Conversion to GAP 4

Alexander Hulpke

April 1999

Whoever has written GAP 3 code before will face the problem of converting it to GAP 4. This talk elaborates on a few aspects. Other points are discussed in the chapter "Migrating to GAP 4" in the tutorial.

The process is based on my experience with converting my own code and library code – it is unlikely to cope with *every* piece of legacy code.

Nobody forces you to use all features!

A Rough Outline

Converting a few thousand lines of code looks like a daunting task.

Unless you want to take the opportunity to go over your code again and improve particulars (or you were particularly clever to exploit GAP 3 features) however it is much less work than one might fear.

In almost any cases the "heart" of the routines can stay almost the same, the changes occur where it interfaces with the rest of GAP.

For the code I had to convert, I used the following approach. (The times given are rough estimates for a 1000 line file.)

I) Replacement This is almost an automatic process that

- Changes names and slightly changed syntax (`Info`) of functions.
- Replaces direct attribute accesses (`G.size`) with a proper (`Size(G)`) one.
  If the components are user defined one has to decide whether they should become an attribute.
- Instead of creating epimorphic objects create the homomorphisms and its `Image`.

($\frac{1}{2}$ day)

II) Installations Functions that were components of operations records must be installed as methods. Usually the function bodies can remain almost the same.

Own definitions of operations records become filter declarations.

(2 hours)

III) Trial When running the code a few further problems may show up:

- Changed syntax which was not spotted in the first stage.
- Immutable objects must be copied before changing them.

(1 day)

IV) Improvement The features of GAP 4 might make it possible to improve the code.

- Complicated `if` constructs can be resolved using method selection.constructs can be resolved using method selection.
- Instead of converting to an `AgGroup` it can be sufficient to test solvability.
- In algorithms use `ModuloPcgs` instead of factor groups with a new collector.
- Make composite objects like lists immutable if they are put in sorted lists.

(up to ∞)

If you spot situations where the performance has gone notably down, please tell us!

Changed Names

A few functions or operations have changed their name for various reasons. A list is in the manual.

`Ag` is called `Pc` now, `CharTable` is `CharacterTable`, `Elements` becomes `As`*`Sorted`*`List`

Properties like "solvability" are defined not only for groups. However being solvable as a Lie algebra is something different than being solvable as a group. Therefore properties are defined more specific as: `IsSolvableGroup`, `IsSimpleGroup`.

Changed Syntax

`Order` does not need the group any longer because it has the family. In particular `OrderPerm` and `OrderMat` are obsolete.

Several operations now return `fail` instead of `false` if they cannot perform the task.

Epimorphic Objects vs. Homomorphisms

In GAP 3 functions for different representations always created objects which had the connecting homomorphism hidden in component `.bijection`.

In GAP 4 the library function always returns an isomorphism, the object can be obtained as `Image` of this isomorphism. (Note that the isomorphism is the other way than *obj*`.bijection` in GAP 3.)

Similarly factor groups are created in GAP 4 via `NaturalHomomorphismByNormalSubgroup`.

Records vs. Objects

In GAP 3 more complicated objects were simulated via records. The GAP 4 representation `IsComponentObjectRep` is closest to this.

(If *many* objects are to be created and every object needs to store very little information, the representation `IsPositionalObjectRep` may be advantageous instead.)

Instead of assigning an operations record, the code calls `Objectify` with the right type.

There are still records in GAP 4, however they cannot simulate complex objects any more and thus have lost much of their prominence.

Record components

In GAP 3 essentially every object was a record. This is not the case in GAP 4. Thus access to record components has to be changed.

For component objects, component access can be translated verbatim:

```
x:=a.y;      becomes:   x:=a!.y;
a.z:=x+1;               a!.z:=x+1;
```

However such components are "private". The common interface for stored information about an object now is via attributes.

