Interfacing computer algebra and formal proof systems

Saarland University

Master's Thesis

submitted in partial fulfillment of the requirements
for the degree of Master of Science

by

Cedric Holle

July 2024

Supervisor: Prof. Dr. Laurent Bartholdi

Contents

1 Introduction

Computer algebra systems are powerful tools with a significant impact on modern mathematics. They have grown considerably over the past few decades and helped solving many conjectures. For instance, OSCAR has been used to prove that the Cox ring of blowups of 7 points in $\mathbb{P}^3$ is quadratically generated [1]. Other notable examples include the proof of the Kepler conjecture [12] and the proof of the Hecking-Keel-Tevelev conjecture on degenerations of Grassmannians [7].

Computer algebra systems like GAP [10] have been evolving for decades, becoming increasingly complex over time. The official GitHub repositorium of GAP has 59 contributors with more than 10,000 commits and this does not even include the numerous packages. Historically, most computer algebra systems specialized in specific subfields of mathematics. However, there is now considerable effort to connect these systems. For example, SageMath and OSCAR are large projects built upon multiple substantial systems such as GAP and polymake as well as numerous low-level libraries such as Flint. By combining these systems, SageMath and OSCAR exceed the mathematical capabilities of their individual components. However, this also significantly increases their complexity, raising the question how much we can trust the computer generated results.

Simultaneously, formal mathematics is a rapidly growing field. Interactive Theorem Prover allow users to model mathematics effectively. Systems such as Coq [2] and Isabelle [14] have existed for over thirty years and were used to verify some proofs of interesting problems such as the four color theorem. More recently, there has been much effort to develop extensive mathematical libraries. Mathlib [5], a Lean library, now consists of over 1.5 million lines of code, 80,000 definitions and 150,000 theorems with more than 300 contributors. After seven years of development, its growth shows no signs of slowing down.

While formal systems benefit greatly from advancements in automation, they do not autonomously solve theorems. Users must provide step-by-step instructions for constructing proofs, with the system acting as a verifier to ensure correctness. It offers a clear overview of remaining tasks and available hypotheses at each step. However, the mathematician must devise the proof ideas and guide the system.

This thesis aims to integrate both systems. This allows us to transfer the advantages of the individual systems to the other system. The computer algebra system provides powerful search heuristics to find solutions for the formal system, while the verifier can validate calculations, ensuring correctness. In this thesis, the focus is on using the computer algebra system to find solutions and automatically validate them. This approach is particularly useful in scenarios where finding a solution is much more complex or computationally expensive than validating it. Any potential bugs in the computer algebra system will be detected during verification, ensuring safety.

I implement an interface for the proof system Lean and the computer algebra system OSCAR. By adding support for the versatile data format MaRDI to Lean, the tool becomes highly general and easy to extend. The format is designed to enable serialization of various mathematical data and its modular structure ensures the interface's extensibility. Incorporating other computer algebra systems should be straightforward if they support the format.

I present three specific tactics to demonstrate the capabilities of the interface:

- First, I show a tactic to automatically invert a matrix. This is an easy example and a good entry point to observe the general procedure.
- In the second example, I present a tactic to validate the membership in permutation groups. This builds upon factorization of group elements and can for example be used to solve the Rubik's cube. Factorization is a typical problems which is hard to solve but easy to verify.
- In the last example, we use Knuth-Bendix algorithm to solve equations in finitely presented groups. This is an undecidable problem. With good heuristics however, it is possible to tackle most practical problems. Some proofs might involve too many steps to do effectively by hand. Hence, it is very helpful to have this process automated. We had to slightly manipulate the computer algebra system to print out a more extensive trace of the calculations involved.

The results show that it is possible to validate some computations of computer algebra systems at relatively low cost in terms of compute power and implementation time. This increases the safety and correctness and therefore the trust we put into these complex architectures. Additionally, we can use the computer algebra system to enhance the automation of the formal system and access the vast amount of work done for computer algebra systems.

1.1 Prior work

There are many existing interfaces between computer algebra systems and proof systems, underscoring the interest and potential of such integrations. The tactic polyrith in Mathlib, for example, uses SageMath for polynomial computations and integrates the result into the proof. However, most similar programs known to me focus on specific use cases and involve multiple abstract syntax trees to convert data. This induces considerable effort for each new application. The goal is therefore to make the interface as general and easy extensible as possible. We are building on the shoulders of giants like GAP and polymake and I want to give the users access to the whole systems and not only to selected parts.

There are also various interfaces between formal systems and other tools such as automated theorem provers. The possible benefits are quite similar. A prominent example is sledgehammer, a powerful tactic in Isabelle. Sledgehammer translates the theorem into first order logic and passes it to automated theorem provers such as Vampire, E and Z3. If one of the automated theorem provers successfully solves the problem, it returns the lemmas used in the proof. This is often enough information to repeat the proof in Isabelle.

1.2 Sample session

This is an example for how a typical Lean file could look like. First we should import Mathlib and the interface.

```
import Oscar
import Mathlib
```

We start a server and test if it works. `#echo` will send the data to the server which should send it back. This is practical for debugging serialization tasks. `#start_server` can be skipped and the first command trying to access the server will start it. The server is here is simple julia instance waiting for commands.

```
#start_server
#echo (5 : ℕ)
#echo (6 : Int)
```

We can save and read MaRDI files.

```
def word : FreeGroup (Fin 4) := by load_file "free_group_word"
#writeMrdi word to "free_group_word"
#readMrdi FreeGroup (Fin 4) from "free_group_word"
```

Let us next define a matrix `A` and use the tactic `matrix_inverse` to invert it. The tactic sends `A` to the server, where it will be inverted and send back.

```
def A : Matrix (Fin 3) (Fin 3) ℚ := !![3, 0, 4; 5, 10, 6; 1, 2, 3]
def A_inv : Matrix (Fin 3) (Fin 3) ℚ := by matrix_inverse A
```

It is not difficult to prove that `A_inv` has the desired properties.

```
example : A * A_inv = 1 := by
  simp [A, A_inv]
  ext i j
  fin_cases i
  all_goals fin_cases j
  any_goals norm_num [_root_.mkRat, Rat.normalize]
  any_goals rfl
```

For the next example, let us take a look at the group `G` of transformations of Rubik's magic cube. The six permutations correspond to a rotation of the Rubik's cube around a face of the cube and together they generate the group.

```
def top    : Equiv.Perm (Fin 48) := c[ 0,  2,  7,  5] * c[ 1,  4,  6,  3]
  * c[ 8, 32, 24, 16] * c[ 9, 33, 25, 17] * c[10, 34, 26, 18]
def left   : Equiv.Perm (Fin 48) := c[ 8, 10, 15, 13] * c[ 9, 12, 14, 11]
  * c[ 0, 16, 40, 39] * c[ 3, 19, 43, 36] * c[ 5, 21, 45, 34]
def front  : Equiv.Perm (Fin 48) := c[16, 18, 23, 21] * c[17, 20, 22, 19]
  * c[ 5, 24, 42, 15] * c[ 6, 27, 41, 12] * c[ 7, 29, 40, 10]
def right  : Equiv.Perm (Fin 48) := c[24, 26, 31, 29] * c[25, 28, 30, 27]
  * c[ 2, 37, 42, 18] * c[ 4, 35, 44, 20] * c[ 7, 32, 47, 23]
```

```
def rear   : Equiv.Perm (Fin 48) := c[32, 34, 39, 37] * c[33, 36, 38, 35]
  * c[ 2,  8, 45, 31] * c[ 1, 11, 46, 28] * c[ 0, 13, 47, 26]
def bottom : Equiv.Perm (Fin 48) := c[40, 42, 47, 45] * c[41, 44, 46, 43]
  * c[13, 21, 29, 37] * c[14, 22, 30, 38] * c[15, 23, 31, 39]
def G := Group.closure {top, left, front, right, rear, bottom}
```

