Objects, types and method selection in GAP

Max Neunhöffer

University of St Andrews

1.8.2013

The idea

GAP objects represent mathematical objects.

The idea

GAP objects represent mathematical objects.

There are “operations” and “methods”.

The idea

GAP objects represent mathematical objects.

There are “operations” and “methods”.

Properties of objects (their type)

$\Downarrow$

Selection of the “right” method

The idea

GAP objects represent mathematical objects.

There are “operations” and “methods”.

Properties of objects (their type)

$\Downarrow$

Selection of the “right” method

Objects can “learn” during their lifetime

(i.e. change their type)

The idea

GAP objects represent mathematical objects.

There are “operations” and “methods”.

Properties of objects (their type)

$\Downarrow$

Selection of the “right” method

Objects can “learn” during their lifetime

(i.e. change their type)

The methods used change as a consequence!

The idea

GAP objects represent mathematical objects.

There are “operations” and “methods”.

Properties of objects (their type)

$\Downarrow$

Selection of the “right” method

Objects can “learn” during their lifetime
(i.e. change their type)

The methods used change as a consequence!

GAP thus uses:

- dynamic typing at runtime

The idea

GAP objects represent mathematical objects.

There are “operations” and “methods”.

Properties of objects (their type)

$\Downarrow$

Selection of the “right” method

Objects can “learn” during their lifetime
(i.e. change their type)

The methods used change as a consequence!

GAP thus uses:

- dynamic typing at runtime
- a static database of methods

The idea

GAP objects represent mathematical objects.

There are “operations” and “methods”.

Properties of objects (their type)

$\Downarrow$

Selection of the “right” method

Objects can “learn” during their lifetime
(i.e. change their type)

The methods used change as a consequence!

GAP thus uses:

- dynamic typing at runtime
- a static database of methods
- “just in time” method selection

Types

A type in GAP is a pair:

(a “family”, a bit list of “elementary filters”)

Types

A type in GAP is a pair:

(a “family”, a bit list of “elementary filters”)

The families form a partition of the set of objects.

one part is for example the `PermutationsFamily`

Types

A type in GAP is a pair:

(a “family”, a bit list of “elementary filters”)

The families form a partition of the set of objects.
one part is for example the `PermutationsFamily`

An elementary filter is both

- a bit in the 2nd component of the type and
- the set of all objects, which have that bit set in their type.

Types

A type in GAP is a pair:

(a “family”, a bit list of “elementary filters”)

The families form a partition of the set of objects.
one part is for example the `PermutationsFamily`

An elementary filter is both

- a bit in the 2nd component of the type and
- the set of all objects, which have that bit set in their type.

A filter is either

- an elementary filter or
- an and-composition of elementary filters.

Types

A type in GAP is a pair:

(a “family”, a bit list of “elementary filters”)

The families form a partition of the set of objects.
one part is for example the `PermutationsFamily`

An elementary filter is both

- a bit in the 2nd component of the type and
- the set of all objects, which have that bit set in their type.

A filter is either

- an elementary filter or
- an and-composition of elementary filters.

Every object o “is” either “in some given filter” or not.
This can be tested with `FILTERNAME(o)`.

Types

A type in GAP is a pair:

(a “family”, a bit list of “elementary filters”)

The families form a partition of the set of objects.
one part is for example the `PermutationsFamily`

An elementary filter is both

- a bit in the 2nd component of the type and
- the set of all objects, which have that bit set in their type.

A filter is either

- an elementary filter or
- an and-composition of elementary filters.

Every object o “is” either “in some given filter” or not.
This can be tested with `FILTERNAME(o)`.

Examples: `IsSolvable`, `IsNilpotent`, `IsAbelian`

Operations and methods

An operation is a collection of methods.
One declares

- the name,
- the number of arguments, and
- a filter for each argument.