Fetching stored attributes

```
if IsBound(b.mycomponent) then a:=b.mycomponent;fi;
```

In GAP 4 attributes are always accessed via functions:

```
if HasMycomponent(b) then a:=Mycomponent(b);fi;
```

Note that access is even permissible if the attribute is not yet known, this will cause the value to be computed. Dispatcher functions become obsolete this way.

```
a:=Mycomponent(b);
```


Telling an object about itself

In GAP 4 this can be done via the setter functions of attributes:

```
Setter(Size)(G,123);
SetSize(G,123);
```

Attributes can only be stored for component objects and if the representation includes `IsAttributeStoringRep`

(Do *not* refer to `G!.`*`Attribute`* components, even if they are used at the moment for storing.)

Own Components

Unless you want a component to store private information that is *only used in your algorithm* the cleanest way is to use attributes.

If the one-argument operation `Mycomponent` was installed as a dispatcher in the GAP 3 library, it is likely to be defined already as an attribute in GAP 4.

```
Mycomponent:=NewAttribute("Mycomponent",Filter);
```

If in doubt you can always use `IsObject` for `Filter`, this will only weaken some sanity checks.

Note that attributes are by default immutable. (If the component will change over time, the attribute has to be declared mutable. Even then however it is not possible to *replace* the attribute completely.)

If the component accumulates information of which only part is returned to the user (for example the Sylow subgroup for one prime) the attribute should be declared to be mutable (→ Mutability, below, → Library structure talk).

Explicit Operations Record References

```
U:=GroupOps.Subgroup(G,l);
```

Serves three aims:

Avoid Tests. The standard GAP 3 dispatcher functions test whether the parameter are permissible. GAP 4 does not require such dispatchers. Instead there are `NC` (no check) versions of many operations.

```
U:=SubgroupNC(G,l);
```

Default methods. A function in a "higher level" operations record gets called because the current method cannot do better.

The obvious way in GAP 4 would be `TryNextMethod` or redispatch. However we cannot always rely on an absolute ordering of methods and must be careful to avoid infinite recursions that would happen here:

```
InstallMethod(Op,"special",true,[special],0,
function(a)
  if not Condition(a) then return Op(a);fi;
[...]
```


The quick (but not so clean) solution is to enforce ranking:

```
InstallMethod(Op,"special",true,[filter1],
  10, # <- enforced higher ranking
function(a)
  if not Condition(a) then TryNextMethod();fi;
[...]

InstallMethod(Op,"default",true,[filter2],0,
function(a)
```

A better alternative is to make the default method a global function:

```
DefMethOp := function(a) [...]end;

InstallMethod(Op,"special",true,[filter1],0,
function(a)
  if not Condition(a) then return DefMethOp(a);fi;
[...]
InstallMethod(Op,"default",true,[filter2],0,DefMethOp);
```


Re-Use of constructors.

A typical GAP 3 use is to call a "higher level" constructor and change entries afterwards:

```
U:=GroupOps.Subobject(G,l);
Y.operations:=MyGroupOps;
```

This becomes impossible in GAP 4: Once objects are created one should not change categories or representations. The cleanest way around this is a creator function to which we pass filters as parameters:

```
BuildSubobject := function(obj,filt)
  [...]
  NewType(fam,filter1 and filter2 and filt)
  [...]
end;
```

Then the different methods call the builder function.

```
InstallMethod(Subobject,"default",true,[filter1],0,
function(obj)
  obj:=BuildSubobject(obj,DefaultFilter);
```



```
  return obj;
end);
```

Adding filters will create objects of different type.

```
InstallMethod(Subobject,"mine",true,[filter2],0,
function(obj)
  obj:=BuildSubobject(obj,DefaultFilter and MyFilter);
  return obj;
end);
```


Operations Records

Typical GAP 3 code to create own objects creates an own operations record looks like this:

```
MyOps:=ShallowCopy(ParentDomainOps);
MyOps.name:="MyOps";
```

In GAP 4 we use can use a category to distinguish the new objects.

```
IsMyCategory:=NewCategory("IsMyCategory",ParentCategory);
```

The objects also need a representation. This example assumes that we will use a record-like representation and permit storage of attributes (this is closest to how it was in GAP 3:

```
IsMyRep:=NewRepresentation("IsMyRep",
       IsComponentObjectRep and IsAttributeStoringRep
       and IsMyRep,["parameter"]);
```

If you do not want to *dispatch* on different properties of your objects you do not need to declare more.

