Certifying Solutions to Permutation Group Problems

Arjeh Cohen[1], Scott H. Murray[1,⋆], Martin Pollet[2,⋆], and Volker Sorge[3,⋆⋆]

[1] RIACA, Technische Universiteit Eindhoven, The Netherlands
{amc,smurray}@win.tue.nl
http://www.win.tue.nl/~{amc,smurray}
[2] Fachbereich Informatik, Universität des Saarlandes, Germany
pollet@ags.uni-sb.de
http://www.ags.uni-sb.de/~pollet
[3] School of Computer Science, University of Birmingham, UK
V.Sorge@cs.bham.ac.uk
http://www.cs.bham.ac.uk/~vxs

Abstract. We describe the integration of permutation group algorithms with proof planning. We consider eight basic questions arising in computational permutation group theory, for which our code provides both answers and a set of certificates enabling a user, or an intelligent software system, to provide a full proof of correctness of the answer. To guarantee correctness we use proof planning techniques, which construct proofs in a human-oriented reasoning style. This gives the human mathematician the necessary insight into the computed solution, as well as making it feasible to check the solution for relatively large groups.

1 Introduction

In this paper, we describe the integration of permutation group algorithms from computer algebra with proof planning. We consider eight basic questions arising in computational permutation group theory, for which a computer algebra system provides solutions together with sets of certificates. A certificate is data that enables a human mathematician to easily check the computed result. We employ the same certificates to formally guarantee correctness with proof planning.

The experiments were carried out by combining the computer algebra system GAP [6] and the proof planner of the Omega system [4]. We chose GAP, since it is particularly good in group theory and has a rich collection of permutation group algorithms (we could equally well have chosen Magma, however). The choice of Omega was motivated by the fact that it enables the construction of proofs in a human-oriented reasoning style.

The results of our experiments can be summarised as follows: (i) We provide GAP functions which handle eight basic queries, ranging from "Is this permutation in that permutation group?" to "What is the order of this permutation

⋆ The author's work is supported by EU IHP grant Calculemus HPRN-CT-2000-00102.
⋆⋆ The author's work was supported by British Council Travel Grant PPS732

F. Baader (Ed.): CADE-19, LNAI 2741, pp. 258–273, 2003.

group?" The functions provide certificates as well as solutions, enabling a user or an intelligent software system to provide a full proof of correctness of the solution. (ii) We provide proof planning constructs in Omega to prove that the answers given by GAP to these eight queries are correct; each query can essentially be modelled as a theorem and the proof planner can verify GAP's answers using the additional certificates to guide the planning process. Omega treats single queries independently from GAP, in a modular and easily extensible approach, thus showing that the certificates are not only sufficient to construct a formal verification but also to plan a corresponding proof problem independently of the actual computation. As far as possible, we model a human-oriented reasoning style to give a human mathematician the necessary insights into the computed solutions.

(iii) To this end we also implemented a collection of functions in GAP for turning answers and certificates into natural language proofs using simple templates. Similarly, the proofs produced by Omega can be turned into natural language by means of the related P.Rex system [5], which employs elaborate linguistic techniques. Notice that the natural language proof from GAP does not satisfy a correctness criterion, as it is implemented by humans and so a mistake in the proof could remain unnoticed.

The need to integrate automated proof assistants with computer algebra systems comes from two sources. On the computer algebra side, the production of certificates becomes necessary when systems are used via the Internet as 'oracles.' In this setting, it is likely that users will not know where the answer is coming from and so will need to be convinced of its correctness. With a certificate, it is a relatively easy task to perform a verification. On the proof planning side, we wish to demonstrate that we meet the challenge of reconstructing a proof from exactly the same mathematical data that a human would require. It would be unreasonable to expect more than that from a proof developing system. The fact that Omega can produce a proof from a certificate also ensures that the certificates have indeed supplied sufficient information.

We emphasise that this work extends the boundaries of what is feasible (cf. [7,9,13,2]). Clearly, the eight permutation group queries we are dealing with can all be handled by simple enumeration. For instance, in order to decide if a given permutation g belongs to a permutation group G, you could just enumerate all elements of G and check whether g is one of these. However, G can be exponentially large as a function of n, the number of letters permuted by G, and so this soon becomes impractical. We use GAP's sophisticated group-theoretic algorithms for finding proofs in cases far beyond reach of such enumeration.