Operations and methods

An operation is a collection of methods.
One declares

- the name,
- the number of arguments, and
- a filter for each argument.

```
DeclareOperation("Size",[IsGroup]);
```

Operations and methods

An operation is a collection of methods.
One declares

- the name,
- the number of arguments, and
- a filter for each argument.

```
DeclareOperation("Size",[IsGroup]);
```

One installs one or more methods:

- These are functions with the right number of arguments.
- One can give further restrictions:

Operations and methods

An operation is a collection of methods.
One declares

- the name,
- the number of arguments, and
- a filter for each argument.

```
DeclareOperation("Size",[IsGroup]);
```

One installs one or more methods:

- These are functions with the right number of arguments.
- One can give further restrictions:

```
InstallMethod(Size,
  [IsGroup and IsPermGroup],
  function(p) ... return ...; end);
```

Operations and methods

An operation is a collection of methods.
One declares

- the name,
- the number of arguments, and
- a filter for each argument.

```
DeclareOperation("Size",[IsGroup]);
```

One installs one or more methods:

- These are functions with the right number of arguments.
- One can give further restrictions:

```
InstallMethod(Size,
  [IsGroup and IsPermGroup],
  function(p) ... return ...; end);
```

We call these restrictions “required filters”.

The method selection

If somebody calls `Size(g)` for an object `g`,

The method selection

If somebody calls `Size(g)` for an object `g`,

- GAP determines the type of `g`,

The method selection

If somebody calls `Size(g)` for an object `g`,

- GAP determines the type of `g`,
- considers all methods for `Size`,

The method selection

If somebody calls `Size(g)` for an object `g`,

- GAP determines the type of `g`,
- considers all methods for `Size`,
- determines, which are applicable (are in all required filters),

The method selection

If somebody calls `Size(g)` for an object `g`,

- GAP determines the type of `g`,
- considers all methods for `Size`,
- determines, which are applicable (are in all required filters),
- and calls the method that
    - is applicable, and
    - has the most required filters
      (if two or more have the same required filters it takes the one which was installed later).

The method selection

If somebody calls `Size(g)` for an object `g`,

- GAP determines the type of `g`,
- considers all methods for `Size`,
- determines, which are applicable (are in all required filters),
- and calls the method that
  - is applicable, and
  - has the most required filters
    (if two or more have the same required filters it takes the one which was installed later).

This only works efficiently by a very tricky method cache!

The method selection

If somebody calls `Size(g)` for an object `g`,

- GAP determines the type of `g`,
- considers all methods for `Size`,
- determines, which are applicable (are in all required filters),
- and calls the method that
  - is applicable, and
  - has the most required filters
    (if two or more have the same required filters it takes the one which was installed later).

This only works efficiently by a very tricky method cache!

More accurately: Each elementary filter has a “rank”.
The method with the highest sum of ranks of the required filters is chosen.

The idea behind families

The families partition the set of all objects.

In contrast, the filters form a hierarchy of sets.

The idea behind families

The families partition the set of all objects.

In contrast, the filters form a hierarchy of sets.

e.g.: `PermutationsFamily`, `CyclotomicsFamily`.

The idea behind families

The families partition the set of all objects.

In contrast, the filters form a hierarchy of sets.

e.g.: `PermutationsFamily`, `CyclotomicsFamily`.

For FP groups all elements of one such group form a family.

The idea behind families

The families partition the set of all objects.

In contrast, the filters form a hierarchy of sets.

e.g.: `PermutationsFamily`, `CyclotomicsFamily`.

For FP groups all elements of one such group form a family.

A collection consists of objects from the same family.

The idea behind families

The families partition the set of all objects.

In contrast, the filters form a hierarchy of sets.

e.g.: `PermutationsFamily`, `CyclotomicsFamily`.

For FP groups all elements of one such group form a family.

A collection consists of objects from the same family.

One can form the “CollectionsFamily” of any family,

The idea behind families

The families partition the set of all objects.

In contrast, the filters form a hierarchy of sets.

e.g.: `PermutationsFamily`, `CyclotomicsFamily`.

For FP groups all elements of one such group form a family.

A collection consists of objects from the same family.

One can form the "CollectionsFamily" of any family,

and the "ElementsFamily" of each CollectionsFamily:

The idea behind families

The families partition the set of all objects.

In contrast, the filters form a hierarchy of sets.

e.g.: `PermutationsFamily`, `CyclotomicsFamily`.

For FP groups all elements of one such group form a family.

A collection consists of objects from the same family.

One can form the “CollectionsFamily” of any family,

and the “ElementsFamily” of each CollectionsFamily:

```
gap> f:=CollectionsFamily(CyclotomicsFamily);;
gap> CyclotomicsFamily=ElementsFamily(f);
true
gap> FamilyObj((1,2,3))=PermutationsFamily;
true
```

Categories and representations

“Categories” and “representations” are nothing but elementary filters with a bit of philosophy in the background!