We can verify that an element `g` is in `G` and therefore corresponds to a transformation of the Rubik's cube that can be solved. This may take some minutes, so we should increase the default maximum time for tactics. `perm_group_membership` uses OSCAR to factorize `g` and automatically completes the proof.

```
set_option maxRecDepth   1000000000000000000000
set_option maxHeartbeats 1000000000000000000000


def g : Equiv.Perm (Fin 48) := c[16, 18] * c[10, 7] * c[5, 24] * c[6, 27] * c[17, 20]


example : g ∈ G := by
  unfold G
  perm_group_membership
```

In the last example we want to show that $\langle a, b \mid [a,b] \cdot a^{-1} = 1,\, [b,a] \cdot b^{-1} = 1 \rangle$ is trivial. Let us first define the group.

```
@[reducible]
def F := FreeGroup (Fin 2)
@[reducible]
def a : F := FreeGroup.mk [(0, true)]
@[reducible]
def b : F := FreeGroup.mk [(1, true)]
@[reducible]
def rels_list := [a⁻¹ * b⁻¹ * a * b * a⁻¹, b⁻¹ * a⁻¹ * b * a * b⁻¹]
@[reducible]
def rels := List.toSet rels_list
@[reducible]
def G₂ := PresentedGroup rels
```

The tactic `kbmag` sends the problem to OSCAR which delegates it to GAP which loads the GAP package KBMAG and runs Knuth-Bendix algorithm to find a confluent rewriting system. We can manipulate the algorithm to print enough information to reconstruct the calculation in Lean and integrate it into a proof.

```
theorem g_triv : ∀ (x : G₂), x = 1 := by
  kbmag (1 : G₂)
```

1.3 Installation instructions

If you want to use the interface, you need to install all dependencies first. Note that this is only tested on Linux.

You can find installation instructions for Lean at https://leanprover-community.github.io/get_started.html.

For the installation of Julia and OSCAR, I would refer to OSCAR's webpage https://www.oscar-system.org/install/.

Now that all requirements are fulfilled, we can get to the installation of the interface. You can find it at https://github.com/todbeibrot/Lean-Oscar. Using git, you can clone the repository with the command

```
git clone https://github.com/todbeibrot/Lean-Oscar.git
```

Next, open the folder with

```
cd Lean-Oscar
```

and run

```
lean exe cache get
```

This will manage all the Lean libraries we are working with. Finish the installation with

```
lake build
```

If you want to see a sample session or check if everything is correctly installed, open the folder `Lean-Oscar` with an appropriate user interface and take a look at `test.lean`. If you use VS Code, you can open the folder with

```
code .
```

and find the file in the Explorer on the left side. The file `test.lean` should compile without any errors. Though, it might take a few minutes.

2 Lean

Lean is a functional programming language based on Calculus of Inductive Constructions and dependent type theory. This allows to effectively do mathematics on a computer. Lean is mostly used as an interactive theorem prover. This means that one can prove propositions in Lean by giving the computer high level instructions (tactics) on how to construct the proof and the system will use them to build an expression. This process can be done step by step. At each step, the user can see the goals which indicate what is left to proof and the local context which show the currently available hypotheses. Like most formal systems it excels at safety and correctness, making it a good proof checker. Throughout the thesis, I work with Lean 4, the latest version. The language has many useful features such as type classes, extensible syntax and multithreading. The safety of Lean comes from a small kernel, minimizing the parts which have to be trusted.

There are some up and downsides for doing math in a formal way. The promised correctness is obviously a big advantage. Another benefit comes from the joint organization. Most projects in Lean are using the same libraries, in particular Mathlib. This ensures that everyone uses the same definitions and makes it easy to build on other peoples work.

A key factor is the grade of automation. The potential here seems limitless. I will show some examples in a later chapter where nontrivial proofs collapse to a single line. But this is rarely the case. The guaranteed correctness means that we have to prove many trivial facts which we could skip in a paper proof. Sometimes a proof can feel like a big fight against the syntax until the instructions are accepted. This slows down and might be the biggest downside of the formal setting. In my experience, the formal setting consumes more time compared to informal math for similar work. Though, that might change with the progress of automation.

2.1 Sample

Let us prove in Lean that addition of natural numbers is commutative.

```
theorem Nat.add_comm (n m : ℕ) : n + m = m + n := by
  induction m with
  | zero     => rewrite [Nat.add_zero, Nat.zero_add]
                rfl
  | succ k h => rw [Nat.add_succ, Nat.succ_add]
                rw [h]
```

You can find the theorem in Lean with a slightly more refined proof. The theorem is called `add_comm` in the namespace `Nat`. Between the parentheses after the name we introduce two variables of type ℕ. ℕ is an abbreviation for `Nat`. After the colon comes the statement of the theorem.

We start the proof with `:=` and enter tactic mode with `by`. A user interface should now show us the current proof state:

```
n m : ℕ
⊢ n + m = m + n
```

We have two natural numbers `n` and `m` given, and we need to prove `n + m = m + n`. ⊢ signals the start of the current goal.

We want to prove the theorem by induction on `m`, so we start with the tactic `induction`. Afterwards, we have two goals to solve, depending on `m` being `zero` or the successor of another natural number `k`.

The tactic state of the first goal is

```
n : ℕ
⊢ n + zero = zero + n
```

We can solve this goal by using `Nat.add_zero` and `Nat.zero_add`. They state that `n + 0 = n` and `0 + n = n`. Similar to informal math, we often build on previously proven theorems. `rewrite` is a tactic that uses a proven theorem consisting of an equation and replaces appearances of the left hand side in the goal by the right hand sight of the equation. This leaves us with

```
n : ℕ
⊢ n = n
```

The tactic `rfl` solves such trivial goals by using reflexivity.

The second goal is

```
n k : ℕ
h : n + k = k + n
⊢ n + succ k = succ k + n
```

`h` is the induction hypotheses. Here we need `Nat.add_succ` and `Nat.succ_add` to move `succ` to the outside so that we can use `h`. `rw` is short for `rewrite`. The only difference is that it tries `rfl` afterwards.

After the first rewrite, the tactic state is

```
n k : ℕ
h : n + k = k + n
⊢ succ (n + k) = succ (k + n)
```

After rewriting `h`, both sides of the equation will be the same and `rfl` will finish the proof.

2.2 Meta programming

Expressions in Lean are formally terms of type `Expr`. They build an abstract syntax tree for Lean programs. Expressions can be viewed as programs which can be evaluated to get its value. When proving a theorem, we want to construct an expression which has the desired type when evaluated. In the interactive setting this is done in a tactic block step by step or rather tactic by tactic. Tactics are programs which help us to construct and manipulate expressions. The final result of a tactic block will be sent to the kernel and checked for correctness. The process of writing tactics is often called meta programming because the programs build other programs.

Only small parts of Lean such as the kernel are written in C. The majority is written in Lean itself. The whole meta programming framework is written in Lean. This makes it easy to extend the language.

