OSCAR: The Work

Reimer Behrends, Thomas Breuer,
Sebastian Gutsche, William Hart

Tübingen, September 25, 2018

Update on progress

- Resources for you - Sebastian Gutsche

Update on progress

- Resources for you - Sebastian Gutsche
- Polymake/Julia integration - Sebastian Gutsche

Update on progress

- Resources for you - Sebastian Gutsche
- Polymake/Julia integration - Sebastian Gutsche
- Gap/Julia integration - Sebastian Gutsche

Update on progress

- Resources for you - Sebastian Gutsche
- Polymake/Julia integration - Sebastian Gutsche
- Gap/Julia integration - Sebastian Gutsche
- Julia in Gap and the future - Thomas Breuer

Update on progress

- Resources for you - Sebastian Gutsche
- Polymake/Julia integration - Sebastian Gutsche
- Gap/Julia integration - Sebastian Gutsche
- Julia in Gap and the future - Thomas Breuer
- Garbage collection - Reimer Behrends

Update on progress

- Resources for you - Sebastian Gutsche
- Polymake/Julia integration - Sebastian Gutsche
- Gap/Julia integration - Sebastian Gutsche
- Julia in Gap and the future - Thomas Breuer
- Garbage collection - Reimer Behrends
- Documentation - Bill Hart

Update on progress

- Resources for you - Sebastian Gutsche
- Polymake/Julia integration - Sebastian Gutsche
- Gap/Julia integration - Sebastian Gutsche
- Julia in Gap and the future - Thomas Breuer
- Garbage collection - Reimer Behrends
- Documentation - Bill Hart
- Maps in OSCAR - Bill Hart

Introducing the OSCAR developers

- Reimer Behrends - TU Kaiserslautern
  - Parallelisation
  - Low-level infrastructure

Introducing the OSCAR developers

- Reimer Behrends - TU Kaiserslautern
  - Parallelisation
  - Low-level infrastructure
- Thomas Breuer - RWTH Aachen
  - Julia in Gap
  - Representation theory

Introducing the OSCAR developers

- Reimer Behrends - TU Kaiserslautern
  - Parallelisation
  - Low-level infrastructure
- Thomas Breuer - RWTH Aachen
  - Julia in Gap
  - Representation theory
- Sebastian Gutsche - University of Siegen
  - GAP/Julia integration
  - Polymake/Julia integration

Introducing the OSCAR developers

- Reimer Behrends - TU Kaiserslautern
  - Parallelisation
  - Low-level infrastructure
- Thomas Breuer - RWTH Aachen
  - Julia in Gap
  - Representation theory
- Sebastian Gutsche - University of Siegen
  - GAP/Julia integration
  - Polymake/Julia integration
- Bill Hart - TU Kaiserslautern
  - Flint - polynomials and linear algebra over concrete rings
  - Nemo.jl - Finitely presented rings in Julia
  - Singular.jl - Julia/Singular integration

Website

All information about the OSCAR project can be found on

https://oscar.computeralgebra.de

Website

All information about the OSCAR project can be found on

https://oscar.computeralgebra.de

On the page you find

- news,
- blog posts,
- interactive examples,
- installation instructions,
- and a list of all people involved.

Resources

OSCAR is an open source and open development project.

Resources

OSCAR is an open source and open development project.

All code can be found on

https://github.com/oscar-system

Resources

OSCAR is an open source and open development project.

All code can be found on

https://github.com/oscar-system

You can contribute to discussions and implementation!

Some notes about polymake

Some notes about polymake

- Polymake is a Perl/C++ hybrid system

Some notes about polymake

- Polymake is a Perl/C++ hybrid system
- Big objects (polytopes, cones, etc.) are stored in Perl data types, small objects in C++ data types

Some notes about polymake

- Polymake is a Perl/C++ hybrid system
- Big objects (polytopes, cones, etc.) are stored in Perl data types, small objects in C++ data types
- To interface polymake, one needs to handle small and big object in Julia, and provide access to all polymake functions (clients)

