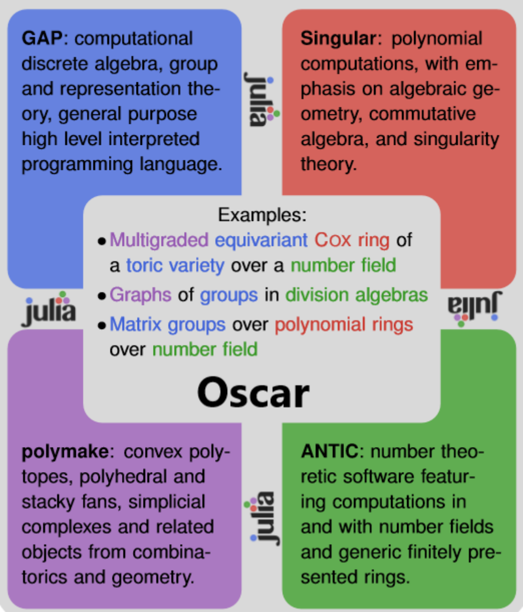
OSCAR: A visionary, new computer algebra system

William Hart, Sebastian Gutsche
Reimer Behrends, Thomas Breuer

September 27, 2017

Develop a visionary, next generation, open source computer algebra system, integrating all systems, libraries and packages developed within the TRR.


GAP: computational discrete algebra, group and representation theory, general purpose high level interpreted programming language.
Singular: polynomial computations, with emphasis on algebraic geometry, commutative algebra, and singularity theory.
julia
Examples:
• Multigraded equivariant Cox ring of a toric variety over a number field
• Graphs of groups in division algebras
• Matrix groups over polynomial rings over number field
Oscar
polymake: convex polytopes, polyhedral and stacky fans, simplicial complexes and related objects from combinatorics and geometry.
ANTIC: number theoretic software featuring computations in and with number fields and generic finitely presented rings.

Update on progress

- Antic number theory software - Bill Hart

Update on progress

- Antic number theory software - Bill Hart
- Singular.jl - integrating Singular and Julia - Bill Hart

Update on progress

- Antic number theory software - Bill Hart
- Singular.jl - integrating Singular and Julia - Bill Hart
- Gap/Julia integration - Sebastian Gutsche

Update on progress

- Antic number theory software - Bill Hart
- Singular.jl - integrating Singular and Julia - Bill Hart
- Gap/Julia integration - Sebastian Gutsche
- Garbage collection - Reimer Behrends

Update on progress

- Antic number theory software - Bill Hart
- Singular.jl - integrating Singular and Julia - Bill Hart
- Gap/Julia integration - Sebastian Gutsche
- Garbage collection - Reimer Behrends
- Julia in Gap and the future - Thomas Breuer

Update on progress

- Antic number theory software - Bill Hart
- Singular.jl - integrating Singular and Julia - Bill Hart
- Gap/Julia integration - Sebastian Gutsche
- Garbage collection - Reimer Behrends
- Julia in Gap and the future - Thomas Breuer

Introducing the OSCAR developers

- Bill Hart - TU Kaiserslautern
  - Flint - polynomials and linear algebra over concrete rings
  - Nemo.jl - Finitely presented rings in Julia
  - Singular.jl - Julia/Singular integration

Introducing the OSCAR developers

- Bill Hart - TU Kaiserslautern
  - Flint - polynomials and linear algebra over concrete rings
  - Nemo.jl - Finitely presented rings in Julia
  - Singular.jl - Julia/Singular integration
- Sebastian Gutsche - Siegen University
  - JuliaInterface/GAP.jl - Julia/GAP integration
  - Julia/polymake integration
  - CAP: Categorical programming

Introducing the OSCAR developers

- Bill Hart - TU Kaiserslautern
  - Flint - polynomials and linear algebra over concrete rings
  - Nemo.jl - Finitely presented rings in Julia
  - Singular.jl - Julia/Singular integration
- Sebastian Gutsche - Siegen University
  - JuliaInterface/GAP.jl - Julia/GAP integration
  - Julia/polymake integration
  - CAP: Categorical programming
- Reimer Behrends - TU Kaiserslautern
  - Parallelisation
  - Low-level infrastructure

Introducing the OSCAR developers

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

Others involved in OSCAR

- Hecke: Claus Fieker, Tommy Hofmann, Carlo Sircana

Others involved in OSCAR

- Hecke: Claus Fieker, Tommy Hofmann, Carlo Sircana
- Singular: Hans Schoenemann, Janko Boehm, others

Others involved in OSCAR

- Hecke: Claus Fieker, Tommy Hofmann, Carlo Sircana
- Singular: Hans Schoenemann, Janko Boehm, others
- PI's: Mohamed Barakat, Wolfram Decker, Claus Fieker, Frank Lübeck, Michael Joswig

Others involved in OSCAR

- Hecke: Claus Fieker, Tommy Hofmann, Carlo Sircana
- Singular: Hans Schoenemann, Janko Boehm, others
- PI's: Mohamed Barakat, Wolfram Decker, Claus Fieker, Frank Lübeck, Michael Joswig
- ... You !!??