Do I need a category or a representation?

If the new objects do not introduce new concepts it can be sufficient to introduce only a new representation and use existing categories.

A `category` describes what you might do conceptually to a mathematical object.

A `representation` describes how the object is implemented in the system.

This distinction is only conceptual, the implementation is almost the same. A user who does not program herself will probably never meet representations.

Family

Finally we need a family for our objects. We can use different families (say for elements in different characteristics) but the simplest way is to create only one family and one type. (There is no rule that forces you to put new objects in different families – You just might find it helpful to use the family as an distinction tool):

```
MyObjectsFamily:=NewFamily("MyObjectsFamily",IsMyCategory);
MyObjectsType:=NewType(MyObjectsFamily,IsMyRep);
```


Creating Objects

The typical GAP 3 code would look like:

```
CreateObject:=function(parameter)
  r:=rec(parameter:=parameter,operations:=MyOps);
end;
```

In GAP 4 this translates to:

```
  r:=Objectify(MyObjectsType,rec(parameter:=parameter));
```

Access to the component `parameter` is via the `!.` operator.

(Instead of using a component `obj!.parameter` we could have used an attribute as well.)

The GAP 3 code now assigns components to the operations record to indicate methods:

```
MyOps.\+:=function(a,b)
  return CreateObject(a.parameter+b.parameter);
end;
```

The corresponding GAP 4 code simply installs the methods:

```
InstallMethod(\+,"my objects",true,
```

```
  [IsMyCategory,IsMyCategory],0,function(a,b)
  return CreateObject(a!.parameter+b!.parameter);
end;
```

Here we dispatch on the category. If we had defined several representations we probably would have dispatched on them as well.

Domains

In GAP 3 some objects had to be domains to be accepted by some functions. This does not hold any longer in GAP 4. The duties of a domain as a common superobject are now dealt with by the families.

In GAP 4 any collection of objects of one family F (a list, a group, ...) will have the family `CollectionsFamily(`F`)`.

(So when creating own objects that represent a collection this is the family they must have.)

It can be convenient to declare a category for collections so that we can dispatch on it.

```
IsMyCatColl:=CategoryCollections(IsMyCategory);
```

GAP will set this category automatically for every collection whose objects are in category `IsMyCategory`. (Provided the family enforces `IsMyCategory`.)

Mutability

In GAP 3 every object was mutable. In GAP 4 a few objects are immutable by default.

a) Attribute Values.
b) Objects. (Immutability means that an object cannot change its identity with respect to "=". However it can acquire further information.)
c) Products of Immutable Matrices/Vectors (in particular products of group generators).

Immutability of type a) usually should not pose any problems as clean GAP 3 code would not try to modify attributes – if it does dangerous inconsistencies may arise. Before changing such objects always a `ShallowCopy` should be taken.

The only exception is attributes that collect information (say the computed Sylow subgroups). Such attributes should be declared as mutable:

```
NewAttribute("ComputedSylowSubgroups",IsGroup,"mutable");
```

Immutability of type b) will be a problem only with hacks that modify existing objects.

In case c) the code will have to be changed slightly. A matrix/vector that arises from multiplication with group generators and is to be changed afterwards has to be copied. (Most frequently this is the identity element, `OneOp` should be used instead of `One`).

Copying

`ShallowCopy` in is an operation. This permits to install more suitable methods than the "top-level-only" copying which always happened in GAP 3.