Some notes about polymake

- Polymake is a Perl/C++ hybrid system
- Big objects (polytopes, cones, etc.) are stored in Perl data types, small objects in C++ data types
- To interface polymake, one needs to handle small and big object in Julia, and provide access to all polymake functions (clients)
- This is possible using the polymake callable library, and a lot of information from polymake itself

Integration of polymake and Julia

First try: Polymake.jl with Lorenz

Integration of polymake and Julia

First try: Polymake.jl with Lorenz

- Integrate polymake and Julia using `Cxx.jl`

Integration of polymake and Julia

First try: Polymake.jl with Lorenz

- Integrate polymake and Julia using `Cxx.jl`
- `Cxx.jl` allows inlining C++ code in Julia

Integration of polymake and Julia

First try: Polymake.jl with Lorenz

- Integrate polymake and Julia using `Cxx.jl`
- `Cxx.jl` allows inlining C++ code in Julia
- Creation of many objects is possible, as calling some functions

Integration of polymake and Julia

First try: Polymake.jl with Lorenz

- Integrate polymake and Julia using `Cxx.jl`
- `Cxx.jl` allows inlining C++ code in Julia
- Creation of many objects is possible, as calling some functions
- But `Cxx.jl` lacks support for many C++(11/14) features polymake relies on

Integration of polymake and Julia

First try: Polymake.jl with Lorenz

- Integrate polymake and Julia using `Cxx.jl`
- `Cxx.jl` allows inlining C++ code in Julia
- Creation of many objects is possible, as calling some functions
- But `Cxx.jl` lacks support for many C++(11/14) features polymake relies on
- So this try failed!

Integration of polymake and Julia

Second try: PolymakeWrap.jl with Kaluba, Lorenz, Timme

Integration of polymake and Julia

Second try: PolymakeWrap.jl with Kaluba, Lorenz, Timme

- Integrate polymake and Julia using `CxxWrap.jl`

Integration of polymake and Julia

Second try: PolymakeWrap.jl with Kaluba, Lorenz, Timme

- Integrate polymake and Julia using `CxxWrap.jl`
- `CxxWrap.jl` lets you create static wrapper for C++ data types and functions, written in pure C++

Integration of polymake and Julia

Second try: PolymakeWrap.jl with Kaluba, Lorenz, Timme

- Integrate polymake and Julia using `CxxWrap.jl`
- `CxxWrap.jl` lets you create static wrapper for C++ data types and functions, written in pure C++
- `CxxWrap.jl` supports enough C++ features

Integration of polymake and Julia

Second try: PolymakeWrap.jl with Kaluba, Lorenz, Timme

- Integrate polymake and Julia using `CxxWrap.jl`
- `CxxWrap.jl` lets you create static wrapper for C++ data types and functions, written in pure C++
- `CxxWrap.jl` supports enough C++ features
- Currently, many small objects and almost all polymake functions are interfaced

Integration of polymake and Julia

Second try: PolymakeWrap.jl with Kaluba, Lorenz, Timme

- Integrate polymake and Julia using `CxxWrap.jl`
- `CxxWrap.jl` lets you create static wrapper for C++ data types and functions, written in pure C++
- `CxxWrap.jl` supports enough C++ features
- Currently, many small objects and almost all polymake functions are interfaced
- Next structural iteration coming soon (this year)

Polymake: Example

Polymake: Example

```
julia> P = PolymakeWrap.rand_sphere(6,20)
pm::Polytope<Rational>
```

Polymake: Example

```
julia> P = PolymakeWrap.rand_sphere(6,20)
pm::Polytope<Rational>

julia> PolymakeWrap.give( P, "F_VECTOR" )
pm::Vector<pm::Integer>
20 164 623 1149 1005 335
```

Polymake: Example

```
julia> P = PolymakeWrap.rand_sphere(6,20)
pm::Polytope<Rational>

julia> PolymakeWrap.give( P, "F_VECTOR" )
pm::Vector<pm::Integer>
20 164 623 1149 1005 335
```

- Conversion from and to certain small objects

Polymake: Example

```
julia> P = PolymakeWrap.rand_sphere(6,20)
pm::Polytope<Rational>

julia> PolymakeWrap.give( P, "F_VECTOR" )
pm::Vector<pm::Integer>
20 164 623 1149 1005 335
```