Others involved in OSCAR

- Hecke: Claus Fieker, Tommy Hofmann, Carlo Sircana
- Singular: Hans Schoenemann, Janko Boehm, others
- PI's: Mohamed Barakat, Wolfram Decker, Claus Fieker, Frank Lübeck, Michael Joswig
- ... You !!??

We are looking for projects that:

- Can be broken down into fundamentals

Others involved in OSCAR

- Hecke: Claus Fieker, Tommy Hofmann, Carlo Sircana
- Singular: Hans Schoenemann, Janko Boehm, others
- PI's: Mohamed Barakat, Wolfram Decker, Claus Fieker, Frank Lübeck, Michael Joswig
- ... You !!??

We are looking for projects that:

- Can be broken down into fundamentals
- Pieces are represented in the four cornerstone systems

Others involved in OSCAR

- Hecke: Claus Fieker, Tommy Hofmann, Carlo Sircana
- Singular: Hans Schoenemann, Janko Boehm, others
- PI's: Mohamed Barakat, Wolfram Decker, Claus Fieker, Frank Lübeck, Michael Joswig
- ... You !!??

We are looking for projects that:

- Can be broken down into fundamentals
- Pieces are represented in the four cornerstone systems
- Relevant to the TRR

Antic cornerstone

C libraries:

- Flint - polynomials and linear algebra

Antic cornerstone

C libraries:

- Flint - polynomials and linear algebra
- Antic - number field arith.

Antic cornerstone

C libraries:

- Flint - polynomials and linear algebra
- Antic - number field arith.
- MPIR (fork of GMP) - bignum arithmetic

Antic cornerstone

C libraries:

- Flint - polynomials and linear algebra
- Antic - number field arith.
- MPIR (fork of GMP) - bignum arithmetic

Julia libraries:

- Nemo.jl - generic, finitely presented rings

Antic cornerstone

C libraries:

- Flint - polynomials and linear algebra
- Antic - number field arith.
- MPIR (fork of GMP) - bignum arithmetic

Julia libraries:

- Nemo.jl - generic, finitely presented rings
- Hecke.jl - number fields, class field theory, algebraic number theory

New features in Flint

- Quadratic sieve integer factorisation

New features in Flint

- Quadratic sieve integer factorisation
- Elliptic curve integer factorisation

New features in Flint

- Quadratic sieve integer factorisation
- Elliptic curve integer factorisation
- APRCL primality test

New features in Flint

- Quadratic sieve integer factorisation
- Elliptic curve integer factorisation
- APRCL primality test
- Parallelised FFT

New features in Flint

- Quadratic sieve integer factorisation
- Elliptic curve integer factorisation
- APRCL primality test
- Parallelised FFT
- Howell form

New features in Flint

- Quadratic sieve integer factorisation
- Elliptic curve integer factorisation
- APRCL primality test
- Parallelised FFT
- Howell form
- Characteristic and minimal polynomial

New features in Flint

- Quadratic sieve integer factorisation
- Elliptic curve integer factorisation
- APRCL primality test
- Parallelised FFT
- Howell form
- Characteristic and minimal polynomial
- van Hoeij factorisation for $\mathbb{Z}[x]$

New features in Flint

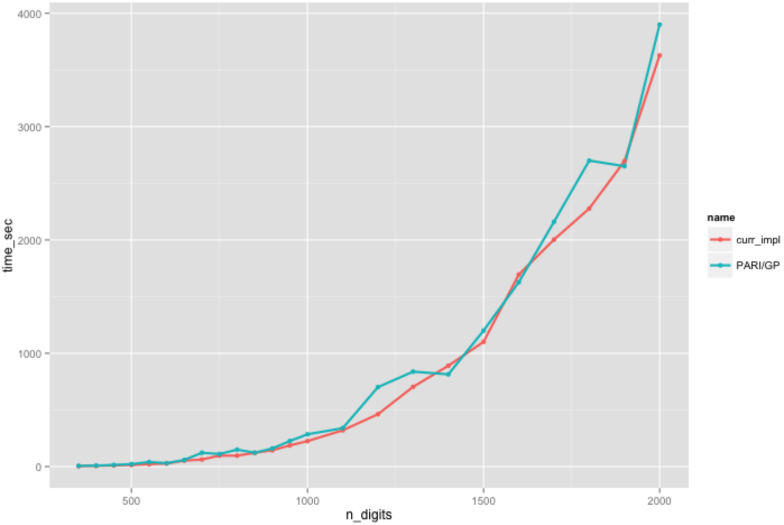
- Quadratic sieve integer factorisation
- Elliptic curve integer factorisation
- APRCL primality test
- Parallelised FFT
- Howell form
- Characteristic and minimal polynomial
- van Hoeij factorisation for $\mathbb{Z}[x]$
- Multivariate polynomial arithmetic $\mathbb{Z}[x, y, z, \ldots]$