The paper is organised as follows: We first introduce the permutation group queries and their implementation in GAP. We then discuss their formalisation in the Omega system. Section 4 gives a brief introduction to the integration of proof planning and computer algebra and Sec. 5 highlights some of the planning issues for the permutation group queries. We then give an overview of experimental results and conclude with a discussion of some related work.

2 Eight Queries Regarding Permutation Groups

In computational permutation group theory, a group G is specified by a list of generating permutations $A = [a_1, a_2, \ldots, a_k]$, where a_i is a permutation on the points $\Omega := \{1, 2, \ldots, n\}$. In other words, the elements of A belong to the symmetric group Sym_n. We often write $G = \langle A \rangle$ to denote that G is generated by A. Our permutations act on the right.

In this section, which is based on [3], we give efficient solutions of eight basic queries. GAP already has efficient built-in functions which answer these queries. We have written functions in the GAP language that provide certificates for the correctness of these answers. The certificates, taken together with the answers themselves, give enough data to construct an informal proof of correctness. Naturally, our certificate is not a complete proof in itself. In particular, we leave things like the computation of the image of a permutation acting on a point (i.e., a member of Ω) to the user. But apart from such elementary evaluations, the completion of the proof should be straightforward. In later sections we confirm this by letting an automated proof planning device find a proof. The following example is used throughout:

Example 1. The Mathieu group on 11 points, denoted by M, is generated by the list $A = [a_1, a_2]$, where: $a_1 = (1, 10)(2, 8)(3, 11)(5, 7)$, $a_2 = (1, 4, 7, 6)(2, 11, 10, 9)$.

2.1 Membership

We first address the question of proving that the permutation g belongs to the group G. A *word* in A is an expression of the form $a_{i_1}^{e_1} a_{i_2}^{e_2} \cdots a_{i_m}^{e_m}$ where the indices i_j are in the range $1, \ldots, k$ and the exponents e_j are integers. It is now easily shown that a permutation $g \in \mathrm{Sym}_n$ is an element of G if, and only if, it can be expressed as a word in A. Writing an arbitrary permutation g as a word in A is a difficult computational problem. We use the existing methods implemented in GAP.

Query input A permutation g and a list A of permutations generating G. We are given the fact that $g \in \langle A \rangle$.

GAP certificate A word w in A that evaluates to g.

Query output By definition, G is generated by A and so G consists of those elements that can be expressed as a word in A. In particular, w is equal to g in Sym_n, and so belongs to G.

Example 2. For our group M of Example 1 and $g = (1, 3, 8, 9)(4, 10, 6, 5)$, our GAP function returns $a_1 a_2^3 a_1$. For $g = (3, 9)(4, 5)(6, 10)(7, 11)$, a GAP certificate is $a_2 a_1 a_2^3 a_1 a_2^2 a_1 a_2^3 a_1 a_2^2 a_1 a_2 a_1 a_2^2 a_1 a_2 a_1 a_2 a_1 a_2^3 a_1$. In fact, the GAP algorithm is non-deterministic, so the certificate may vary on different calls for the same query. We have selected a certificate of moderate length for these examples; in practice, the words can be much longer.

2.2 Subgroup

Suppose H is a permutation group with generating set B and that we wish to prove that H is a subgroup of G. Recall that H is a subgroup of G if, and only if, every element of B is contained in G.

Query input A list A of permutations generating G and a list B of permutations generating H. We are given the fact that H is a subgroup of G.

GAP certificate A list W of words in A indexed by B.

Query output In order to show that H is a subgroup of G, it suffices to show that each element of B belongs to G. The list W gives, for each b in B, an expression for b as a word W_b in A. This establishes that each element of B belongs to G, and so H is a subgroup of G.

Example 3. Consider the permutation group H generated by the list $B = [b_1, b_2]$, where $b_1 = (1,3,8,9)(4,10,6,5)$ and $b_2 = (3,9)(4,5)(6,10)(7,11)$. A GAP certificate is $[a_1a_2^3a_1, a_2a_1a_2^3a_1a_2^2a_1a_2^3a_1a_2^2a_1a_2a_1a_2^2a_1a_2a_1a_2a_1a_2^3a_1]$.

2.3 Orbit

We wish to determine the *orbit* of $x \in \Omega$ under the action of G; i.e., we wish to find $xG = \{xg : g \in G\}$.

Query Input A list A of permutations generating G and a point x.