- Conversion from and to certain small objects
- Creation of big objects

Polymake: Example

```
julia> P = PolymakeWrap.rand_sphere(6,20)
pm::Polytope<Rational>

julia> PolymakeWrap.give( P, "F_VECTOR" )
pm::Vector<pm::Integer>
20 164 623 1149 1005 335
```

- Conversion from and to certain small objects
- Creation of big objects
- Possibility to call many polymake clients

Polymake: Example

```
julia> P = PolymakeWrap.rand_sphere(6,20)
pm::Polytope<Rational>

julia> PolymakeWrap.give( P, "F_VECTOR" )
pm::Vector<pm::Integer>
20 164 623 1149 1005 335
```

- Conversion from and to certain small objects
- Creation of big objects
- Possibility to call many polymake clients
- Current issue: Interfaces to the remaining small objects

Polymake: Example

```
julia> P = PolymakeWrap.rand_sphere(6,20)
pm::Polytope<Rational>

julia> PolymakeWrap.give( P, "F_VECTOR" )
pm::Vector<pm::Integer>
20 164 623 1149 1005 335
```

- Conversion from and to certain small objects
- Creation of big objects
- Possibility to call many polymake clients
- Current issue: Interfaces to the remaining small objects and remaining clients

Integration of polymake and Julia

Next iteration for the polymake Julia interface

Integration of polymake and Julia

Next iteration for the polymake Julia interface

1. Export data from polymake about its clients (in a JSON format)

Integration of polymake and Julia

Next iteration for the polymake Julia interface

1. Export data from polymake about its clients (in a JSON format)
2. We use Julia to translate the JSON data into C++ wrapper code

Integration of polymake and Julia

Next iteration for the polymake Julia interface

1. Export data from polymake about its clients (in a JSON format)
2. We use Julia to translate the JSON data into C++ wrapper code
3. Using `CxxWrap.jl`, we can compile this wrapper and load all polymake functionality into Julia

Next iteration for the polymake Julia interface

1. Export data from polymake about its clients (in a JSON format)
2. We use Julia to translate the JSON data into C++ wrapper code
3. Using `CxxWrap.jl`, we can compile this wrapper and load all polymake functionality into Julia
4. ...

Integration of polymake and Julia

Next iteration for the polymake Julia interface

1. Export data from polymake about its clients (in a JSON format)
2. We use Julia to translate the JSON data into C++ wrapper code
3. Using `CxxWrap.jl`, we can compile this wrapper and load all polymake functionality into Julia
4. ...
5. SUCCESS!

GAP: JuliaInterface and GAP.jl

GAP: JuliaInterface and GAP.jl

GAP package JuliaInterface and Julia module GAP.jl

GAP: JuliaInterface and GAP.jl

GAP package JuliaInterface and Julia module GAP.jl

$$\text{GAP} \longleftrightarrow \text{Julia}$$

GAP: JuliaInterface and GAP.jl

GAP package JuliaInterface and Julia module GAP.jl

$$\text{GAP} \longleftrightarrow \text{Julia}$$

JuliaInterface and GAP.jl provide

GAP: JuliaInterface and GAP.jl

GAP package JuliaInterface and Julia module GAP.jl

$$\text{GAP} \longleftrightarrow \text{Julia}$$

JuliaInterface and GAP.jl provide

- Conversion of basic data types (e.g., integers, lists, permutations) between GAP and Julia

GAP: JuliaInterface and GAP.jl

GAP package JuliaInterface and Julia module GAP.jl

$$\text{GAP} \longleftrightarrow \text{Julia}$$

JuliaInterface and GAP.jl provide

- Conversion of basic data types (e.g., integers, lists, permutations) between GAP and Julia
- Use of GAP data types in Julia and Julia data types in GAP

GAP: JuliaInterface and GAP.jl

GAP package JuliaInterface and Julia module GAP.jl

$$\text{GAP} \longleftrightarrow \text{Julia}$$

JuliaInterface and GAP.jl provide

- Conversion of basic data types (e.g., integers, lists, permutations) between GAP and Julia
- Use of GAP data types in Julia and Julia data types in GAP
- Use of Julia functions in GAP and GAP functions in Julia