Integer factorisation : Quadratic sieve

Table: Quadratic sieve timings

Digits	Pari/GP	Flint (1 core)	Flint (4 cores)
50	0.43	0.55	0.39
59	3.8	3.0	1.7
68	38	21	14
77	257	140	52
83	2200	1500	540

APRCL primality test timings



FFT: Integer and polynomial multiplication

Table: FFT timings

Words	1 core	4 cores	8 cores
110k	0.07s	0.05s	0.05s
360k	0.3s	0.1	0.1s
1.3m	1.1s	0.4s	0.3s
4.6m	4.5s	1.5s	1.0s
26m	28s	9s	6s
120m	140s	48s	33s
500m	800s	240s	150s

Characteristic and minimal polynomial

Table: Charpoly and minpoly timings

Op	Sage 6.9	Pari 2.7.4	Magma 2.21-4	Giac 1.2.2	Flint
Charpoly	0.2s	0.6s	0.06s	0.06s	0.04s
Minpoly	0.07s	>160 hrs	0.05s	0.06s	0.04s

for 80×80 matrix over $\mathbb{Z}$ with entries in $[-20, 20]$ and minpoly of degree 40.

Multivariate multiplication

Table: “Dense” Fateman multiply bench

n	Sage	Singular	Magma	Giac	Piranha	Trip	Flint
5	0.0063s	0.0048s	0.0018s	0.00023s	0.0011s	0.00057s	0.00023s
10	0.51s	0.11s	0.12s	0.0056s	0.029s	0.023s	0.0043s
15	9.1s	1.4s	1.9s	0.11s	0.39s	0.21s	0.045s
20	75s	21s	16s	0.62s	2.9s	2.3s	0.48s
25	474s	156s	98s	2.8s	14s	12s	2.3s
30	1667s	561s	440s	14s	56s	41s	10s

4 variables

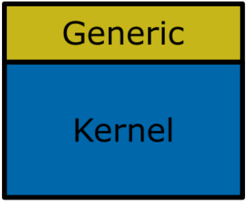
Multivariate multiplication

Table: Sparse multiply benchmark

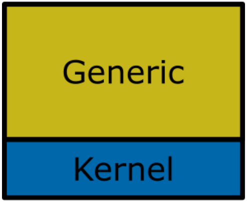
n	Sage	Singular	Magma	Giac	Piranha	Trip	Flint
4	0.0066s	0.0050s	0.0062s	0.0046s	0.0033s	0.0015s	0.0014s
6	0.15s	0.11s	0.080s	0.030s	0.025s	0.016s	0.016s
8	1.6s	0.79s	0.68s	0.28s	0.15s	0.10s	0.10s
10	8s	3.6s	3.0s	1.5s	0.62s	0.40s	0.48s
12	43s	14s	11s	4.8s	2.2s	2.2s	2.0s
14	173s	63s	37s	14s	6.7s	12s	7.2s
16	605s	201s	94s	39s	20s	39s	19s

5 variables

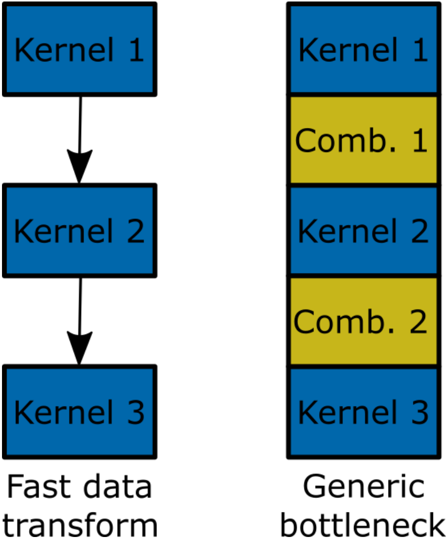
Efficient generics



Fast generics



Slow generics


Kernel 1
Kernel 2
Kernel 3
Fast data transform
Kernel 1
Comb. 1
Kernel 2
Comb. 2
Kernel 3
Generic bottleneck

- JIT compilation : near C performance.
- Designed by mathematically minded people.
- Open Source (MIT License).
- Actively developed since 2009.
- Supports Windows, OSX, Linux, BSD.
- Friendly C/Python-like (imperative) syntax.