GAP certificate A set of points X and a list t of words in A indexed by X. The set X is just the G-orbit xG of x. The word $t(y)$ in A indexed by $y \in X$ maps x to y.

Query output In order to show that X is the G-orbit of x, we need to show that (1) each element of A leaves the set X invariant. This is a straightforward check that the cycles containing points of X do not contain any points not in X. And (2) each element of X is image of x under an element of G. These elements are given in the list t.

Example 4. The M of 1 is $\{1, \ldots, 11\}$. A GAP certificate for this orbit is described by the table on the right. The even columns contain words $t(x)$ for x in the preceding column. These words are found by first computing the corresponding permutations and then proceeding as in Sec. 2.1.

x	$t(x)$	x	$t(x)$	x	$t(x)$	x	$t(x)$
1	$\langle \mathrm{id} \rangle$	4	a_2	7	a_2^2	10	a_1
2	$a_1a_2^2$	5	$a_2^2a_1$	8	$a_1a_2^2a_1$	11	$a_1a_2^3$
3	$a_1a_2^3a_1$	6	a_2^3	9	a_1a_2		

2.4 Schreier Tree

It is inefficient to store all the words $t(y)$ for y in the G-orbit $X = xG$, so instead we construct a *Schreier tree*. This is a tree rooted at x whose its nodes are the elements of X, in which every child z of a node y is connected to it by the label $c \in A \cup A^{-1}$ if $yc = z$. For every $y \in X$, there is a unique path in the Schreier tree from x to y. The labels on this path will form a word representing $t(y)$. This is the *Schreier word*.

In practice, we store the Schreier tree in a linearised form, using two vectors $\omega : X \to X \cup \{0\}$ and $v : X \to \{-m, \ldots, -1, 0, 1, \ldots, m\}$ defined by:

$$\omega(z) = \begin{cases} y \text{ if } z \text{ is a child of } y \\ 0 \text{ if } z = x \end{cases} \qquad v(z) = \begin{cases} l & \text{if } y = \omega(z) \text{ and } ya_l = z \\ -l & \text{if } y = \omega(z) \text{ and } ya_l^{-1} = z \\ 0 & \text{if } z = x \end{cases}$$

Query Input A list $A = [a_1, \ldots, a_k]$ of permutations generating G and a point x of Ω.

GAP certificate A triple $[X, \omega, v]$ of integer sequences consisting of the orbit, the Schreier vector, and the back-pointers.

Query output Consider the triple $[X, \omega, v]$. It has three rows, the first of which represents the orbit $X = xG$. In order to show that X is indeed the G-orbit containing x, see Sec. 2.3. To show that v and ω are a linearised Schreier tree, consider a column of the table, say x_j, ω_j, v_j. It suffices to show that $x_j = \omega_j a_{-v_j}^{-1}$ if $v_j < 0$ and that $x_j = \omega_j a_{v_j}$ if $v_j > 0$. This is a (tedious but) trivial check.

Example 5. The linearised version of a Schreier tree of M rooted at 1 is given in the following table.

X	1	10	4	6	9	11	7	2	3	5	8
ω	0	1	1	1	10	10	4	9	11	7	2
v	0	1	2	-2	2	-2	2	2	1	1	1

The Schreier words $t(y)$ for y in the M-orbit $1M = \{1, \ldots, 11\}$ are as indicated in Example 4.

As suggested by the example, the set U of values of t can be used to construct the table of the proof in Sec. 2.3 that X is an M-orbit.

2.5 Stabiliser

The *stabiliser subgroup* in G of x is defined as $G_x = \{g \in G : xg = x\}$. It is not immediately clear how to compute this subgroup; although the definition gives us a test for whether g is an element of G_x, it does not give us a generating set.

The following lemma establishes a one-to-one correspondence between the orbit of a point and the set of cosets of its stabiliser.

Lemma 1 (Orbit Lemma). *If $y \in xG$, then $\{g \in G : xg = y\}$ is a coset of G_x. In particular, $|xG| = |G|/|G_x|$.*

The Schreier tree enables us to create a set U of coset representatives for G_x in G and a map $t : G \to U$ sending an element g of G to the representative of $G_x g$: given $g \in G$, take $t(g)$ to be the Schreier word $t(xg)$. Then $xt(g) = xg$ and $t(g) = t(hg)$ whenever $h \in G_x$. So, taking U to be the image of the map $t : G \to G$, we find that U and t are as required.

Now that we have a set of coset representatives for G_x, we can use it to compute a generating set for the stabiliser of x in G. This is based upon the following lemma.