2.3 Type Classes

Lean has powerful type classes. Type classes are structures that are inferred by a backtracking search. They are useful for algebraic hierarchies and pleasant notations. For example, Lean has a class for addition with the notation `+` and instances of the class for natural numbers, integers, reals, vectors, functions, etc.

2.4 Mathlib

Mathlib [5] is the standard library for mathematical projects in Lean. It is a community project with more than 300 contributors. It consists of over 1.5 million lines of code, 80,000 definitions, and 150,000 theorems, and it is growing rapidly. There is a broad goal to formalize all important undergraduate-level mathematics. However, the library also contains some higher-level mathematics, such as perfectoid spaces [4].

3 OSCAR

OSCAR is a computer algebra system. It stands for Open Source Computer Algebra Research. "OSCAR features functions for groups, rings, and fields as well as linear and commutative algebra, number theory, algebraic and polyhedral geometry, and more." [15]

OSCAR is written in Julia. The language provides good performance in terms of speed and allows good integration of other languages. This makes it easy to incorporate existing computer algebra systems [3]. OSCAR builds upon the computer algebra systems GAP, Singular, polymake and ANTIC. This allows it to offer a rich variety of features. Furthermore it allows to effectively combine those computer algebra systems.

ANTIC consists of the Julia packages AbstractAlgebra, Hecke and Nemo, which are for computational abstract algebra such as generic polynomial rings, matrix spaces, finite fields, number fields etc. [9]. GAP stands for Groups, Algorithms, Programming. It handles discrete algebra with focus on computational group theory [10]. polymake is made for research in polyhedral geometry [11]. "Singular is a computer algebra system for polynomial computations, with special emphasis on commutative and non-commutative algebra, algebraic geometry, and singularity theory." [8] Diagram 1 shows a detailed overview of the connections.

Bringing all the building blocks together in one system is a complex task. All corner stones are big projects by themselves and rely on lots of packages, some of them written decades ago. In such a dynamic environment, verification can be extremely beneficial.

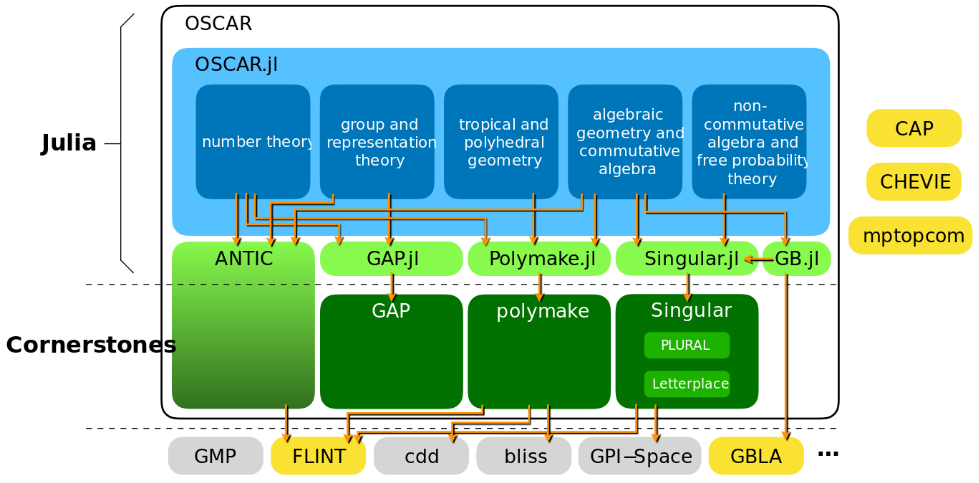

Figure 1: OSCAR-Organigramm [15]

3.1 Session example

This is an example for an interactive Julia session using its REPL (read-eval-print loop) and OSCAR. The REPL can be started from the command line with the command "julia". We take a look at the group of tranformations of the Rubik's Cube. The example is a timeless classic to show basic features of GAP, adapted to OSCAR's syntax. All used functions relay to GAP under the hood. The original can be found at GAP's homepage [10]. The faces of the Rubik's cube will be numbered as in Figure 2.

```
                             +--------------+
                             |              |
                             |  1    2    3 |
                             |              |
                             |  4   top   5 |
                             |              |
                             |  6    7    8 |
                             |              |
              +--------------+--------------+--------------+--------------+
              |              |              |              |              |
              |  9   10   11 | 17   18   19 | 25   26   27 | 33   34   35 |
              |              |              |              |              |
              | 12  left  13 | 20 front  21 | 28 right  29 | 36  rear  37 |
              |              |              |              |              |
              | 14   15   16 | 22   23   24 | 30   31   32 | 38   39   40 |
              |              |              |              |              |
              +--------------+--------------+--------------+--------------+
                             |              |
                             | 41   42   43 |
                             |              |
                             | 44 bottom 45 |
                             |              |
                             | 46   47   48 |
                             |              |
                             +--------------+
```

Figure 2: six faces of Rubik's Cube with numbering [10]

The group of transformations is generated by the six permutations corresponding to the rotation of a face clockwise.

```
julia> cube = permutation_group(48, [
  cperm([ 1, 3, 8, 6], [ 2, 5, 7, 4], [ 9,33,25,17], [10,34,26,18], [11,35,27,19]),
  cperm([ 9,11,16,14], [10,13,15,12], [ 1,17,41,40], [ 4,20,44,37], [ 6,22,46,35]),
  cperm([17,19,24,22], [18,21,23,20], [ 6,25,43,16], [ 7,28,42,13], [ 8,30,41,11]),
  cperm([25,27,32,30], [26,29,31,28], [ 3,38,43,19], [ 5,36,45,21], [ 8,33,48,24]),
  cperm([33,35,40,38], [34,37,39,36], [ 3, 9,46,32], [ 2,12,47,29], [ 1,14,48,27]),
  cperm([41,43,48,46], [42,45,47,44], [14,22,30,38], [15,23,31,39], [16,24,32,40])
])
```

It is a group of degree 48 and size

```
julia> size = order(cube)
43252003274489856000
```

We can factorize the heavy number to get a better illustration.

```
julia> factor(size)
1 * 5^3 * 7^2 * 2^27 * 11 * 3^14
```

The two orbits of the group are computed as follows. The corners are in the first set, the edges in the second.

```
julia> cube_orbits = orbits(cube)
2-element Vector{Oscar.GSetByElements{PermGroup}}:
 G-set of cube with seeds [1, 3, 17, 14, 8, 38, 9, 41, 19, 48, 22, 6, 30,
   33, 43, 11, 46, 40, 24, 27, 25, 35, 16, 32]
 G-set of cube with seeds [2, 5, 12, 7, 36, 10, 47, 4, 28, 45, 34, 13, 29,
   44, 20, 42, 26, 21, 37, 15, 31, 18, 23, 39]
```

Let us take a look at the permutation group of the 24 corner faces.

```
julia> cube1 = image(action_homomorphism(cube_orbits[1]))[1]
Permutation group of degree 24
```

Note that the group acts on the set $\{1, \cdots, 24\}$. It is a group with 88179840 elements.

```
julia> order(cube1)
88179840
```

We can not turn a corner clockwise

```
julia> cperm([1, 7, 22]) in cube1
false
```

without turning another corner counterclockwise.

```
julia> cperm([1, 7, 22],[2, 20, 14]) in cube1
true
```

The edges behave quite similar.

```
julia> cube2 = image(action_homomorphism(cube_orbits[2]))[1]
Permutation group of degree 24
```

`cube2` corresponds to the permutation group of the 24 edges with 980995276800 elements.

```
julia> order(cube2)
980995276800
```

And again, we can not flip an edge

```
julia> cperm([1, 11]) in cube2
false
```

without flipping another edge.

```
julia> cperm([1, 11], [2, 17]) in cube2
true
```

The last two statements could likely be verified in Lean with reasonable effort.