Interfaces to C libraries:

- Flint : univariate polys and matrices over $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}/p\mathbb{Z}$, F_q, Q_p

Interfaces to C libraries:

- Flint : univariate polys and matrices over $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}/p\mathbb{Z}$, F_q, Q_p
- Arb : ball arithmetic, univariate polys and matrices over $\mathbb{R}$ and $\mathbb{C}$, special and transcendental functions

Interfaces to C libraries:

- Flint : univariate polys and matrices over $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}/p\mathbb{Z}$, F_q, Q_p
- Arb : ball arithmetic, univariate polys and matrices over $\mathbb{R}$ and $\mathbb{C}$, special and transcendental functions
- Antic : element arithmetic over abs. number fields

Interfaces to C libraries:

- Flint : univariate polys and matrices over $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}/p\mathbb{Z}$, F_q, Q_p
- Arb : ball arithmetic, univariate polys and matrices over $\mathbb{R}$ and $\mathbb{C}$, special and transcendental functions
- Antic : element arithmetic over abs. number fields

Nemo capabilities:

- Generic rings: residue rings, fraction fields, dense univariate polynomials, sparse distributed multivariate polynomials

Interfaces to C libraries:

- Flint : univariate polys and matrices over $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}/p\mathbb{Z}$, F_q, Q_p
- Arb : ball arithmetic, univariate polys and matrices over $\mathbb{R}$ and $\mathbb{C}$, special and transcendental functions
- Antic : element arithmetic over abs. number fields

Nemo capabilities:

- Generic rings: residue rings, fraction fields, dense univariate polynomials, sparse distributed multivariate polynomials, dense linear algebra, power series, permutation groups

Interfaces to C libraries:

- Flint : univariate polys and matrices over $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}/p\mathbb{Z}$, F_q, Q_p
- Arb : ball arithmetic, univariate polys and matrices over $\mathbb{R}$ and $\mathbb{C}$, special and transcendental functions
- Antic : element arithmetic over abs. number fields

Nemo capabilities:

- Generic rings: residue rings, fraction fields, dense univariate polynomials, sparse distributed multivariate polynomials, dense linear algebra, power series, permutation groups

Highlights:

Generic polynomial resultant, charpoly, minpoly over an integrally closed domain, Smith and Hermite normal form, Popov form, fast generic determinant, fast sparse multivariate arithmetic

Singular.jl

Access to Singular kernel functions and data types:

- Coefficient rings $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}/n\mathbb{Z}$, GF(p), etc.

Singular.jl

Access to Singular kernel functions and data types:

- Coefficient rings $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}/n\mathbb{Z}$, $\mathrm{GF}(p)$, etc.
- Polynomials, ideals, modules, matrices, etc.

Singular.jl

Access to Singular kernel functions and data types:

- Coefficient rings $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}/n\mathbb{Z}$, GF(p), etc.
- Polynomials, ideals, modules, matrices, etc.
- Groebner basis, resolutions, syzygies

Singular.jl

Access to Singular kernel functions and data types:

- Coefficient rings $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}/n\mathbb{Z}$, GF(p), etc.
- Polynomials, ideals, modules, matrices, etc.
- Groebner basis, resolutions, syzygies

Integration with Nemo.jl:

- Singular polynomials over any Nemo coefficient ring, e.g. Groebner bases over cyclotomic fields

Singular.jl

Access to Singular kernel functions and data types:

- Coefficient rings $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}/n\mathbb{Z}$, $\mathrm{GF}(p)$, etc.
- Polynomials, ideals, modules, matrices, etc.
- Groebner basis, resolutions, syzygies

Integration with Nemo.jl:

- Singular polynomials over any Nemo coefficient ring, e.g. Groebner bases over cyclotomic fields
- Nemo generics over any Singular ring

JuliaInterface

GAP package JuliaInterface

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

GAP package JuliaInterface

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

JuliaInterface provides

JuliaInterface

GAP package JuliaInterface

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

JuliaInterface provides

- Conversions of GAP to Julia data and vice versa

GAP package JuliaInterface

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

JuliaInterface provides

- Conversions of GAP to Julia data and vice versa
- Data structures for Julia objects and functions in GAP