Lemma 2 (Schreier's lemma). *Suppose that G is a group with generating set A, and H is a subgroup of G. If U is a set of coset representatives for H in G, and the function $t : G \to U$ maps an element g of G to the representative of Hg, then a generating set for H is given by $\{ua\,t(ua)^{-1} : u \in U, a \in A\}$.*

Query Input A list A of permutations generating G and a point x of Ω.
GAP certificate A set B of generators of a subgroup H of G; a Schreier tree S for G at x; and a sequence of quadruples (y, i, g, h), one for each point $y \in X$ and index $i = 1, \ldots, |A|$. Here g and h both represent the Schreier generator $t(y)a_i t(t(y)a_i)^{-1}$ with a_i the i-th element of A, in case of g as a word in the generators A of G, and in the case of h as a word in the generators B of H.
Query output As in Sec. 2.2, the group H is a subgroup of G. We verify that each element of B, and hence each element of H, fixes x. As before, the triple S is a Schreier tree for G at x. We find from it the Schreier elements $t(y)$ for $y \in X$. Using the quadruples (y, i, g, h), we check that each Schreier element belongs to H. By Lemma 2, we conclude that H is the stabiliser in G of x.

Example 6. The stabiliser in M of $x = 1$ is the group $H = \langle (2,9)(3,6)(4,7)(8,11), (2,3,5,11)(4,7,6,8), (3,8,7,6)(4,9,11,5), (4,5,6,11)(7,9,10,8)\rangle$. As in Sec. 2.2, it can be checked that H is a subgroup of M. Since 1 does not appear in these generating cycles, we know that H stabilises 1. The Schreier data for M at x is as in Example 5. We list the non-trivial Schreier generators, in the format (i, j): word in A for $u_i a_j\, t(u_i a_j)^{-1}$.

$(4,1) : a_2 a_1 a_2^{-1}$ $(2,2) : a_1 a_2^4 a_1$ $(5,2) : a_2^2 a_1 a_2 a_1 a_2^2$
$(6,1) : a_2^{-1} a_1 a_2$ $(3,1) : a_1 a_2^{-1} a_2 a_1$ $(8,2) : a_1 a_2^2 a_1 a_2 a_1 a_2^2 a_1$
$(9,1) : a_1 a_2 a_1 a_2^{-1} a_1$ $(3,2) : a_1 a_2^{-1} a_1 a_2 a_1 a_2 a_1$

By Sec. 2.2, we can verify that each Schreier generator is in H. By Schreier's lemma, H is the stabiliser in M of x.

2.6 Base

We can repeat the process of the previous section to form a chain of stabiliser subgroups. A *base* for G is a finite sequence $B = [x_1, \ldots, x_k]$ of distinct points in Ω such that $G_{x_1,x_2,\ldots,x_k} = \langle \mathrm{id}\rangle$, the trivial group. Hence, the only element of G which fixes all of the points $x_1, x_2, \ldots, x_k$ is the identity. Clearly every permutation group has a base, but not all bases for a given group are of the same length. We have a *stabiliser chain* $G = G^{(0)} \geq G^{(1)} \geq \cdots \geq G^{(k-1)} \geq G^{(k)} = \langle \mathrm{id}\rangle$, if we write $G^{(i)} = G_{x_1,x_2,\ldots,x_i}$.

A base can be constructed by starting with $B = [x_1]$, and recursively choosing a letter x_i in a nontrivial $G_{x_1,\ldots,x_{i-1}}$-orbit and appending it to B. The construction is finished when $G_{x_1,\ldots,x_i} = \langle \mathrm{id}\rangle$.

Query Input A list A of permutations generating G.
GAP certificate A base $B = [x_1, \ldots, x_k]$ and, for each $i = 1, \ldots, k$, a set A_i of generators of the stabiliser of x_i in $\langle A_{i-1}\rangle$.
Query output Since $\langle A_k\rangle = 1$, we conclude that B is a base with stabiliser chain the groups $\langle A_i\rangle$ for $i = 1, \ldots, k$.

Example 7. The Mathieu group on 11 points, M, has a base $[1, 2, 3, 4]$.