Categories and representations

“Categories” and “representations” are nothing but elementary filters with a bit of philosophy in the background!

Objects in the same category are (mathematically) similar objects. Objects never change category!

Categories and representations

“Categories” and “representations” are nothing but elementary filters with a bit of philosophy in the background!

Objects in the same category are (mathematically) similar objects. Objects never change category!

Mathematically similar or equal objects can be represented differently, then they should lie in different representations.

Categories and representations

“Categories” and “representations” are nothing but elementary filters with a bit of philosophy in the background!

Objects in the same category are (mathematically) similar objects. Objects never change category!

Mathematically similar or equal objects can be represented differently, then they should lie in different representations.

`IsPerm` is a category.

`IsPerm2Rep` and `IsPerm4Rep` are representations.

Categories and representations

“Categories” and “representations” are nothing but elementary filters with a bit of philosophy in the background!

Objects in the same category are (mathematically) similar objects. Objects never change category!

Mathematically similar or equal objects can be represented differently, then they should lie in different representations.

`IsPerm` is a category.

`IsPerm2Rep` and `IsPerm4Rep` are representations.

Categories usually occur in declarations of operations, representations usually occur as required filters in method installations.

Inheritance in GAP

Inheritance works with subfilters.

Inheritance in GAP

Inheritance works with subfilters.

One declares subfilters and constructs objects that lie in these additional subfilters.

Inheritance in GAP

Inheritance works with subfilters.

One declares subfilters and constructs objects that lie in these additional subfilters.

If one needs special methods, these are installed with the subfilters as additional requirements.

Inheritance in GAP

Inheritance works with subfilters.

One declares subfilters and constructs objects that lie in these additional subfilters.

If one needs special methods, these are installed with the subfilters as additional requirements.

Hypothetical example:

```
DeclareCategory("IsGroup",IsObject);
DeclareCategory("IsAbelianGroup",IsGroup);
DeclareOperation("Size",[IsGroup]);
InstallMethod(Size,"for arbitrary groups",
              [IsGroup],
              function(g) ... end);
InstallMethod(Size,"for abelian groups",
              [IsAbelianGroup],
              function(a) ... end);
```

The declarations

```
BindGlobal("BlubbsFamily",
           NewFamily("BlubbsFamily"));
DeclareCategory("IsBlubb",
                IsComponentObjectRep);
DeclareRepresentation("IsBlubbDenseRep",
                      IsBlubb,["wo","p"]);
BindGlobal("BlubbDenseType",
  NewType(BlubbsFamily,IsBlubbDenseRep));
```

The declarations

```
BindGlobal("BlubbsFamily",
           NewFamily("BlubbsFamily"));
DeclareCategory("IsBlubb",
                IsComponentObjectRep);
DeclareRepresentation("IsBlubbDenseRep",
                      IsBlubb,["wo","p"]);
BindGlobal("BlubbDenseType",
  NewType(BlubbsFamily,IsBlubbDenseRep));

DeclareOperation("Blubb",[IsString,IsInt]);
DeclareOperation("IsShort",[IsBlubb]);
DeclareOperation("NrLetters",[IsBlubb]);
```

The declarations

```
BindGlobal("BlubbsFamily",
           NewFamily("BlubbsFamily"));
DeclareCategory("IsBlubb",
                IsComponentObjectRep);
DeclareRepresentation("IsBlubbDenseRep",
                      IsBlubb,["wo","p"]);
BindGlobal("BlubbDenseType",
  NewType(BlubbsFamily,IsBlubbDenseRep));

DeclareOperation("Blubb",[IsString,IsInt]);
DeclareOperation("IsShort",[IsBlubb]);
DeclareOperation("NrLetters",[IsBlubb]);

InstallMethod(Blubb,"constructor",
  [IsString,IsInt], function(s,i)
    local r;
    r := rec(wo:=s,p:=i);
    return Objectify(BlubbDenseType,r);
  end);
```

The implementations

```
InstallMethod(IsShort,"for dense Blubbs",
  [IsBlubbDenseRep],
  function(bl)
    return Length(bl!.wo) <= 5;
  end);
```

The implementations

```
InstallMethod(IsShort,"for dense Blubbs",
  [IsBlubbDenseRep],
  function(bl)
    return Length(bl!.wo) <= 5;
  end);

InstallMethod(NrLetters,"for dense Blubbs",
  [IsBlubbDenseRep],
  function(bl)
    return Length(Set(bl!.wo));
  end);
```