JuliaInterface

GAP package JuliaInterface

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

JuliaInterface provides

- Conversions of GAP to Julia data and vice versa
- Data structures for Julia objects and functions in GAP
- Possibility to add compiled Julia functions as kernel functions to GAP

JuliaInterface data structures: Objects

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

JuliaInterface data structures: Objects

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

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

JuliaInterface data structures: Objects

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

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

gap> JuliaUnbox( b );
2
```

JuliaInterface data structures: Objects

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

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

gap> JuliaUnbox( b );
2
```

Possible conversions:

- Integers

JuliaInterface data structures: Objects

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

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

gap> JuliaUnbox( b );
2
```

Possible conversions:

- Integers
- Floats

JuliaInterface data structures: Objects

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

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

gap> JuliaUnbox( b );
2
```

Possible conversions:

- Integers
- Floats
- Permutations

JuliaInterface data structures: Objects

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

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

gap> JuliaUnbox( b );
2
```

Possible conversions:

- Integers
- Floats
- Permutations
- Finite field elements

JuliaInterface data structures: Objects

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

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

gap> JuliaUnbox( b );
2
```

Possible conversions:

- Integers
- Floats
- Permutations
- Finite field elements
- Nested lists of the above to Arrays

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>  jl_sqrt := JuliaFunction( "sqrt" );
<Julia function: sqrt>
```

JuliaInterface data structures: Functions

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

```
gap> jl_sqrt := JuliaFunction( "sqrt" );
<Julia function: sqrt>

gap> jl_sqrt( 4 );
2.
```

JuliaInterface data structures: Functions

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

```
gap> jl_sqrt := JuliaFunction( "sqrt" );
<Julia function: sqrt>

gap> jl_sqrt( 4 );
2.
```

- 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> jl_sqrt := JuliaFunction( "sqrt" );
<Julia function: sqrt>

gap> jl_sqrt( 4 );
2.
```

- Julia functions can be used like GAP functions
- Input data is converted to Julia, return value is converted back to GAP

JuliaInterface data structures: Functions

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

```
gap> jl_sqrt := JuliaFunction( "sqrt" );
<Julia function: sqrt>

gap> jl_sqrt( 4 );
2.
```

- Julia functions can be used like GAP functions
- Input data is converted to Julia, return value is converted back to GAP
- Calling only possible for convertible types

JuliaInterface data structures: Functions

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

```
gap> jl_sqrt := JuliaFunction( "sqrt" );
<Julia function: sqrt>

gap> jl_sqrt( 4 );
2.
```

- Julia functions can be used like GAP functions
- Input data is converted to Julia, return value is converted back to GAP
- Calling only possible for convertible types

JuliaInterface: Julia functions as kernel modules

Using JuliaInterface, it is possible to write Julia functions and use them as GAP kernel functions:

JuliaInterface: Julia functions as kernel modules

Using JuliaInterface, it is possible to write Julia functions and use them as GAP kernel functions:

```
function orbit( self, element, generators, action )
  work_set = [ element ]
  return_set = [ element ]
  generator_length = gap_LengthPlist(generators)
  while length(work_set) != 0
    current_element = pop!(work_set)
    for current_generator_number = 1:generator_length
      current_generator = gap_ListElement(generators,
                                          current_generator_number)
      current_result = gap_CallFunc2Args(action,current_element,
                                         current_generator)
      is_in_set = false
      for  i in return_set
        if i == current_result
          is_in_set = true
          break
        end
      end
      if ! is_in_set
        push!( work_set, current_result )
        push!( return_set, current_result )
      end
    end
  end
  return return_set
end
```

JuliaInterface: Julia functions as kernel modules

Using JuliaInterface, it is possible to write Julia functions and use them as GAP kernel functions:

JuliaInterface: Julia functions as kernel modules

Using JuliaInterface, it is possible to write Julia functions and use them as GAP kernel functions:

```
gap> JuliaIncludeFile( "orbits.jl" );
gap> JuliaBindCFunction( "orbit", "orbit_jl", 3 );
```

JuliaInterface: Julia functions as kernel modules

Using JuliaInterface, it is possible to write Julia functions and use them as GAP kernel functions:

```
gap> JuliaIncludeFile( "orbits.jl" );
gap> JuliaBindCFunction( "orbit", "orbit_jl", 3 );
```

Compiled Julia functions come close to the performance of kernel functions:

JuliaInterface: Julia functions as kernel modules

Using JuliaInterface, it is possible to write Julia functions and use them as GAP kernel functions:

```
gap> JuliaIncludeFile( "orbits.jl" );
gap> JuliaBindCFunction( "orbit", "orbit_jl", 3 );
```

Compiled Julia functions come close to the performance of kernel functions:

```
gap> S := GeneratorsOfGroup( SymmetricGroup( 10000 ) );;
```