2.7 Nonmembership

Here we deal with the query that is complementary to the first one treated: Prove that the permutation g does not belong to G. Suppose we have a base $B = [x_1, \ldots, x_k]$. Take $G^{(i)} = G_{x_1,\ldots,x_{i-1}}$ and $U^{(i)} = t_i(G^{(i)})$, the set of Schreier elements corresponding to a Schreier tree for $G^{(i)}$ rooted at x_{i-1}. Then we get a chain of subgroups $G = G^{(0)} \geq G^{(1)} \geq \cdots \geq G^{(k-1)} \geq G^{(k)} = \langle \mathrm{id} \rangle$ and sets $U^{(i)}$ of coset representatives for $G^{(i+1)}$ in $G^{(i)}$.

An element g of G is contained in exactly one coset of $G^{(1)}$ in $G^{(0)}$, so $g = h_1 u_0$ for some $h_1 \in G^{(1)}$ and $u_0 \in U^{(0)}$. By induction, we can show that $g = u_k u_{k-1} \cdots u_1 u_0$ where each $u_i \in U^{(i)}$ is uniquely determined by g. This process, called *sifting* an element, gives a canonical form for the elements of G and underpins most of the more advanced applications of stabiliser chains.

On the other hand, if g is not in G, then sifting fails because at some stage we get that $x_i h_{i-1}$ is not in the orbit $x_i G^{(i-1)}$, and so h_{i-1} is not in $G^{(i-1)}$. This gives us our proof of nonmembership. However, multiplying the h_i together we can simplify the result so that we only need to return one additional element $h \in G$.

Query Input A list A of permutations generating G and a permutation g. The fact that g is not in G.

GAP certificate A list $B = [x_1, \ldots, x_k]$, the orbit $x_k G^{(k-1)}$ of x_k under the stabiliser $G^{(k-1)} = G_{x_1,\ldots,x_{k-1}}$ of all elements of B except for x_k, and a permutation $h \in G$ such that gh fixes $x_1, \ldots, x_{k-1}$ and $x_k gh \notin x_k G^{(k-1)}$.

Query output As $x_k gh \notin x_k G^{(k-1)}$, the element gh is not in $G^{(k-1)}$, whence not in G. As $h \in G$, this implies that g does not belong to G.

Example 8. Let $g = (1, 2)$. We show that g does not belong to M. Take $B = [1, 2, 3, 4, 5]$. Let $h = (1, 2)(5, 8)(7, 10)(9, 11)$. Then $h \in M$ (proof as in Sec. 2.1). By verification, $gh = (5, 8)(7, 10)(9, 11)$ fixes each element of B except for 5. By a proof as in Sec. 2.6, the stabiliser in M of $[1, 2, 3, 4]$ is the trivial group. Now consider the point $x = 5$. Its image under gh is the point $xgh = 5(5, 8)(7, 10)(9, 11) = 8$. But the stabiliser in M of $[1, 2, 3, 4]$ is the trivial group and so cannot move 5 to 8. Therefore, gh does not belong to M. As h belongs to M, this implies that g is not in M.

2.8 Order

The order of a permutation group can now be effectively computed.

Lemma 3 (Order lemma). *Suppose that G is a permutation group and $B = [x_1, \ldots, x_k]$ is a base for G. Then $|G| = \prod_{i=1}^{k} |x_i G^{(i-1)}|$.*

Query Input A list A of permutations generating G.

GAP certificate A base $B = [x_1, \ldots, x_k]$, the corresponding stabiliser chain $G^{(i)}$, and the sizes of the orbits $x_i G^{(i-1)}$.

Query output The proof that the base and the stabiliser chain are correct is given in Sec. 2.6. By the Order lemma 3, the order of G is the product of the orbit sizes $|x_i G^{(i-1)}|$ for $i = 1, \ldots, k$.

Example 9. From Example 7 it immediately follows that M has order
$|1\,M| \cdot |2\,M^{(1)}| \cdot |3\,M^{(2)}| \cdot |4\,M^{(3)}| = 11 \cdot 10 \cdot 9 \cdot 8 = 7920.$

3 Formalisation Issues

We briefly discuss some of the issues of formalising the concepts and problems above in Omega's simply typed higher-order lambda calculus.

The primary objects of interest are permutations. While there are different notations in mathematics to express permutations, the cycle notation is usually preferred and used by GAP. In this notation a permutation consist of duplicate-free disjoint cycles, i.e. lists $(n_1, \ldots, n_k)$ of points with $k \geq 1$ and $n_i \neq n_j$ for $i \neq j$. A cycle maps the point n_i to n_{i+1} for $i = 1, \ldots, k-1$ and n_k to n_1. A permutation is then either a set containing disjoint cycles or the composition of permutations. Both cycles and permutations are interpreted as mappings by an application operator @ that takes a permutation and a point in its domain, and returns the image. We identify cycles and permutations if their application results in the same mapping.