The implementations

```
InstallMethod(IsShort,"for dense Blubbs",
  [IsBlubbDenseRep],
  function(bl)
    return Length(bl!.wo) <= 5;
  end);

InstallMethod(NrLetters,"for dense Blubbs",
  [IsBlubbDenseRep],
  function(bl)
    return Length(Set(bl!.wo));
  end);

InstallMethod(ViewObj,"for dense Blubbs",
  [IsBlubbDenseRep],
  function(bl)
    Print("<a dense blubb wo=",bl!.wo,
          " p=",bl!.p,">");
  end);
```

Usage

One can now use `Blubb`-objects as follows:

```
gap> b := Blubb("abac",17);
<a dense blubb wo=abac p=17>
gap> NrLetters(b);
3
gap> IsShort(b);
true
gap> b!.wo;
"abac"
gap> b!.p;
17
```

Usage

One can now use `Blubb`-objects as follows:

```
gap> b := Blubb("abac",17);
<a dense blubb wo=abac p=17>
gap> NrLetters(b);
3
gap> IsShort(b);
true
gap> b!.wo;
"abac"
gap> b!.p;
17
```

One should install methods for

- `ViewObj` (for the user to see a concise description)
- `PrintObj` (if possible GAP-parsable)
- and possibly `Display` (nicely formatted description for the user).

Properties

A Property “XYZ” is realised by:

- an elementary filter `HasXYZ` and
- an elementary filter `XYZ`.

Properties

A Property “XYZ” is realised by:

- an elementary filter `HasXYZ` and
- an elementary filter `XYZ`.

Properties are declared like this:

```
DeclareProperty("IsShort",IsBlubb);
```

Properties

A Property “XYZ” is realised by:

- an elementary filter `HasXYZ` and
- an elementary filter `XYZ`.

Properties are declared like this:

```
DeclareProperty("IsShort",IsBlubb);
```

This automatically defines

- an elementary filter `HasIsShort`,

Properties

A Property "XYZ" is realised by:

- an elementary filter `HasXYZ` and
- an elementary filter `XYZ`.

Properties are declared like this:

```
DeclareProperty("IsShort",IsBlubb);
```

This automatically defines

- an elementary filter `HasIsShort`,
- an elementary filter `IsShort`,

Properties

A Property "XYZ" is realised by:

- an elementary filter `HasXYZ` and
- an elementary filter `XYZ`.

Properties are declared like this:

```
DeclareProperty("IsShort",IsBlubb);
```

This automatically defines

- an elementary filter `HasIsShort`,
- an elementary filter `IsShort`,
- an operation `IsShort`,

Properties

A Property "XYZ" is realised by:

- an elementary filter `HasXYZ` and
- an elementary filter `XYZ`.

Properties are declared like this:

```
DeclareProperty("IsShort",IsBlubb);
```

This automatically defines

- an elementary filter `HasIsShort`,
- an elementary filter `IsShort`,
- an operation `IsShort`,
- a method for `IsShort` for objects in the filter `IsBlubb and HasIsShort`, which just checks the type, and

Properties

A Property "XYZ" is realised by:

- an elementary filter `HasXYZ` and
- an elementary filter `XYZ`.

Properties are declared like this:

```
DeclareProperty("IsShort",IsBlubb);
```

This automatically defines

- an elementary filter `HasIsShort`,
- an elementary filter `IsShort`,
- an operation `IsShort`,
- a method for `IsShort` for objects in the filter `IsBlubb and HasIsShort`, which just checks the type, and
- an operation with method `SetIsShort`.

Attributes

```
DeclareAttribute("NrLetters",IsBlubb);
```

Attributes

```
DeclareAttribute("NrLetters",IsBlubb);
```

defines automatically

- an elementary filter `HasXYZ`,

Attributes

```
DeclareAttribute("NrLetters",IsBlubb);
```

defines automatically

- an elementary filter `HasXYZ`,
- an operation `XYZ`.

Attributes

```
DeclareAttribute("NrLetters",IsBlubb);
```

defines automatically

- an elementary filter `HasXYZ`,
- an operation `XYZ`.