JuliaInterface: Julia functions as kernel modules

Using JuliaInterface, it is possible to write Julia functions and use them as GAP kernel functions:

```
gap> JuliaIncludeFile( "orbits.jl" );
gap> JuliaBindCFunction( "orbit", "orbit_jl", 3 );
```

Compiled Julia functions come close to the performance of kernel functions:

```
gap> S := GeneratorsOfGroup( SymmetricGroup( 10000 ) );;

gap> orbit( 1, S, OnPoints );; time;
5769
```

JuliaInterface: Julia functions as kernel modules

Using JuliaInterface, it is possible to write Julia functions and use them as GAP kernel functions:

```
gap> JuliaIncludeFile( "orbits.jl" );
gap> JuliaBindCFunction( "orbit", "orbit_jl", 3 );
```

Compiled Julia functions come close to the performance of kernel functions:

```
gap> S := GeneratorsOfGroup( SymmetricGroup( 10000 ) );;

gap> orbit( 1, S, OnPoints );; time;
5769

gap> orbit_jl( 1, S, OnPoints );; time;
84
```

JuliaInterface: Julia functions as kernel modules

Using JuliaInterface, it is possible to write Julia functions and use them as GAP kernel functions:

```
gap> JuliaIncludeFile( "orbits.jl" );
gap> JuliaBindCFunction( "orbit", "orbit_jl", 3 );
```

Compiled Julia functions come close to the performance of kernel functions:

```
gap> S := GeneratorsOfGroup( SymmetricGroup( 10000 ) );;

gap> orbit( 1, S, OnPoints );; time;
5769

gap> orbit_jl( 1, S, OnPoints );; time;
84

gap> orbit_c( 1, S, OnPoints );; time;
46
```

JuliaInterface: Next steps

Next development steps in JuliaInterface include

JuliaInterface: Next steps

Next development steps in JuliaInterface include

- stabilization of Syntax for GAP calls in Julia

JuliaInterface: Next steps

Next development steps in JuliaInterface include

- stabilization of Syntax for GAP calls in Julia
- providing sufficient amount of integration of GAP data types on the Julia side

JuliaInterface: Next steps

Next development steps in JuliaInterface include

- stabilization of Syntax for GAP calls in Julia
- providing sufficient amount of integration of GAP data types on the Julia side
- unifying GAP and Julia memory management

Coordinating garbage collection for GAP and Julia

- Both GAP and Julia use garbage collection for memory management.

Coordinating garbage collection for GAP and Julia

- Both GAP and Julia use garbage collection for memory management.
- Garbage collection: At intervals, find out which objects aren't in use anymore and throw them away.

Coordinating garbage collection for GAP and Julia

- Both GAP and Julia use garbage collection for memory management.
- Garbage collection: At intervals, find out which objects aren't in use anymore and throw them away.
- Problem: GAP and Julia have two distinct, incompatible implementations of garbage collection.

Coordinating garbage collection for GAP and Julia

- Both GAP and Julia use garbage collection for memory management.
- Garbage collection: At intervals, find out which objects aren't in use anymore and throw them away.
- Problem: GAP and Julia have two distinct, incompatible implementations of garbage collection.
- Without additional work, objects may be freed prematurely, leading to memory corruption.

How does garbage collection work?

- Garbage collection is (in principle) a simple graph algorithm.

How does garbage collection work?

- Garbage collection is (in principle) a simple graph algorithm.
- Find every object reachable from a *root*.

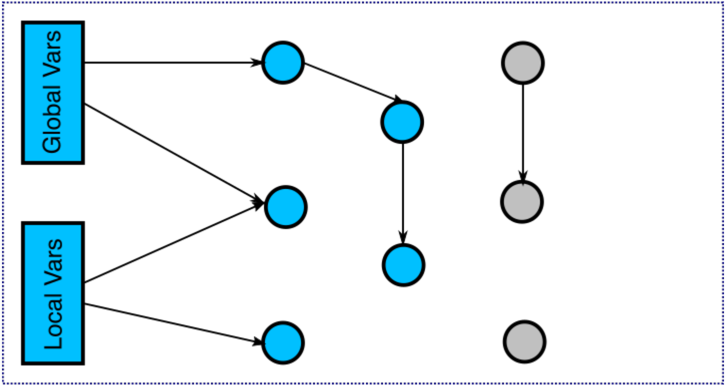
How does garbage collection work?

- Garbage collection is (in principle) a simple graph algorithm.
- Find every object reachable from a *root*.
- Dispose of objects that could not be reached.

How does garbage collection work?

