple:
  - A C kernel – basic arithmetics, language interpreter/compiler, user and system interface, garbage collector
  - A GAP library – most algorithms
- Also includes large databases and contributed "share packages"
- Free software – Gnu-like "copyleft"
- Programmability/extendibility is a key goal – including programming by people with limited contact

Method Selection – The Problem

- Many different algorithms may be used to "multiply" two algebraic objects, or to compute a property such as "Size" or "Derived Subgroup", depending on what the objects are
- How do we represent all these, and make the correct decisions, while retaining modularity and efficiency?

Some Terminology

Overloading Use of one name to refer to multiple functions (or objects)

- The '+' operation in most programming languages
- In mathematics, the Centre of a group and the Centre of a Lie algebra are defined differently

Polymorphism Use of the same code to compute with different types of data

- a function to reverse a list of objects of any type
- a function to test whether a group (given by generators) is abelian – relies on overloading of '*' (and perhaps '=')

Method Selection 1 – Dodge the Problem

- One option is to give each algorithm a separate name: `IntMult`, `RatMult`, `FPGroupWithRWSMult`, `PermutationMult`, etc.
- Programmer then chooses method
- Prevents polymorphic programming – need a version of "power by repeated squaring" for every type of object
- Also prevents nice syntax – can't write `a := b*c`

Method Selection 2 – Monster Case Statements

- A single function `Multiply` which "knows about" every form of multiplication in the system
- Can easily be invoked by infix '*'
- Allows polymorphic programming
- Makes the system hard to extend/maintain – every new kind of multipliable object required changes to this one procedure (and probably also procedures for '=', '+', etc.
- Possible performance problems

Method Selection 3 – The Object Oriented Approach

- Every object (actually or implicitly) carries with it the functions (methods) that apply to it
- The operations (actually or implicitly) look inside the object for methods
- Gather objects into "classes" – objects in a class have the same methods
- Establish relationships between classes – inheritance
- Polymorphism is fine – abstract classes make it safe(r)
- Overloading is OK if you are careful
- This is done in C++, Java, GAP3, Magma, Axiom, ...

Limitations of Object Orientation 1

GAP3 experience found two main limitations.

1 Operations may have more than one argument

- which object gets to supply the method
- that method has to choose an algorithm based on the other argument(s)
- Risk of infinite recursion

Limitations of Object Orientation 2

2 We may learn or compute new information about an object during its life, which we would like to use effectively

In GAP3, we had a choice between

- Forcing computation of the information
- Ignoring the information, even when it was, in fact, already there
- Messy non-polymorphic *ad hoc* tests for the presence of information
- Messily changing the class of an object when information is discovered

The GAP4 Method Selection Concept

- An *Operation* is a mathematically well-defined function for arguments 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 argument objects
- In GAP4 Methods are associated to Operations, not Objects
- Two Methods for an Operation should agree, where they are both valid – one may be much faster
- An Operation is created once, and then any number of Methods may be installed for it – each installation specifies the domain of validity of the Method
- These domain specifications can include “non-mathematical issues” – representation of an object, what is already known about it, etc.
- When the Operation is invoked with particular arguments, the most specialised applicable Method is determined and applied

Example

```
MyExponent := NewOperation("MyExponent",[IsGroup]);

InstallMethod(MyExponent, "Abelian method based on generators",
        true, [HasGeneratorsOfGroup and IsAbelian and IsGroup], 0,
        function(g)
    return Lcm(List(GeneratorsOfGroup(g),Order));
end);

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);
```

Example Continued

```
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
```

Method Installation in More Detail

- The general format of `NewOperation` is

  `NewOperation(` ⟨`name`⟩`,` ⟨`requirements`⟩`)`

- The general format of `InstallMethod` is

  `InstallMethod(` ⟨`operation`⟩`, [`⟨`description`⟩`,]` ⟨`fampred`⟩`,`
  ⟨`requirements`⟩`,` ⟨`rank`⟩`,` ⟨`method`⟩`)`

- Requirements for a method and an operation are expressed in terms of *filters*

Filters and Flag lists

- 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

- Filters are combined with `and` to produce flag lists – `IsGroup` is constructed as `IsMagmaWithInverses and IsAssociative`.
- Certain (combinations of) filters automatically imply certain other filters. When an object acquires one, the others are filled in.