The size of `cube` is half of the product of the size of `cube1` and the size of `cube2`. So it must be a subgroup of index 2 of the direct product of `cube1` and `cube2`.

```
julia> order(cube)
43252003274489856000
```

```
julia> order(cube1) * order(cube2)
86504006548979712000
```

Corners and edges are not independent in `cube`. If we want to swap two corners (edges), we have to swap two edges (corners).

```
julia> cperm([17, 19], [11, 8], [6, 25]) in cube
false
```

```
julia> cperm([7, 28], [18, 21]) in cube
false
```

```
julia> cperm([17, 19], [11, 8], [6, 25], [7, 28], [18, 21]) in cube
true
```

We verify the last statement in section 6.2.1.

The center of a group is the set of commuting elements and builds a normal subgroup. The center of `cube` has one nontrivial element which corresponds to flipping all edges.

```
julia> z = center(cube)[1]
Permutation group of degree 48 and order 2
```

We define an epimorphism from a free group to cube mapping generators to generators. The preimage of an element in cube with respect to the epimorphism is computed as a word of the generators of the free group with inverses. Following the epimorphism back to `cube`, we get a factorization of the original element. The factorization can be used to turn a transformed Rubik's cube back into its solved form.

```
julia> epi = epimorphism_from_free_group(cube)
Group homomorphism
  from free group of rank 6
  to cube
```

We can solve the Rubik's cube in 94 steps if all edges are flipped. This is not the shortest solution but still good enough to apply in practice.

```
julia> z1_preimage = preimage(epi, gens(z)[1])
x3*x4*x1*x4^-1*x3^-1*x2^-1*x5^-1*x1^-1*x5*x2^2*x3^-1*x2^-1*x3*x1*x3*x1^-1*
x3^2*x2*x3*x2^-1*x1^-1*x2^-1*x1*x2*x1^-1*x2*(x3*x1*x3^-1*x1^-1*x2^-1)^2*
x1^-1*x2*x1*x2^-1*x1*x5^-1*x1*x5*x3^-1*x2*x3*x2*x1*x2*x1^-1*x2^-1*x3*x1^-1*
x3^-2*x2^-1*x3*x6*x3^-1*x6^-1*x1^-2*x3^-1*x1*x2^-1*x3*x1^-2*x4^-1*x3^-1*x4*
x6*x4*x6^-1*x3^-1*x5^-1*x4*x5*x3*x6*x4^-1*x3^-1*x6^-1*x4^-2*x1^-2*x2^-1*
x6^-1*x4^-1*x5^-1
```

```
julia> length(z1_preimage)
94
```

The same can be done for arbitrary elements.

```
julia> g1 = preimage(epi, cperm([17, 19], [11, 8], [6, 25], [7, 28], [18, 21]))
x2^-1*x1^-1*x2*x3*x4*x1*x4^-1*x3^-1*x2*x1*x3*x1^-1*x3^-1*x2^-1*x1^2*x3*x1*
x2*x1*x2^-1*x3^-1*x2*x1^-1*x2^-1*x3*x1^-1*x3^-1*x2*x1*x2^-1*x1*x2*x1^-2*
x2^-1*x3*(x1*x4*x1^-1*x4^-1)^2*x3^-1*x1*x2*x3^-1*x2^-1*x3*x2^-1*x1^-1*x2*
x1^-2*x3*x1*(x3^-1*x2^-1)^2*x2^-1*x3*x2*x5^-1*x1*x5*x2*x1^-1*x5^-1*x1^-1*
x5*x2*x6*x3^-1*x6^-1

julia> length(g1)
77
```

Here we let GAP generate a random element and complete the example by showing that the factorization is correct.

```
julia> r = rand(cube)
(1,38,3,35,32,27,9,48,33)(2,44,28,26,23,45,36,12,39,34,15,21,5,42,31,29,37,47)
(4,20,7,10,13,18)(8,46,25,40,19,14)(16,24)(22,30)(41,43)

julia> r_preimage = preimage(epi, r)
x3*x4*x1*x4^-1*x3^-1*x2^-1*x5^-1*x1^-1*x5*x2*x1^2*x3*x1*x2*x1*x2^-1*x3^-1*
x2*x1^-1*x2^-1*x3*x1^-1*x3^-1*x1*x3^-1*x1^-1*x2^-1*x1*x2*x3*x1^-1*x2*x1*x3*
x1^-1*x3^-1*(x2^-1*x1)^2*x2*x3^-1*x2*x3*x2^-2*x1*x2*x3*x1*(x3^-1*x2^-1)^2*
x3*x2*x1^-1*x5^-1*(x1^-1*x5)^2*x2*x5^-1*x6*x3^-2*x6^-1*x3^-1*x1*x2^-1*x3*
x2^-1*x3^-1*x6*x4*x6^-1*x3^-1*x4^-2*x1^-2*x2^-1*x5^-2

julia> length(r_preimage)
86

julia> image(epi, r_preimage) == r
true
```

4 Interfacing both systems

The mission of this thesis is to interface OSCAR and Lean and making the interface as general and extensible as possible. The code will be described in section 5. The interface has to be able to transmit tasks as well as calculated solutions. Both may contain complex mathematical objects. Therefore, we need to consider how to serialize (convert data to strings) mathematical objects. To achieve this, we will use the MaRDI file format [16].

4.1 MaRDI

MaRDI is a JSON based file format for mathematical objects. It follows the JSON schema shown in Figure 3 that stipulates the structure. It is developed as part of the Mathematics Research Data Initiative (MaRDI) [6]. This format can be used to save and load diverse mathematical data. This enables saving results of algebraic computations and loading them again instead of repeating the computation. It also extends the possibilities for communication with other systems which support the file format. Serialization of mathematical data in OSCAR uses the MaRDI file format. So we only have to develop support for it in Lean. The file extension is `.mrdi`.

4.1.1 Structure

When serialized, each object needs a type that tells us what kind of data we are working with. Namespace, data, references and id are optional.

The namespace(`_ns`) should contain information about the defining system, such as "Oscar", and additional details like version and a URL linking to the system's homepage.

"The actual data is stored in the property with the same name." [6] The structure for data (`_data`) is recursively defined. It can be a string, an array consisting of data or an object where the properties are again data.

The only required field `type` can be represented by a simple string or an object with a `name` and parameters(`params`). The name should be a string and the parameters have the same structure as data.

Often the serialized data only makes sense in the context of other objects. These are stored as references(`_refs`) with a universally unique identifier (UUID) and use the same structure.

```json
{
  "$defs": {
    "data": {
      "oneOf": [
        {
          "type": "string"
        },
        {
          "not": {
            "required": [
              "_ns"
            ]
          },
          "patternProperties": {
            "^[a-zA-Z0-9_]*": {
              "$ref": "#/$defs/data"
            }
          },
          "type": "object"
        },
        {
          "items": {
            "$ref": "#/$defs/data"
          },
          "type": "array"
        },
        {
          "$ref": "https://polymake.org/schemas/data.json"
        }
      ]
    }
  },
  "$id": "https://oscar-system.org/schemas/mrdi.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "properties": {
    "_ns": {
      "type": "object"
    },
    "_refs": {
      "patternProperties": {
        "^[0-9a-fA-F]{8}-([0-9a-fA-F]{4}-){3}[0-9a-fA-F]{12}$": {
          "$ref": "#"
        }
      },
      "type": "object"
    },
    "_type": {
      "oneOf": [
        {
          "type": "string"
        },
        {
          "properties": {
            "name": {
              "type": "string"
            },
            "params": {
              "$ref": "#/$defs/data"
            }
          },
          "type": "object"
        }
      ]
    },
    "data": {
      "$ref": "#/$defs/data"
    }
  },
  "required": [
    "_type"
  ],
  "type": "object"
}
```

Figure 3: JSON schema [17] for the MaRDI file format.