- Garbage collection is (in principle) a simple graph algorithm.
- Find every object reachable from a *root*.
- Dispose of objects that could not be reached.
- Roots are:
  - Global variables (static memory).
  - Local variables and temporary values (stack, registers).

Example



GAP & Julia

- Problem: Two distinct reachability relations.

GAP & Julia

- Problem: Two distinct reachability relations.
- GAP's GC does not know the structure of Julia objects and thus which GAP objects may be reachable from Julia objects or Julia roots.

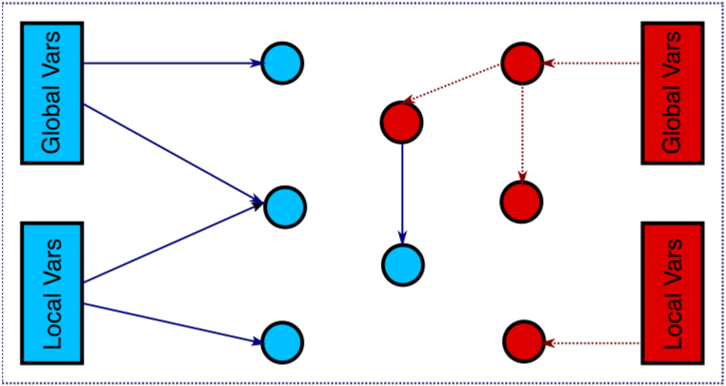
GAP & Julia

- Problem: Two distinct reachability relations.
- GAP's GC does not know the structure of Julia objects and thus which GAP objects may be reachable from Julia objects or Julia roots.
- Julia's GC does not know the structure of GAP objects and thus which Julia objects may be reachable from GAP objects or GAP roots.

GAP & Julia

- Problem: Two distinct reachability relations.
- GAP's GC does not know the structure of Julia objects and thus which GAP objects may be reachable from Julia objects or Julia roots.
- Julia's GC does not know the structure of GAP objects and thus which Julia objects may be reachable from GAP objects or GAP roots.
- Result: GAP or Julia objects may be freed prematurely.

Example



Solution A: Mutual recognition

- GAP tells Julia about any reference from a GAP to a Julia object it has. Julia stores those in a multiset.

Solution A: Mutual recognition

- GAP tells Julia about any reference from a GAP to a Julia object it has. Julia stores those in a multiset.
- Julia tells GAP about any reference from a Julia to a GAP object it has. GAP stores those in a multiset.

Solution A: Mutual recognition

- GAP tells Julia about any reference from a GAP to a Julia object it has. Julia stores those in a multiset.
- Julia tells GAP about any reference from a Julia to a GAP object it has. GAP stores those in a multiset.
- Both GAP and Julia use those multisets as additional roots for their reachability algorithms.

Solution A: Mutual recognition

- GAP tells Julia about any reference from a GAP to a Julia object it has. Julia stores those in a multiset.
- Julia tells GAP about any reference from a Julia to a GAP object it has. GAP stores those in a multiset.
- Both GAP and Julia use those multisets as additional roots for their reachability algorithms.

Advantages and disadvantages

Pros:

- Relatively straightforward to implement.
- Either GC does not need to know how the other works.
- Keeps working when GC implementations change.

Advantages and disadvantages

Pros:

- Relatively straightforward to implement.
- Either GC does not need to know how the other works.
- Keeps working when GC implementations change.

Cons:

- Avoidable inefficiencies (multiset implementation).
- Unreachable cycles that involve both GAP and Julia objects will not be reclaimed (potential memory leak).

Solution B: One GC to rule them all

- Idea: use the same GC for both GAP and Julia.

Solution B: One GC to rule them all

- Idea: use the same GC for both GAP and Julia.
- It is not possible to use Julia with the GAP GC, but:
- It is possible to use Julia's GC for GAP (with some modifications).

Solution B: One GC to rule them all

- Idea: use the same GC for both GAP and Julia.
- It is not possible to use Julia with the GAP GC, but:
- It is possible to use Julia's GC for GAP (with some modifications).
- GAP supports *almost* everything the Julia GC requires.

Solution B: One GC to rule them all

- Idea: use the same GC for both GAP and Julia.
- It is not possible to use Julia with the GAP GC, but:
- It is possible to use Julia's GC for GAP (with some modifications).
- GAP supports *almost* everything the Julia GC requires.
- Exception: root scanning.
  - Julia's GC determines local variable roots *precisely*.
  - GAP's GC assumes *conservative* scanning for local variables.

Conservative stack scanning

- Scan the entire stack and CPU registers word by word.

Conservative stack scanning

- Scan the entire stack and CPU registers word by word.
- Anything that *may* be a pointer to an object is treated like one.