Method Selection

- Every method has a rank (total number of filters required, plus the *rank* parameter)
- Every object carries a flags list (sometimes implicitly)
- When an operation is invoked on actual arguments, the highest rank method whose requirements they meet is called
- A method can exit with `TryNextMethod()`

Attributes and Properties

- An Attribute is unary Operations with no side-effects (eg `Exponent`)
- It has an associated filter (tester) usually called `HasXxx` (eg `HasSize`) and another operation (setter) usually called `SetXxx`
- Methods for setters may store the attribute value in the object and set the tester
- A high-rank method requiring the tester is installed to recover the stored object
- A Boolean attribute is a Property (eg `IsAbelian`).
- A Property is a filter in its own right

Types of Filter

These distinctions have no effect on method selection, but alter the way filters are created and set

Categories These are fixed when an object is created, and control what basic operations are defined for the object (a bit like abstract classes in C++) (eg `IsAdditiveElement` implies that a + is defined)

Representations Fixed when an object is created, describe how the object is stored, used to select low-level access methods

Attribute testers Set when an attribute value is stored (eg `HasIsAbelian`)

Properties Set when a property is known to hold (eg `IsAssociative`)

Others A small number of special filters (eg `CanEasilyComputePcgs`)

Relations between Objects

- The filters are essentially static – the complete set of filters is created when the libraries are read
- This does not allow relationships between objects to be captured
- For example

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

  might get called with a permutation x and a free group element y
- To handle this in the method would add the kind of tedious checking that we are trying to avoid
- There are infinitely many different kinds of group elements – for example elements of different fp groups

Families

- These relationships are captured by the *family*
- Every object has a family (equal objects are in the same family)
- Families can be created dynamically – for example the elements of each fp group form a family
- The *fampred* argument of `InstallMethod` is passed the families of the arguments. If it returns `true` then the method is applicable
- Potential performance problem, but common cases (like family equality) are optimized
- The flags lists and family of an object together make up its *Type*, which determines the methods that will be applied to it

True Overloading

- Where a name is used for mathematically different meanings (eg Centre of a group $\{x \in G : xg = gx \forall g \in G\}$, Centre of a Lie Algebra $\{x \in L : xl = 0 \forall l \in L\}$
- Have two Attributes: **`CentreOfGroup`** and **`CentreOfLieAlgebra`**
- For the user supply an Operation **`Centre`**, with methods that call the appropriate Attribute
- If there is ever an object which both a group and a LA, there *must* be a method to decide (or object)
- This doesn't happen too often

Other Features 1

Kernel objects Objects like permutations have an implicit type, which cannot change. For basic operations ('+', '*', `PrintObj`, etc.) the kernel selects the method. Other operations see the implicit type.

Constructors A few functions, like `AbelianGroup` take a flags list as first argument. The method selected is one that will return an object with at least those flags (contravariant)

OperationArgs No method selection for functions of variable numbers of arguments – just one method allowed

Often these functions can call an underlying method with fixed number of args – `Print`, `PrintObj`

Other Features 2

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

Our philosophy in GAP4 is to make as much as possible immutable. Immutable objects can acquire information, or even change representation, but their behaviour under '=' never changes

Caching and Hashing A hash table of types is used to ensure that two equal types or families are always identical (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 it was called with and the method selected. Most calls hit this cache.

Also cache the result of `and` for flags lists and a few other things

Other Features 3

Immediate Methods These one-argument methods are run *whenever* an object acquires the required filters. They are used to “fill in” cheap consequences of new information

Virtual Lists and Records The list access and assignment operators `\[\]`, `\[\]\:\=` and related things are Operations. An object can have the IsList filter and methods installed for these operations and be a “virtual” list (possibly infinite).

Iterators In a similar way, the `for` statement will call the `Iterator` operation to obtain a way fo running through an object.

Results and Experience

- Used in current GAP 4 library –
- 2000+ operations, 7000+ methods, 1100+ filters, 200K + lines of code
- Basically seems to work well – more natural than the GAP3 system
- Individual methods are small and (more) independent
- Some problems with "wrong" methods being called, but manageable
- Tedious input checking and case dividing code largely removed
- Setting up a new kind of object is a bit more complicated, but more powerful
- Definition and specification (`NewOperation`, `NewCategory`) separated from implementation (`InstallMethod`, `NewRepresentation`) (we use `.gd` and `.gi` files).

Comparison to Other Type Systems

- Purely dynamic – no useful type information for objects is available before run-time
- Although operations and methods effectively specify their argument types, there is no indication of the return type
- Objects can change type during their lifetimes
- The system can be used well for run-time method selection, the compiler makes almost no use of it.