4.1.2 Example

Let us take a look at an actual file for better visualization. We serialize the cycle $(2, 3, 4)$ as element of the symmetric group $S(5)$. The resulting MaRDI can be seen in Figure 4. The property `_ns` contains the namespace, in this case "Oscar", with a URL to the homepage and the exact version. The `_type` indicates that the cycle is element of a permutation group. The parameters (`params`) contain a UUID which refers to the permutation group. The group can be found in the `_refs` property under the appropriate UUID. It is again a valid MaRDI object. We can see that $S(5)$ is saved as a permutation group of degree 5 with two generators $(1, 2, 3, 4, 5)$ and $(1, 2)$. The `data` field specifies the saved element. It is stored as an array where the permutation maps i to the ith element of the array. The array is pruned as long as it ends like the identity map.

```json
{
  "_ns": {
    "Oscar": [
      "https://github.com/oscar-system/Oscar.jl",
      "1.0.0-DEV-fbd34b88fbedbbcb729a1e2ea5037b1860cda204"
    ]
  },
  "_type": {
    "name": "PermGroupElem",
    "params": "d98e9fdd-9834-4e21-9cea-f37cf9524ce2"
  },
  "data": ["1","3","4","2"],
  "_refs": {
    "d98e9fdd-9834-4e21-9cea-f37cf9524ce2": {
      "_type": "PermGroup",
      "data": {
        "degree": "5",
        "gens": [
          ["2","3","4","5","1"],
          ["2","1"]
        ]
      }
    }
  }
}
```

Figure 4: JSON description of the cycle $(2, 3, 4)$ in $S(5)$

5 Lean - Oscar

This work comes with code and I will describe some of it in more detail in this section. The goal is to make it as easy as possible for the reader to use the implemented tools and extend them to use more computer algebra features. So I will focus on the parts which are important for new applications.

The repository is separated into two bigger parts. The `Mrdi` folder is mostly about handling serializing and deserializing. This includes building a MaRDI object of its individual parts and disassembling it. We also need to support basic behaviors like printing, parsing and converting to Lean expressions.

In the `Oscar` folder we can find the implementation for the connection to OSCAR and the tactics for the actual applications.

Examples for applications and implemented tools can be found in the file `test.lean`. When going through the code, I would recommend to start here to get an overview and see some results.

5.1 Mrdi

Let us go through all files in the `Mrdi` folder, starting with the basics and build on it.

In `UUID.lean`, I defined UUIDs and added basic features. They are needed for the references of MaRDI objects.

The JSON implementation in Mathlib uses red-black trees. Since we want to build on it, MaRDI should be defined in a similar way. In `RBNode.lean` are some features for red-black trees which come in handy when adding functions for MaRDI.

When writing tactics, we need to convert some objects into Lean expressions. These instances can be found in `ToExpr.lean`.

The definition of a MaRDI object in Lean can be found in `Basic.lean`. It is an inductive type with only one constructor. Therefore, we can construct a MaRDI by giving all the components. The file contains also a lot of definitions to get specific parts of an MaRDI. We also define the single components here and add some auxiliary functions for building and disassembling them. Everything is basically JSON with some structure. So we should have features for getting the underlying JSON and converting JSON to MaRDI, assuming it has the correct structure. This enables the transfer of JSON features in Mathlib to MaRDI, in particular parsing and printing. We also want to convert some MaRDI into expressions. Thanks to the work in previous files, this can now be derived automatically.

The next step is to convert MaRDI object to its corresponding type in Lean. To achieve this, we define a class `FromMrdi` in the file with the same name and fill the file with lots of instances. I added instances for basic types like strings, integers, natural numbers, booleans and rationals. There are also instances for common types like lists, vectors, tuples and arrays. Additionally, I implemented support for some more abstract types like polynomials, permutations, words in a finite free group and matrices.

So far, I did not implement automatic detection of the type of the MaRDI. In practice, we have to infer the expected type beforehand. But if we know the type, the only important component left in the MaRDI is the data. So it is enough to implement an instance for `FromMrdiData` and `FromMrdi` can be derived from it.

This might be a part which users of the repository might extend a lot. So let us add some examples.

We want to load the Lean Vector $[1, 2, 3]$. Figure 5 shows the Julia integer vector $[1, 2, 3]$ in MaRDI format. Note that these are not the same. In Lean a Vector is a List with a specified length. The shown vector should have type Vector ℤ 3 in Lean. So we can use the FromMrdiData instance for Lists to load the data as List. We check if the length of the List is correct and if so, construct the Vector. In Lean code:

```
instance [FromMrdiData α] : FromMrdiData $ Vector α n where
  fromMrdiData? data := do
    let l : List α ← fromMrdiData? data
    if h : l.length = n then
      return ⟨l, h⟩
    else throw s!"data list does not have length {n}"
```

h is here a proof that l has the desired length. We do not have to specify how to construct h because the problem is proven in Lean to be decidable.

Actually, many types in Lean not only contain data but also require proofs that the data fulfills certain properties. For example, permutations are defined as bijections from a type to itself. They consist of a function, an inverse function and proofs that the inverse function is really the inverse of the first function.

Generating these proofs can be complicated. I skipped some of them. These are obviously concerns for safety.

[FromMrdiData α] before the colon indicates that we require an instance for getting an element of α from data for this instance. In our example that would be FromMrdiData ℤ.

The instance for Vectors can now be used to do the same for $m \times n$-matrices.

```
instance [FromMrdiData α] : FromMrdiData (Matrix (Fin m) (Fin n) α) where
  fromMrdiData? data := do
    let v : Vector (Vector α n) m ← fromMrdiData? data
    return Matrix.of (fun x y => (v.get x).get y)
```

The data will be first converted to a Vector of Vectors which we use to define the matrix. An example for a matrix in MaRDI format can be found in Figure 6. Most types in Lean have some auxiliary functions

```
{
  "_ns": {
    "Oscar": [
      "https://github.com/oscar-system/Oscar.jl",
      "1.0.0-DEV-fbd34b88fbedbbcb729a1e2ea5037b1860cda204"
    ]
  },
  "_type": {
    "name": "Vector",
    "params": "Base.Int"
  },
  "data": ["1","2","3"]
}
```

Figure 5: JSON description of the vector $[1, 2, 3]$

which can be used to construct them from simpler data.

Converting Lean objects to MaRDI is more work but does not require any complicated proofs. The class `ToMrdi` can again be found in the Lean file with the same name. Instances for `ToMrdi` should be added by implementing instances for the individual parts `ToData`, `ToMrdiType` and `ToRefs`.

Let us take a look on the implementation for `Arrays`. The resulting MaRDI should look like the object in Figure 5. In most cases we should write an instance for every property in the MaRDI.