Conservative stack scanning

- Scan the entire stack and CPU registers word by word.
- Anything that *may* be a pointer to an object is treated like one.
- Overly conservative in keeping objects alive.

Conservative stack scanning

- Scan the entire stack and CPU registers word by word.
- Anything that *may* be a pointer to an object is treated like one.
- Overly conservative in keeping objects alive.
- GAP needs conservative scanning, but Julia doesn't support it.

Retrofit conservative stack scanning to Julia

- Need to derive whether a machine word represents an address pointing to an object:

Retrofit conservative stack scanning to Julia

- Need to derive whether a machine word represents an address pointing to an object:
  1. Can mostly be derived from Julia's data structures
  2. For some cases this needs to be tracked in a separate data structure
- We have a proof-of-concept implementation.

Advantages and disadvantages

Pros:

- Avoids the inefficiencies of solution A.
- Handles cycles properly and avoids memory leaks.

Advantages and disadvantages

Pros:

- Avoids the inefficiencies of solution A.
- Handles cycles properly and avoids memory leaks.

Cons:

- Requires modified versions of GAP and Julia.

Goal

- Neither approach is perfect.

Goal

- Neither approach is perfect.
- Pursue solutions A and B in parallel.

Goal

- Neither approach is perfect.
- Pursue solutions A and B in parallel.
- Solution A is minimally invasive and is already used in JuliaInterface.

Goal

- Neither approach is perfect.
- Pursue solutions A and B in parallel.
- Solution A is minimally invasive and is already used in JuliaInterface.
- We have a partial prototype for solution B.

Goal

- Neither approach is perfect.
- Pursue solutions A and B in parallel.
- Solution A is minimally invasive and is already used in JuliaInterface.
- We have a partial prototype for solution B.
- Next step: Production-ready version of solution B as a minimal patch for Julia/GAP.

From GAP's point of view, Julia can provide

- new functionality
- speedup via reimplementing pieces of GAP code in Julia
- eventually an alternative to parts of GAP?

How to speed up GAP code?

Classical recommendation:

- Identify the (small) time critical parts of the code.
- Rewrite them in C. ("Move them into the GAP kernel".)

How to speed up GAP code?

Classical recommendation:

- Identify the (small) time critical parts of the code.
- Rewrite them in C. ("Move them into the GAP kernel".)

Problem: 95% of mathematicians are not C programmers!

How to speed up GAP code?

Classical recommendation:

- Identify the (small) time critical parts of the code.
- Rewrite them in C. ("Move them into the GAP kernel".)

Problem: 95% of mathematicians are not C programmers!

Now:

- Identify the time critical parts of the code.
- Rewrite them in Julia.

How to speed up GAP code?

Classical recommendation:

- Identify the (small) time critical parts of the code.
- Rewrite them in C. (“Move them into the GAP kernel”.)

Problem: 95% of mathematicians are not C programmers!

Now:

- Identify the time critical parts of the code.
- Rewrite them in Julia.

Hope to get code that is both

as fast as C code

and as flexible as GAP code.

How to speed up GAP code?

Classical recommendation:

- Identify the (small) time critical parts of the code.
- Rewrite them in C. ("Move them into the GAP kernel".)

Problem: 95% of mathematicians are not C programmers!

Now:

- Identify the time critical parts of the code.
- Rewrite them in Julia.

Hope to get code that is both
as fast as C code
and as flexible as GAP code.

(Is it easy enough for GAP programmers to take this approach?)

Which parts of GAP are suitable for this approach?

“Low level”:

few calls to GAP functions,
long nested loops over simple objects

(why not also GAP’s C code?)

Which parts of GAP are suitable for this approach?

- functions for handling permutations
  C code in GAP

Which parts of GAP are suitable for this approach?

- functions for handling permutations
  C code in GAP
- lattice functions
  LLL, OrthogonalEmbeddings

Which parts of GAP are suitable for this approach?

- functions for handling permutations
  C code in GAP
- lattice functions
  LLL, OrthogonalEmbeddings
- coset enumeration functions
  tables of small integers

Which parts of GAP are suitable for this approach?

- functions for handling permutations
  C code in GAP
- lattice functions
  LLL, OrthogonalEmbeddings
- coset enumeration functions
  tables of small integers
- character theory
  arithmetics with vectors of (algebraic) integers

Which parts of GAP are suitable for this approach?

- functions for handling permutations
  C code in GAP
- lattice functions
  LLL, OrthogonalEmbeddings
- coset enumeration functions
  tables of small integers
- character theory
  arithmetics with vectors of (algebraic) integers
- your suggestions?