If one inherits from `IsComponentObjectRep and IsAttributeStoringRep`, then one also gets:

- An operation `SetXYZ` for `[IsBlubb,IsObject]` that stores the 2nd argument in the `!.XYZ`-component and sets `HasXYZ`.

Attributes

```
DeclareAttribute("NrLetters",IsBlubb);
```

defines automatically

- an elementary filter `HasXYZ`,
- an operation `XYZ`.

If one inherits from `IsComponentObjectRep and IsAttributeStoringRep`, then one also gets:

- An operation `SetXYZ` for `[IsBlubb,IsObject]` that stores the 2nd argument in the `!.XYZ`-component and sets `HasXYZ`.
- Every method for `XYZ` stores its result automatically in that component and sets `HasXYZ`.

Attributes

```
DeclareAttribute("NrLetters",IsBlubb);
```

defines automatically

- an elementary filter `HasXYZ`,
- an operation `XYZ`.

If one inherits from `IsComponentObjectRep and IsAttributeStoringRep`, then one also gets:

- An operation `SetXYZ` for `[IsBlubb,IsObject]` that stores the 2nd argument in the `!.XYZ`-component and sets `HasXYZ`.
- Every method for `XYZ` stores its result automatically in that component and sets `HasXYZ`.
- A very highly ranked method for `XYZ` for objects in the filter `IsBlubb and HasXYZ` that simply returns `!.XYZ`.

In our example, we can simply replace

```
DeclareCategory("IsBlubb",
                IsComponentObjectRep);
DeclareOperation("IsShort",[IsBlubb]);
DeclareOperation("NrLetters",[IsBlubb]);
```

by

```
DeclareCategory("IsBlubb",
                IsAttributeStoringRep);
DeclareProperty("IsShort",IsBlubb);
DeclareAttribute("NrLetters",IsBlubb);
```

and automatically get caching:

In our example, we can simply replace

```
DeclareCategory("IsBlubb",
                IsComponentObjectRep);
DeclareOperation("IsShort",[IsBlubb]);
DeclareOperation("NrLetters",[IsBlubb]);
```

by

```
DeclareCategory("IsBlubb",
                IsAttributeStoringRep);
DeclareProperty("IsShort",IsBlubb);
DeclareAttribute("NrLetters",IsBlubb);
```

and automatically get caching:

```
gap> b := Blubb("abac",17);
<a dense blubb wo=abac p=17>
gap> HasNrLetters(b);
false
gap> NrLetters(b);;
gap> HasNrLetters(b);
true
```

Debugging

If you want to see which methods are available:

```
gap> ApplicableMethod(NrLetters,[b],3,"all");
#I  Searching Method for NrLetters with 1 \
                               arguments:
#I  Total: 2 entries
#I  Method 1: ``NrLetters: system getter'',\
                        value: 2*SUM_FLAGS+4
#I   - 1st argument needs \
                 [ "IsAttributeStoringRep",\
                   "Tester(NrLetters)" ]
#I  Method 2: ``NrLetters: for dense \
                          Blubbs'', value: 3
#I  Skipped:
[ function( bl ) ... end ]
```

The complete example

```
BindGlobal("BlubbsFamily",
           NewFamily("BlubbsFamily"));
DeclareCategory("IsBlubb",
                IsAttributeStoringRep);
DeclareRepresentation("IsBlubbDenseRep",
                      IsBlubb,["wo","p"]);
BindGlobal("BlubbDenseType",
  NewType(BlubbsFamily,IsBlubbDenseRep));

DeclareOperation("Blubb",[IsString,IsInt]);
DeclareProperty("IsShort",IsBlubb);
DeclareAttribute("NrLetters",IsBlubb);

InstallMethod(Blubb,"constructor",
  [IsString,IsInt], function(s,i)
    local r;
    r := rec(wo:=s,p:=i);
    return Objectify(BlubbDenseType,r);
  end);
```

The complete example, continued

```
InstallMethod(IsShort,"for dense Blubbs",
  [IsBlubbDenseRep],
  function(bl)
    return Length(bl!.wo) <= 5;
  end);

InstallMethod(NrLetters,"for dense Blubbs",
  [IsBlubbDenseRep],
  function(bl)
    return Length(Set(bl!.wo));
  end);

InstallMethod(ViewObj,"for dense Blubbs",
  [IsBlubbDenseRep],
  function(bl)
    Print("<a dense blubb wo=",bl!.wo,
          " p=",bl!.p,">");
  end);
```