The data is again a simple array containing the data of every element.

```
instance [ToData α] : ToData (Array α) where
  toData uuids a := return Data.arr (← a.mapM (toData uuids))
```

The property `_type` is an object the two fields `name` and `params`. The `name` should be "Vector" for an array and the `params` should specify the type of the elements. We use `toMrdiType` with a default element of type α to get the type. Afterwards we have to convert it to data because `MrdiType.obj` expects data and not a type.

```
instance [ToMrdiType α] [Inhabited α] : ToMrdiType (Array α) :=
  ⟨fun uuids _ => return MrdiType.obj "Vector" (Mrdi.TypeToData (← toMrdiType uuids (default :
    α)))⟩
```

The array itself does not need any references but the elements of the array might. All elements should require the same references. So we use an arbitrary default one which is given by `Inhabited` α. For example, if we have an array of polynomials, we should specify the underlying ring of the polynomials.

```
instance [ToRefs α] [Inhabited α] : ToRefs (Array α) where
  toRefs uuids _ := toRefs uuids (default : α)
```

In the next example, we take a look at a free group. The `Typewrapper` indicates that we want to convert the group itself and not an element of the group. The type of a free group is specified by the simple string "FPGroup".

```
instance : ToMrdiType (TypeWrapper (FreeGroup α)) where
  toMrdiType _ _ := return MrdiType.str "FPGroup"
```

```
{
  "_ns": {
    "Oscar": [
      "https://github.com/oscar-system/Oscar.jl",
      "1.0.0-DEV-fbd34b88fbedbbcb729a1e2ea5037b1860cda204"
    ]
  },
  "_type": {
    "name": "Matrix",
    "params": "Base.Int"
  },
  "data": [["1","2"],["3","4"]]
}
```

Figure 6: JSON description of the 2×2-matrix $[[1, 2], [3, 4]]$

The data should specify what kind of finitely presented group we are working with. We should specify the GapType to indicate that it is a free group and the names of the generators. I chose to call them here $x1, x2, \ldots$

```
instance [Fintype α] : ToData (TypeWrapper (FreeGroup α)) where
  toData _ _ := do
    let names := Data.arr ⟨List.map (Mrdi.Data.str s!"x{· + 1}") (List.range (Fintype.card α))⟩
    let x := Data.mkObj [
      ("GapType", Data.str "IsFreeGroup"),
      ("wfilt", Data.str "IsLetterWordsFamily"),
      ("names", names)
    ]
    return Data.mkObj [("X", x)]
```

An example for a free group in MaRDI format can be found in Figure 7.

```
{
  "_ns": {
    "Oscar": [
      "https://github.com/oscar-system/Oscar.jl",
      "1.0.0-DEV-fbd34b88fbedbbcb729a1e2ea5037b1860cda204"
    ]
  },
  "_type": "FPGroup",
  "data": {
    "X": {
      "GapType": "IsFreeGroup",
      "wfilt": "IsLetterWordsFamily",
      "names": ["x1","x2"]
    }
  }
}
```

Figure 7: JSON description of a free group with two generators

Definitions in Mathlib are often as general as possible. This can be a problem because we can only serialize concrete objects. In particular noncomputable definitions are quite troublesome. For example, `PresentedGroup` takes a set of relations. The set can be infinite and noncomputable. But we expect a concrete, finitely presented group. The best solution I found is to expect a list of relations and build a set of it. In Lean:

```
instance [FinEnum α] {rels : List (FreeGroup α)} : ToMrdi (PresentedGroup (List.toSet rels))
```

Alternatively, one can write a meta function to extract a list from the set. `Set.toList` in `ToMrdiNoncomputable.lean` does it. But I would not recommend it because it is unnecessarily complex.

`Stream.lean` contains some meta function for reading and writing a MaRDI to a stream. These are used in the commands `#readMrdi` and `#writeMrdi` which read and write a Lean object from and to a MaRDI file. It also contains meta functions for converting a Lean expression to MaRDI and the other way around. These are important for writing tactics. When building the MaRDI, the appropriate `ToMrdi` should be implemented. To construct the expression, we need a `FromMrdi` instance. However, this might not be sufficient in practice. The instance is comparable to a program which returns the desired value. But it is preferable to use the value directly. We can obtain it by evaluating the expression. But then, we need to convert the value back into an expression to use it in tactics. This can be achieved by writing a `ToExpr` instance. But this might be tricky for complex types and adds a lot of work. Unfortunately, I did not find a better solution.

5.2 Oscar

The `Oscar` folder contains the implementation for the connection to OSCAR and the tactics for the actual applications.

`Server.jl` is a minimalist while loop which checks for a command and executes a function depending on the message. All functions implemented by me follow the same schema: parse some input, calculate what we want and send it back. The file includes also the Julia part of the applications.

Features to start and access the server are defined in `Server.lean`. It will spawn once per file with the first tactic or command in the file which wants to use it. The setup should also work for every computer algebra system other than OSCAR with minimal changes. When writing tactics which use OSCAR, the function `julia` might be useful. It takes a command and some data as MaRDI and will send it to the server. It returns the result of the calculation in Julia in form of a MaRDI. In most cases, the next step is to convert the result to an expression. The variant `julia`' incorporates this step. To do so, it also requires the desired type of the result.

The tactics used in the applications can be found in `Tactics.lean`. It also contains a tactic to load a MaRDI file and assign the corresponding expression to the goal. This can easily be used for definitions.

5.3 Sorrys

In Lean it is possible to replace a proof with `sorry`, apologizing to the reader for the missing proof. That is naturally a concern for safety and will raise a warning. As I prioritized results over filling all holes, some `sorry`s are still left in the code.

Most of them should be easily fixable. The missing proofs for the heuristics used in the tactics should not be too complicated. I can not say the same for the `sorry`s in the `FromMrdi` instances.

6 Applications

Let us get to the most interesting question: what is working in practice? I will present three applications. The examples are ordered by complexity starting with a simple one. The goal is to automate processes which are based on computations which can be done by OSCAR algorithms.

6.1 Matrix Inverse

We start with a simple example: inverting a matrix. Although it might not be the most exciting application, it showcases the overall workflow. The end result will look like this in Lean:

```
def A : Matrix (Fin 3) (Fin 3) ℚ := !![3, 0, 4; 5, 10, 6; 1, 2, 3]
def A_inv : Matrix (Fin 3) (Fin 3) ℚ := by matrix_inverse A
```

First we define a 3×3-matrix `A`. Then we define its inverse using a tactic that utilizes the inversion method in Julia. Proving that `A_inv` is the inverse of `A` can be done in a few lines.

The `matrix_inverse` tactic works as follows: First we convert the matrix to MaRDI. Then, we send a request to the server with a command, followed by the serialized matrix. The server will take the command, read the MaRDI, rebuild the matrix, invert it, serialize the result and send it back. After parsing the result in Lean, we have to turn the new MaRDI into a matrix. We start by deriving the type of the initial matrix, as it has the same type as the desired result. This is needed to find the correct instance of `FromMrdi`. Now we evaluate the expression and convert it back into an expression using a `ToExpr` instance. This step could be skipped but then `A_inv` would be defined as the result of a computation, making it very unpleasant to work with. For example, I could not manage to show `A * A_inv = 1` in reasonable time when skipping this step. The downside is of course that we have to implement the `ToExpr` instance which can be tricky for some types. Finally, we assign the resulting expression as the definition of `A_inv`.

