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.
OSCAR: A visionary, new computer algebra system

**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.

**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

- Sebastian Gutsche - Siegen University
  - JuliaInterface/GAP.jl - Julia/GAP integration
  - Julia/polymake integration

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

- Thomas Breuer - RWTH Aachen
  - Julia in Gap
  - Representation theory

Behrends, Breuer, Gutsche, Hart  OSCAR: A visionary, new computer algebra system
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

OSCAR: A visionary, new computer algebra system
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

We are looking for projects that:

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

Behrends, Breuer, Gutsche, Hart

OSCAR: A visionary, new computer algebra system
Others involved in OSCAR

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

We are looking for projects that:

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

OSCAR: A visionary, new computer algebra system
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 !!??

We are looking for projects that:

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

Behrends, Breuer, Gutsche, Hart
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

- Heck: 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
C libraries:

- Flint - polynomials and linear algebra
C libraries:

- Flint - polynomials and linear algebra
- Antic - number field arith.
C libraries:

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

<table>
<thead>
<tr>
<th>Digits</th>
<th>Pari/GP</th>
<th>Flint (1 core)</th>
<th>Flint (4 cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.43</td>
<td>0.55</td>
<td>0.39</td>
</tr>
<tr>
<td>59</td>
<td>3.8</td>
<td>3.0</td>
<td>1.7</td>
</tr>
<tr>
<td>68</td>
<td>38</td>
<td>21</td>
<td>14</td>
</tr>
<tr>
<td>77</td>
<td>257</td>
<td>140</td>
<td>52</td>
</tr>
<tr>
<td>83</td>
<td>2200</td>
<td>1500</td>
<td>540</td>
</tr>
</tbody>
</table>
### Table: FFT timings

<table>
<thead>
<tr>
<th>Words</th>
<th>1 core</th>
<th>4 cores</th>
<th>8 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>110k</td>
<td>0.07s</td>
<td>0.05s</td>
<td>0.05s</td>
</tr>
<tr>
<td>360k</td>
<td>0.3s</td>
<td>0.1</td>
<td>0.1s</td>
</tr>
<tr>
<td>1.3m</td>
<td>1.1s</td>
<td>0.4s</td>
<td>0.3s</td>
</tr>
<tr>
<td>4.6m</td>
<td>4.5s</td>
<td>1.5s</td>
<td>1.0s</td>
</tr>
<tr>
<td>26m</td>
<td>28s</td>
<td>9s</td>
<td>6s</td>
</tr>
<tr>
<td>120m</td>
<td>140s</td>
<td>48s</td>
<td>33s</td>
</tr>
<tr>
<td>500m</td>
<td>800s</td>
<td>240s</td>
<td>150s</td>
</tr>
</tbody>
</table>
### Table: Charpoly and minpoly timings

<table>
<thead>
<tr>
<th>Op</th>
<th>Sage 6.9</th>
<th>Pari 2.7.4</th>
<th>Magma 2.21-4</th>
<th>Giac 1.2.2</th>
<th>Flint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charpoly</td>
<td>0.2s</td>
<td>0.6s</td>
<td>0.06s</td>
<td>0.06s</td>
<td>0.04s</td>
</tr>
<tr>
<td>Minpoly</td>
<td>0.07s</td>
<td>&gt;160 hrs</td>
<td>0.05s</td>
<td>0.06s</td>
<td>0.04s</td>
</tr>
</tbody>
</table>

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

Table: “Dense” Fateman multiply bench

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

4 variables
Multivariate multiplication

Table: Sparse multiply benchmark

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

5 variables
Efficient generics

Fast generics

Slow generics
Efficient 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.
- 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 $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}/p\mathbb{Z}$, $\mathbb{F}_q$, $\mathbb{Q}_p$

▶ 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 polys, sparse distributed multivariate polys, 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

Behrends, Breuer, Gutsche, Hart

OSCAR: A visionary, new computer algebra system
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

**Behrends, Breuer, Gutsche, Hart**

**OSCAR**: A visionary, new computer algebra system
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

**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

**OSCAR**: A visionary, new computer algebra system
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

**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
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
Access to Singular kernel functions and data types:

- Coefficient rings $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}/n\mathbb{Z}$, $\text{GF}(p)$, etc.
Access to Singular kernel functions and data types:

- Coefficient rings $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}/n\mathbb{Z}$, $\text{GF}(p)$, etc.
- Polynomials, ideals, modules, matrices, etc.
Access to Singular kernel functions and data types:

- Coefficient rings \( \mathbb{Z}, \mathbb{Q}, \mathbb{Z}/n\mathbb{Z}, \text{GF}(p) \), etc.
- Polynomials, ideals, modules, matrices, etc.
- Groebner basis, resolutions, syzygies
Access to Singular kernel functions and data types:

- Coefficient rings \( \mathbb{Z}, \mathbb{Q}, \mathbb{Z}/n\mathbb{Z}, \text{GF}(p) \), 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
Access to Singular kernel functions and data types:

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

GAP ↔ Julia
GAP package JuliaInterface

GAP $\leftrightarrow$ Julia

JuliaInterface provides
GAP package JuliaInterface

GAP $\leftrightarrow$ Julia

JuliaInterface provides

- Conversions of GAP to Julia data and vice versa
GAP package JuliaInterface

GAP $\leftrightarrow$ 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

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

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

Possible conversions:

- Integers
- Floats
- Permutations
- Finite field elements
- Nested lists of the above to Arrays
JuliaInterface data structures: Objects

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

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

Possible conversions:
- Integers
- Floats
- Permutations
- Finite field elements
- Nested lists of the above to Arrays
JuliaInterface data structures: Objects

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

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

gap> JuliaUnbox( b );
2
```
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 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 contains GAP data structures that can hold pointers to Julia objects:

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

gap> JuliaUnbox( b );
2
```

Possible conversions:

- Integers
- Floats
- Permutations
- Finite field elements
JuliaInterface contains GAP data structures that can hold pointers to Julia objects:

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

```gap
jl_sqrt := JuliaFunction( "sqrt" );
<Julia function: sqrt>
```
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 provides the possibility to call Julia functions by converting GAP objects:

\[ \text{gap} > \quad \text{jl}\_\text{sqrt} := \text{JuliaFunction}( \text{"sqrt"} ); \]
\[ <\text{Julia function: sqrt}> \]

\[ \text{gap} > \quad \text{jl}\_\text{sqrt}( 4 ); \]
\[ 2. \]

- Julia functions can be used like GAP functions
JuliaInterface provides the possibility to call Julia functions by converting GAP objects:

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

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

```plaintext
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:

```gap
JuliaIncludeFile( "orbits.jl" );
JuliaBindCFunction( "orbit", "orbit_jl", 3 );
Compiled Julia functions come close to the performance of kernel functions:
S := GeneratorsOfGroup( SymmetricGroup( 10000 ) );
orbit( 1, S, OnPoints );; time;
5769
orbit_jl( 1, S, OnPoints );; time;
84
orbit_c( 1, S, OnPoints );; time;
46
```
Using JuliaInterface, it is possible to write Julia functions and use them as GAP kernel functions:

```gap
gap> JuliaIncludeFile( "orbits.jl" );
gap> JuliaBindCFunction( "orbit", "orbit_jl", 3 );
```
Using JuliaInterface, it is possible to write Julia functions and use them as GAP kernel functions:

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

Compiled Julia functions come close to the performance of kernel functions:
Using JuliaInterface, it is possible to write Julia functions and use them as GAP kernel functions:

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

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

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

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

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

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

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

gap> orbit( 1, S, OnPoints );; time;
5769
```
Using JuliaInterface, it is possible to write Julia functions and use them as GAP kernel functions:

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

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

```gap
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
```
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
Next development steps in JuliaInterface include

- stabilization of Syntax for GAP calls in Julia
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
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
Both GAP and Julia use garbage collection for memory management.
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.
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.
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.
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

Global Vars

Local Vars

OSCAR: A visionary, new computer algebra system
Problem: Two distinct reachability relations.
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.
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.
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

Behrends, Breuer, Gutsche, Hart

OSCAR: A visionary, new computer algebra system
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.
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.
Neither approach is perfect.
Goal

- Neither approach is perfect.
- Pursue solutions A and B in parallel.
Neither approach is perfect.
- Pursue solutions A and B in parallel.
- Solution A is minimally invasive and is already used in JuliaInterface.
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.
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.
Integration of GAP and Julia – Ideas and Experiments

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?