GAP: JuliaInterface and GAP.jl

GAP package JuliaInterface and Julia module GAP.jl

$$\text{GAP} \longleftrightarrow \text{Julia}$$

JuliaInterface and GAP.jl provide

- Conversion of basic data types (e.g., integers, lists, permutations) between GAP and Julia
- Use of GAP data types in Julia and Julia data types in GAP
- Use of Julia functions in GAP and GAP functions in Julia

https://github.com/oscar-system/GAPJulia

JuliaInterface data structures: Objects

JuliaInterface contains GAP data structures that hold pointers to Julia objects:

JuliaInterface data structures: Objects

JuliaInterface contains GAP data structures that hold pointers to Julia objects:

```
gap> a := 2;
2
```

JuliaInterface data structures: Objects

JuliaInterface contains GAP data structures that hold pointers to Julia objects:

```
gap> a := 2;
2

gap> b := ConvertedToJulia( a );
<Julia: 2>
```

JuliaInterface data structures: Objects

JuliaInterface contains GAP data structures that hold pointers to Julia objects:

```
gap> a := 2;
2

gap> b := ConvertedToJulia( a );
<Julia: 2>

gap> ConvertedFromJulia( b );
2
```

JuliaInterface data structures: Objects

JuliaInterface contains GAP data structures that hold pointers to Julia objects:

```
gap> a := 2;
2

gap> b := ConvertedToJulia( a );
<Julia: 2>

gap> ConvertedFromJulia( b );
2
```

Possible conversions:

- (small) Integers

JuliaInterface data structures: Objects

JuliaInterface contains GAP data structures that hold pointers to Julia objects:

```
gap> a := 2;
2

gap> b := ConvertedToJulia( a );
<Julia: 2>

gap> ConvertedFromJulia( b );
2
```

Possible conversions:

- (small) Integers
- Floats

JuliaInterface data structures: Objects

JuliaInterface contains GAP data structures that hold pointers to Julia objects:

```
gap> a := 2;
2

gap> b := ConvertedToJulia( a );
<Julia: 2>

gap> ConvertedFromJulia( b );
2
```

Possible conversions:

- (small) Integers
- Floats
- Strings

JuliaInterface data structures: Objects

JuliaInterface contains GAP data structures that hold pointers to Julia objects:

```
gap> a := 2;
2

gap> b := ConvertedToJulia( a );
<Julia: 2>

gap> ConvertedFromJulia( b );
2
```

Possible conversions:

- (small) Integers
- Floats
- Strings
- Booleans

JuliaInterface data structures: Objects

JuliaInterface contains GAP data structures that hold pointers to Julia objects:

```
gap> a := 2;
2

gap> b := ConvertedToJulia( a );
<Julia: 2>

gap> ConvertedFromJulia( b );
2
```

Possible conversions:

- (small) Integers
- Floats
- Strings
- Booleans
- Nested lists of the above to Arrays or Tuples

JuliaInterface data structures: Functions

JuliaInterface provides the possibility to call Julia functions by converting GAP objects:

JuliaInterface data structures: Functions

JuliaInterface provides the possibility to call Julia functions by converting GAP objects:

```
gap>  ImportJuliaModuleIntoGAP( "Base" );
```

JuliaInterface data structures: Functions

JuliaInterface provides the possibility to call Julia functions by converting GAP objects:

```
gap>  ImportJuliaModuleIntoGAP( "Base" );

gap>  Julia.Base.sqrt( 4 );
<Julia: 2.0>
```

JuliaInterface data structures: Functions

JuliaInterface provides the possibility to call Julia functions by converting GAP objects:

```
gap> ImportJuliaModuleIntoGAP( "Base" );

gap> Julia.Base.sqrt( 4 );
<Julia: 2.0>
```

- Julia functions can be used like GAP functions

JuliaInterface data structures: Functions

JuliaInterface provides the possibility to call Julia functions by converting GAP objects:

```
gap>  ImportJuliaModuleIntoGAP( "Base" );

gap>  Julia.Base.sqrt( 4 );
<Julia: 2.0>
```