6.2 Permutation Group Membership

In the next example we automate proving that a permutation `g` is in the group G generated by finitely many permutations `g`$_i$. In Lean:

```
example : g ∈ Group.closure {g₀, g₁, g₂, g₃, g₄} := by
  perm_group_membership
```

The idea is to rewrite `g` as product of the generators and their inverses. With such a representation, the proof can be easily constructed using the facts that the group is closed under multiplication and inverses. The difficulty is obviously in finding such a representation. Luckily, there are some algorithms in OSCAR which can do the work for us.

We start by constructing a list of the permutations to send it to Julia. Here we already encounter the first problem: `Group.closure` takes a set of permutations, not a list. This poses a problem as there is no way to turn every set into a list containing the same elements. Even if we require the set to be finite, we will have trouble with noncomputable definitions. I do not think that there is a perfect solution for this problem. My approach here is to check if the set is constructed in a special way. This works with the syntax shown above.

Once we have a list of the permutations, we can serialize it and send it to Julia. We use the permutations to define G and then construct an epimorphism ϕ from a free group with the same number of generators to G by mapping generators to generators. Thus, we look for the preimage w of g with respect to ϕ. This is one simple, inconspicuous line in Julia where the true magic of this proof happens. Finding the preimage is nontrivial and makes use of efficient algorithms of the computer algebra system. The free group word w will be send back to Lean. $\phi(w)$ gives us a representation of g as a product of the generators. So we can finish the proof as described above.

In Julia:

```
input = IOBuffer(readline())
l = load(input)
g = l[1]
gens = l[2:end]
G = permutation_group(degree(parent(g)), gens)
epi = epimorphism_from_free_group(G)
w = preimage(epi, g)
save(stdout, w)
```

Note: For a clean solution, the epimorphism ϕ should also be sent back. It is only canonical up to permutation of the generators. Unfortunately, saving epimorphisms as a MaRDI was not implemented in OSCAR at the time I wrote the code. In practice, it works anyway because we can guess which generator is mapped to which.

The preimage is computed by using stabilizer chains. A stabilizer chain is a sequence of subgroups $S_1, \cdots, S_k$ with $S_i \subset S_{i+1}$ and S_{i+1} stabilizes a point which is not stabilized by S_i. There are numerous choices to make. Implementing good heuristics is complicated but has a big impact on the speed of the algorithm. This may lead to fast but uncertified results, making a validation valuable.

If the algorithm fails to find a preimage, for example, if no preimage exists, it will raise an error. This error will be sent to Lean, causing the tactic to fail and the error to be reported to the user.

This is a typical example for a problem involving a factorization. The hard part of the proof, finding the factors, can be carried by the computer algebra system. The verification is here much easier.

6.2.1 Sample

The attentive reader might recognize the similarities to the example involving the Rubik's cube. We can use the tactic to show that a transformation of the cube can be reached by legal moves. Let us first define the six permutations corresponding to the rotation around a face of the cube.

```
def top    : Equiv.Perm (Fin 48) := c[ 0,  2,  7,  5] * c[ 1,  4,  6,  3]
  * c[ 8, 32, 24, 16] * c[ 9, 33, 25, 17] * c[10, 34, 26, 18]
def left   : Equiv.Perm (Fin 48) := c[ 8, 10, 15, 13] * c[ 9, 12, 14, 11]
  * c[ 0, 16, 40, 39] * c[ 3, 19, 43, 36] * c[ 5, 21, 45, 34]
```

```
def front  : Equiv.Perm (Fin 48) := c[16, 18, 23, 21] * c[17, 20, 22, 19]
  * c[ 5, 24, 42, 15] * c[ 6, 27, 41, 12] * c[ 7, 29, 40, 10]
def right  : Equiv.Perm (Fin 48) := c[24, 26, 31, 29] * c[25, 28, 30, 27]
  * c[ 2, 37, 42, 18] * c[ 4, 35, 44, 20] * c[ 7, 32, 47, 23]
def rear   : Equiv.Perm (Fin 48) := c[32, 34, 39, 37] * c[33, 36, 38, 35]
  * c[ 2,  8, 45, 31] * c[ 1, 11, 46, 28] * c[ 0, 13, 47, 26]
def bottom : Equiv.Perm (Fin 48) := c[40, 42, 47, 45] * c[41, 44, 46, 43]
  * c[13, 21, 29, 37] * c[14, 22, 30, 38] * c[15, 23, 31, 39]
```

Note that the numbers start at 0 and not at 1. So we have to subtract 1 from each number.
Now we can define the `G` as the group of transformations of Rubik's magic cube

```
def G := Group.closure {top, left, front, right, rear, bottom}
```

and verify if an element `g` is in `G`.

```
def g : Equiv.Perm (Fin 48) := c[16, 18] * c[10, 7] * c[5, 24] * c[6, 27] * c[17, 20]


set_option maxRecDepth   1000000000000000000000
set_option maxHeartbeats 1000000000000000000000


theorem g_in_G : g ∈ G := by
  unfold G
  perm_group_membership
```

You might need some patience for bigger examples. This one might take a minute. You can also use the factorization to solve the Rubik's cube from any transformation without many changes to the proof.

6.3 KBMAG

The goal of the next example is to prove word problems in finitely presented groups. We want do decide if two expressions are equal with respect to the relations of the finitely presented group. This is an undecidable problem. So there can be no algorithm which will always get us the correct result. Nethertheless, there are some algorithms which work well in most situations that might come up in practice.

6.3.1 Knuth-Bendix algorithm

We use the GAP package KBMAG [13]. It enables running the Knuth-Bendix completion algorithm on finitely presented groups to compute an automatic structure. The algorithm generates a set of rewrite rules, all derived from the relations and the identities $g * g^{-1} = 1$ for all generators g. New rules are generated by overlapping two existing rules, after which we check if these new rules can be simplified using other rules. This process is repeated until the rewrite system is confluent or the process is aborted. An example for a rewrite rule would be $g_1 \cdot g_2 \to 1$. It indicates that $g_1 \cdot g_2 = 1$ and if we find the term $g_1 \cdot g_2$ in another term we can replace it by 1. If we start with two rules $g_2 \cdot g_1 \to 1$ and $g_1 \cdot g_3 \to g_2$, we can overlap them because the left side of the first starts with the same term (g_1) as the end of the left side of the second rule. We can deduce the new rule $g_2^2 \to g_3$. The implementation in KBMAG tests all new equations for overlaps with the identities $g * g^{-1} = 1$ for all generators g. If the rules overlap and generate a new rule which is not redundant, the new rule is added to the rewrite system. The procedure is repeated until no overlaps produce new rules. The algorithm has to generate lots of rules to get to the desired result. The amount of possibilities along the way is huge. So it is crucial to have good heuristics on where to continue exploring. Otherwise, available resources like memory might exhaust before a confluent rewrite system is found.

The rewriting rules form a rewriting system that can be used to reduce words. Given an total ordering o for the group, the Knuth-Bendix algorithm tries to find a rewriting system R such that R reduces all elements to normal form. The normal form of an element g with respect to o is the presentation of g which minimizes o. Such an rewriting system is called confluent. Obviously, all rewriting rules should decrease the ordering. Notably, the order in which the rewrite rules are applied does not matter in a confluent system. The choice of the ordering can determine if a confluent system exists or not and is therefore essential. The standard ordering is shortlex which orders primarily by the length of the word and secondary lexicographical. The algorithm offers effective solutions for many naturally appearing groups such as hyperbolic and Euclidean groups.