In GAP 3 there also was a function `Copy` that copied a whole object recursively.

This often copied more then was necessary (losing memory and performance) and could even yield strange results (copying `Cyclotomics` gave a second, different, field of cyclotomic numbers).

Therefore the function has been renamed to `StructuralCopy` in GAP 4.

Note that `StructuralCopy` will *not* duplicate immutable subobjects but return pointers to the same subobject.

Usually a call to `Copy` can be replaced by a call to `StructuralCopy` or (preferably) even `ShallowCopy`.

The main problem is with matrices (which are indistinguishable from lists of vectors). `ShallowCopy` will only copy the topmost level, to get a full copy one has to call `List(mat,ShallowCopy);`.

Finitely Presented Groups

In GAP 3 a finitely presented group was a free group which was told relations.

In particular the elements of the f.p. group were elements of the free group.

This is not only mathematically wrong but also made it impossible to compare elements of an f.p. group.

Consequentially many generic algorithms failed.

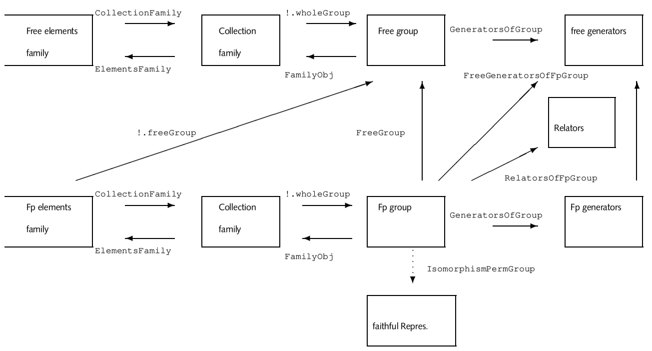
In GAP 4 every finitely presented group has its own family.

For elements of the f.p. group, `UnderlyingElement` gives the representing word in the free group, vice versa `ElementOfFpGroup` wraps a word.

For the full f.p. group (which is stored in the component *`fam`*`!.wholeGroup` of the elements family) the attributes `FreeGeneratorsOfFpGroup` and `RelatorsOfFpGroup` return elements of the free group.

To compare elements, the family will compute a faithful (permutation) representation.

The following picture sums up the situation:



Dark Alleys

There are a few ways to tweak GAP 3 which are impossible to translate directly to GAP 4.

Removal of Attribute components

In general GAP code should *never* remove components that are set by other functions as this may make an object inconsistent.

Attributes in GAP 4 can never be removed as the `Has`*`Attribute`* filter already might have implied other filters.

There are essentially two ways to still get rid of unwanted (large) attributes in GAP 4:

1. Create a new, equal, object X, compute the attribute value for X and trash X afterwards. (Alternatively, compute the attribute for the old object, trash it and keep X.)
   To make this efficient it might be necessary to transfer some other attribute values (in particular `Size`) from the old object to X
2. The function `AttributeValueNotSet` will call the method for an attribute (if it is known, this is the system getter) but not call the setter afterwards to store the result.
   Caveat: The method may dispatch to other, similar operations. For example `AsList` may call `AsSSortedList` if the algorithm will always produce a sorted result. In this

case calling `AttributeValueNotSet` for `AsList` would still store the `AsSSortedList` Attribute.
(It would be nice to make such a feature an option, then it could be inherited by "subattributes".)

Reassignment to Operations Record Components

In GAP 3 an assignment `MyOps.Name:=func;` replaced the existing method. In GAP 4 one can use `Install(Other)Method` to install a new method (when the flags have the same value, the method installed most recently has priority).

Be careful however if the method might call `TryNextMethod()`.

Converting Documentation

The documentation is now based on TeX instead of LaTeX. Unless the manual used many LaTeX tricks, conversion is mainly mechanic.

A rudimentary translation script can be found under `etc/transl.sed`