- Julia functions can be used like GAP functions
- Input data can be converted to Julia, or passed as GAP object pointers to Julia

JuliaInterface data structures: Functions

JuliaInterface provides the possibility to call Julia functions by converting GAP objects:

```
gap> ImportJuliaModuleIntoGAP( "Base" );

gap> Julia.Base.sqrt( 4 );
<Julia: 2.0>
```

- Julia functions can be used like GAP functions
- Input data can be converted to Julia, or passed as GAP object pointers to Julia
- Method dispatch is handled by Julia itself

GAP.jl: using GAP from Julia

The Julia module GAP.jl provides access to GAP's data structures and functions from Julia

GAP.jl: using GAP from Julia

The Julia module GAP.jl provides access to GAP's data structures and functions from Julia

```
julia> S3 = GAP.SymmetricGroup( LibGAP.to_gap( 3 ) )
GAP: SymmetricGroup( [ 1 .. 3 ] )
```

GAP.jl: using GAP from Julia

The Julia module GAP.jl provides access to GAP's data structures and functions from Julia

```
julia> S3 = GAP.SymmetricGroup( LibGAP.to_gap( 3 ) )
GAP: SymmetricGroup( [ 1 .. 3 ] )

julia> size_gap = GAP.Size( S3 )
GAP: 6
```

GAP.jl: using GAP from Julia

The Julia module GAP.jl provides access to GAP's data structures and functions from Julia

```
julia> S3 = GAP.SymmetricGroup( LibGAP.to_gap( 3 ) )
GAP: SymmetricGroup( [ 1 .. 3 ] )

julia> size_gap = GAP.Size( S3 )
GAP: 6

julia> LibGAP.from_gap( size_gap, Int64 )
6
```

GAP: How far we got: function calls

Previously: Calling Julia functions from GAP had a massive overhead.

GAP: How far we got: function calls

Now: Calling Julia functions from GAP works with no overhead:

GAP: How far we got: function calls

Now: Calling Julia functions from GAP works with no overhead:

Calling a pure GAP function

GAP: How far we got: function calls

Now: Calling Julia functions from GAP works with no overhead:

Calling a pure GAP function

```
gap>  ListX([1..10^5], [1..10], {i,j} -> i);; time;
207
```

GAP: How far we got: function calls

Now: Calling Julia functions from GAP works with no overhead:

Calling a pure GAP function

```
gap> ListX([1..10^5], [1..10], {i,j} -> i);; time;
207
```

Calling a (variadic) C function

GAP: How far we got: function calls

Now: Calling Julia functions from GAP works with no overhead:

Calling a pure GAP function

```
gap> ListX([1..10^5], [1..10], {i,j} -> i);; time;
207
```

Calling a (variadic) C function

```
gap> ListX([1..10^5], [1..10], ReturnFirst);; time;
207
```

GAP: How far we got: function calls

Now: Calling Julia functions from GAP works with no overhead:

Calling a pure GAP function

```
gap> ListX([1..10^5], [1..10], {i,j} -> i);; time;
207
```

Calling a (variadic) C function

```
gap> ListX([1..10^5], [1..10], ReturnFirst);; time;
207
```

Calling a Julia function (compiled via `@cfunction`)

GAP: How far we got: function calls

Now: Calling Julia functions from GAP works with no overhead:

Calling a pure GAP function

```
gap> ListX([1..10^5], [1..10], {i,j} -> i);; time;
207
```

Calling a (variadic) C function

```
gap> ListX([1..10^5], [1..10], ReturnFirst);; time;
207
```

Calling a Julia function (compiled via `@cfunction`)

```
gap> ListX([1..10^5], [1..10], ReturnFirstJL);; time;
195
```

Ongoing work: GAP–Julia integration

- use Singular from GAP, via `Singular.jl`
- use Antic from GAP, via `Nemo.jl`
- develop examples how to use GAP–Julia integration in research.

An example: Use Julia for speedup.