6.3.2 Integration in Lean

If the program succeeds, the automatic structure allows us to efficiently solve word problems in the group, a task which is generally undecidable. In this case, we want to replicate the calculation in Lean. The tactic starts by inferring the type of the group, the relations and the degree. We have to submit an element of the group. This makes it easier to get all the information. With some more effort this could be received from the goal. The group is then sent to OSCAR to find a confluent rewrite system using the Knuth-Bendix algorithm. While running KBMAG, we print all rewriting rules and the rules used to generate them. This information is sufficient to introduce the rules as equations in Lean. I did not manage in time to manipulate the algorithm to also print which equations are used to simplify other equations. Instead, I use some overly complicated heuristics and all previously introduced equations to simplify new equations. This does not necessarily lead to the same result and can sometimes oversimplify the equation, even reducing it to `True`, which makes it absolutely useless. It also results in a lot of additional computing time.

Despite these problems, the approach works on multiple test cases. We can for example automatically prove that $a = 1$ and $b = 1$ in the group $\langle a, b \mid [a, b] \cdot a^{-1} = 1,\ [b, a] \cdot b^{-1} = 1 \rangle$ and therefore conclude that the group is trivial. But it is not difficult to find examples where the tactic fails. In Lean:

```
@[reducible]
def f := FreeGroup (Fin 2)


@[reducible]
def a : f := FreeGroup.mk [(0, true)]
@[reducible]
def b : f := FreeGroup.mk [(1, true)]
```

```
@[reducible]
def rels_list := [a⁻¹ * b⁻¹ * a * b * a⁻¹, b⁻¹ * a⁻¹ * b * a * b⁻¹]
@[reducible]
def rels := List.toSet rels_list

@[reducible]
def g := PresentedGroup rels

set_option maxRecDepth   10000000000000000000000
set_option maxHeartbeats 10000000000000000000000

theorem g_triv : ∀ (x : g), x = 1 := by
  kbmag (1 : g)

theorem a_eq_b : (PresentedGroup.of 0 : g) = (PresentedGroup.of 1 : g) := by
  kbmag (1 : g)
```

There is room for further improvement. It is likely that many rules do not have any effect on the end result. Filtering those out would also reduce the effort required to construct the proof.

7 Conclusion

The implemented interface is functional and extendable. The effort required for serializing and deserializing each type of data needs to be done only once and the structure for doing so is similar across all data types.

I showcased three potential applications to further enhance automation in Lean. And there is no reason to stop here. There is much room for future work building on the implemented tools.

The approach is not confined to a single computer algebra system. If other systems support the same serialization as OSCAR, the amount of work required to incorporate them would be minimal.

7.1 Further Work

There are a numerous ways to expand the project. Covering more types of (de-)serializable types can obviously never hurt. Same goes for adding tactics to solve various problems. The most promising targets are tasks that are computationally intensive but easy to verify. This includes factorization problems like polynomial factorization or prime factorization. QR decomposition and eigenvalues of matrices should also be feasible challenges. Solving (differential) equations is also a hard task, but verifying the solutions should be straightforward in most cases.

If the goal is to make the tool available to as many people as possible, I would recommend following the approach used by the tactic polyrith in Mathlib. It passes on the computations to a server instead of doing it locally. This way, the user skips all the setup troubles.

There are some issues that still need attention. All the `sorry`s should be replaced by proofs. Some more jobs are marked as "TODO" in the code.

It is still more work per data type than I anticipated. Particularly the need for `ToExpr` instances is quite annoying. Intuitively I would guess that there should be a way to eliminate the problem. This might be an ambitious project but it would be a big improvement.

So far the focus was on enhancing automation in formalization. But ensuring correctness of computations is also desirable and can be done using the interface. This might be valuable for basic operation if it can be automated because they are used in many algorithms.

Verifying computations which are performed once but used a lot afterwards might also be lucrative. Databases belong to this category. For instance, GAP has multiple libraries containing collections of groups. These should of course come with some properties which could be validated.

References

[1] Mara Belotti and Marta Panizzut. *Discrete geometry of Cox rings of blow-ups of* $\mathbb{P}^3$. 2022. arXiv: 2208.05258 [math.AG]. URL: https://arxiv.org/abs/2208.05258.

[2] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[3] Jeff Bezanson et al. "Julia: A fresh approach to numerical computing". In: *SIAM Review* 59.1 (2017), pp. 65–98. DOI: 10.1137/141000671. URL: https://epubs.siam.org/doi/10.1137/141000671.

[4] Kevin Buzzard, Johan Commelin, and Patrick Massot. "Formalising perfectoid spaces". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. POPL '20. ACM, Jan. 2020. DOI: 10.1145/3372885.3373830. URL: http://dx.doi.org/10.1145/3372885.3373830.

[5] The mathlib Community. "The lean mathematical library". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 367–381. ISBN: 9781450370974. DOI: 10.1145/3372885.3373824. URL: https://doi.org/10.1145/3372885.3373824.

[6] The MaRDI consortium. *MaRDI: Mathematical Research Data Initiative Proposal*. May 2022. DOI: 10.5281/zenodo.6552436. URL: https://doi.org/10.5281/zenodo.6552436.

[7] Daniel Corey and Dante Luber. "Degenerations of the Grassmannian via Matroidal Subdivisions of the Hypersimplex". In: *Séminaire Lotharingien de Combinatoire* 89B.Article #44 (2023). Proceedings of the 35th Conference on Formal Power Series and Algebraic Combinatorics (Davis), 12 pp.

[8] Wolfram Decker et al. SINGULAR *4-3-0 — A computer algebra system for polynomial computations*. http://www.singular.uni-kl.de. 2022.

[9] Claus Fieker et al. "Nemo/Hecke: Computer Algebra and Number Theory Packages for the Julia Programming Language". In: *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*. ISSAC '17. New York, NY, USA: ACM, 2017, pp. 157–164. DOI: 10.1145/3087604.3087611. URL: https://doi.acm.org/10.1145/3087604.3087611.

[10] *GAP – Groups, Algorithms, and Programming, Version 4.12.2*. The GAP Group. 2022. URL: https://www.gap-system.org.

[11] Ewgenij Gawrilow and Michael Joswig. "polymake: a framework for analyzing convex polytopes". In: *Polytopes—combinatorics and computation (Oberwolfach, 1997)*. Vol. 29. DMV Sem. Basel: Birkhäuser, 2000, pp. 43–73.

[12] Thomas C. Hales. "A proof of the Kepler conjecture". English. In: *Ann. Math. (2)* 162.3 (2005), pp. 1065–1185. ISSN: 0003-486X. DOI: 10.4007/annals.2005.162.1065.

[13] D. Holt and T. The GAP Team. *kbmag, Knuth-Bendix on Monoids and Automatic Groups, Version 1.5.11*. https://gap-packages.github.io/kbmag. Refereed GAP package. Jan. 2023.

[14] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media, 2002.

[15] *OSCAR – Open Source Computer Algebra Research system, Version 1.0.0*. The OSCAR Team, 2024. URL: https://www.oscar-system.org.

[16] Antony Della Vecchia, Michael Joswig, and Benjamin Lorenz. *A FAIR File Format for Mathematical Software*. 2023. arXiv: 2309.00465 [cs.MS].

[17] Austin Wright et al. *JSON Schema: A media type for describing JSON documents*. https://json-schema.org/draft/2020-12/json-schema-core.html. JSON Schema, 2020.