$q, n \in \mathbb{N}, q > 1$

e dividing $q^n - 1$

$z = (q^n - 1)/e$

field F

$A = A(q, n, e) = \bigoplus_{i=0}^{z} Fb_i$

with multiplication

$$b_i b_j = \begin{cases} b_{i+j} & ; \text{ no carry in } q\text{-adic addition } ie + je \\ 0 & ; \text{ otherwise} \end{cases}$$

$J(A)$ Jacobson radical

$(\dim(J(A)^{i-1}/J(A)^i))_{i \geq 0}$ Loewy structure of A

$LL(A) = \min\{i; J(A)^i = \{0\}\}$ Loewy length

Implement $A(q, n, e)$

in GAP: algebra via structure constants table

deal with the algebra, its elements, substructures

```
gap> a:= SingerAlgebra( 5, 2, 4 );
A(5,2,4)
gap> DimensionsLoewyFactors( a );
[ 1, 5, 1 ]
gap> LoewyLength( a );
3
gap> a:= SingerAlgebra( 5, 2, 6 );
A(5,2,6)
gap> DimensionsLoewyFactors( a );
[ 1, 1, 1, 1, 1 ]
gap> LoewyLength( a );
5
```

Implement $A(q, n, e)$

```
gap> a:= SingerAlgebra( 6, 11, 115 );
A(6,11,115)
gap> LoewyLength( a );
12
```

Implement $A(q, n, e)$

```
gap> a:= SingerAlgebra( 6, 11, 115 );
A(6,11,115)
gap> LoewyLength( a );
12

gap> Dimension( a );
3154758
```

Combinatorial setup for $A = A(q, n, e)$

- for computing $LL(A)$, we do not need to deal with elements of A
- interpret $LL(A) - 1$ as length of a longest nonzero product of b_i
- distribute the b_i to Loewy layers
- in GAP: possible but slow
- try to combine GAP and Julia

The Julia part

```
function LoewyLayersData( q::Int, n::Int, e )
ord = div( q^n - 1, e )      # deal with integer overflow!
monomials = [ zeros( Int, n ) ]
layers = [ 1 ]
for i in 1:ord
  mon = coeffs( i, q, n )    # a small julia function
  lambda = 1
  for j in 2:i
    if lambda < layers[j]
         && islessorequal( monomials[j], mon, n )
      lambda = layers[j]
    end
  end
  push!( monomials, mon )
  push!( layers, lambda + 1 )
end
return Dict( "monomials" => monomials, "layers" => layers )
end;
```

The GAP part

```
DeclareAttribute( "LoewyStructureInfo", IsSingerAlgebra );

InstallMethod( LoewyStructureInfo,
 [ "IsSingerAlgebra" ],
 A -> ConvertedFromJuliaRecordFromDictionary(
       CallFuncList( Julia.LoewyStructure.LoewyLayersData,
        ParametersOfSingerAlgebra( A ) ) ) );

DeclareAttribute( "DimensionsLoewyFactors", IsSingerAlgebra );

InstallMethod( DimensionsLoewyFactors,
 [ "IsSingerAlgebra" ],
 A -> StructuralConvertedFromJulia(
       Julia.LoewyStructure.LoewyVector(
        LoewyStructureInfo( A ) ) ) );
```

Results

- speedup by a factor of 10 (Julia vs. GAP)
- extensible: let Julia compute more data (later)
- more elaborate version:
  - about 700 lines of Julia code
  - about 350 lines of GAP code

Lessons learned

- reasonable Julia code can look very similar to reasonable GAP code
- be aware of, e. g., integer overflow in Julia
- avoid local Julia functions
- ...

Julia GC in GAP — the short version

- `cd gap`
- `./configure --with-gc=julia`
  `--with-julia=/path/to/julia/usr`
- make
- ./gap

```
+--------+   GAP 4.8.8-6005-g64b84d0 of today
|  GAP   |   https://www.gap-system.org
+--------+   Architecture: x86_64-pc-linux-gnu-default64
Configuration:  gmp 6.1.2, Julia 1.1.0-DEV, readline
Loading the library and packages ...
```

Garbage collection basics

- Identify all reachable objects.
- Reachable
- = referenced by a local or global variable (roots) or
- = referenced by another reachable object (repeat recursively).
- Discard all unreachable objects.

Problem 1: GAP vs. Julia object layouts

- Julia: Records or arrays of scalars/records.
- GAP: Typically, list of tagged pointers.
- $\Rightarrow$ Cannot describe GAP object layout in a way that the Julia GC understands.

Problem 2: Global roots

- Julia: All global roots must be variables in a Julia module.
- GAP: Roots can be arbitrary C variables that can be updated from C code.
- $\Rightarrow$ No possibility to tell the Julia GC about them.

Problem 3: Local roots & stack scanning

- Julia: Julia knows the layout of the Julia stack and tracks variables there.
- GAP: We do not always know the layout of C stack frames/registers and even if we did, we could not easily tell Julia about that.
- GAP uses a *conservative* approach to stack scanning.
- $\Rightarrow$ Difficult to even determine which objects are referenced by local variables.

Making the Julia GC work for GAP

New Julia GC extensions for foreign code (not just GAP):

1. Support custom mark functions for foreign types.
2. Allow foreign code to supply additional roots.
3. Support conservative scanning to identify local variables.

Result: Pull request #28368 for Julia on GitHub (approved, though not yet merged).

The next GAP release (4.10, November 2018) will already support Julia integration.

Documentation

{Demo documentation}

What infrastructure is needed for a CAS?

```
function gcd(a, b)
   # do something
end

d = gcd(a, b)
```

Distinguishing functions (dispatch)

```
f = x^2 + 2x + 3
g = x^3 + 3x + 1

d = f.gcd(g)
```

Multimethods

```
function gcd(f::Poly, g::Poly)
   # do something
end

d = gcd(f, g)
```

Parameterised types

```
function gcd(f::Poly{T}, g::Poly{T})
                where T <: FieldElement
   # do something
end

d = gcd(f, g)
```

Too much parameterisation!

```
function gcd(f::Poly{Zmod{T}}, g::Poly{Zmod{T}})
                where T
   # do something
end

d = gcd(f, g)
```

Maps

```
function myfun(f::Map, n::Integer)
   # do something
end

d = myfun(f, 12)
```

Different kinds of maps

- Maps between groups/rings/modules/etc.

Different kinds of maps

- Maps between groups/rings/modules/etc.
- Cached maps

Different kinds of maps

- Maps between groups/rings/modules/etc.
- Cached maps
- Composite maps

Different kinds of maps

- Maps between groups/rings/modules/etc.
- Cached maps
- Composite maps
- Identity maps

Different kinds of maps

- Maps between groups/rings/modules/etc.
- Cached maps
- Composite maps
- Identity maps
- Maps with retractions/sections

Different kinds of maps

- Maps between groups/rings/modules/etc.
- Cached maps
- Composite maps
- Identity maps
- Maps with retractions/sections
- Maps as morphisms in a category

Maps between domains

```
function myfun(f::Map{C, D}, n::Integer)
                  where C <: Group, D <: Group
   # do something
end

d = myfun(f, 12)
```

Inheritance and traits

May want maps to have certain features:

```
function myfun(f::Map{C, D, T}, n::Integer)
                  where C <: Group, D <: Group,
                        T <: IsCacheable
   # do something
end

d = myfun(f, 12)
```

Inheritance and traits

May want maps to have certain features:

```
function myfun(f::Map{C, D, T}, n::Integer)
                  where C <: Group, D <: Group,
                        T <: IsCacheable
   # do something
end

d = myfun(f, 12)
```

Problem : no multiple inheritance, need parameter for each new "trait"

Additional problems

- May also want traits to inherit

Additional problems

- May also want traits to inherit
- What about classes of map (CompositeMap, CachedMap, etc.)

Four parameter types

```
function myfun(f::Map{C, D, T, U}, n::Integer)
                  where C <: Group, D <: Group,
                        T <: MapClass, U <: MapType
   # do something
end

d = myfun(f, 12)
```

Usability improvements

```
function myfun(f::Map)

function myfun(f::Map(C, D))

function myfun(f::Map(CompositeMap))

function myfun(f::Map(MyMap))

function myfun(f::Map(C, D, MyMap))
```