In order to facilitate the use of concrete permutations in our problem domain, we have implemented finite sets, cycles and lists as *annotated constants*, similar to rational numbers in Omega. Annotated constants are treated as normal logical constants in a proof, but are annotated with a concrete mathematical object, such as a set or a number. Special tactics can then directly compute with these objects. Thus, with annotated constants, trivial properties of concrete objects are already implemented on the term level. Annotated constants can still be expanded to their definitions on a more primitive term level. For instance the declarations $\{a, b, c\}$ and $\{b, a, c\}$ both denote the same object, namely a constant that has the term $\lambda x.(x = a \vee x = b \vee x = c)$ as its definition. Similarly the permutations $(1, 2)(3, 4)$ and $(3, 4)(2, 1)$ are equal terms in Omega.

The definitions of other important concepts are given in the following table[1]:

$$
\begin{aligned}
\mathit{Orbit}(G_{\alpha\to o}, @_{\alpha\to\beta\to\beta}, x_\beta) &\equiv \lambda y_\beta.\, \exists g{:}G.\, y = g@x \\
\mathit{Stabiliser}(G_{\alpha\to o}, @_{\alpha\to\beta\to\beta}, x_\beta) &\equiv \lambda g_\alpha.\, g \in G \wedge g@x = x \\
\mathit{StabChain}(G_{\alpha\to o}, @_{\alpha\to\beta\to\beta}, (a :: l)_{list}) &\equiv \mathit{Stabiliser}(\mathit{StabChain}(G, @, l), @, a) \\
\mathit{StabChain}(G_{\alpha\to o}, @_{\alpha\to\beta\to\beta}, ()_{list}) &\equiv G \\
\mathit{Base}(G_{\alpha\to o}, @_{\alpha\to\beta\to\beta}, l_{list}) &\equiv \mathit{StabChain}(G, @, l) = \{id\}
\end{aligned}
$$

While for the above concepts we followed precisely the definitions used in the queries, we decided not to formalise coset representations as Schreier trees. A Schreier tree is used in GAP to allow efficient computation with coset representatives. This efficiency does not carry over when Schreier trees are formalised in Omega, since additional proof obligations for the data structure of trees appear and access functions (like 'return the elements of a path through the tree') have to be modeled explicitly. Therefore, we formalised a set of coset representatives as a discrete function which maps an orbit point $y \in xG$ to a permutation $g \in G$

[1] The terms ::, (), and *id* denote the list constructor, the empty list, and the identity permutation, respectively.

with $y = xg$. This function is sufficient for the application of Schreier's lemma and can easily be generated from a Schreier tree computed by GAP.

In the queries of Sec. 2 GAP is asked either to check a property (such as being element of a group), or to compute an object (such as a stabiliser) and check the computation. Whereas the problem formalisation for the first type is straightforward, it is not so obvious for the second type. In our formal system it is not possible to distinguish concrete from abstract terms on the object level, so the computation of the stabiliser expressed as $\exists x. x = G_c$ could be proved by inserting the formula G_c, and using reflexivity of equality. One way to overcome this problem is to use syntactic restrictions in the formalisation, e.g., $\exists x. \langle x \rangle = G_c$, to force the instantiation of x in the intended way. However a syntactic restriction can be seen as a meta-level description for something that is not expressible in the object language and any restriction can be circumvented[2]. Hence we decided to use a formalisation without syntactic restrictions but force the proof planner to introduce concrete objects to achieve the desired results.

4 Proof Planning and Computer Algebra

Proof planning [1] considers mathematical theorems as planning problems where an *initial partial plan* is composed of the *assumptions* and the theorem as an *open goal*. A proof plan is then constructed with the help of abstract planning steps, called *methods*, that are essentially partial specifications of tactics known from tactical theorem proving. In order to ensure correctness, proof plans must be executed to generate a sound calculus level proof. In the Omega system [4], *control rules* provide the possibility of introducing mathematical knowledge on how to proceed in the proof planning process by influencing the planner's behaviour at choice points (e.g., which goal to tackle next or which method to prefer) [12].

4.1 Hierarchical Proof Planning

Omega also allows for hierarchical proof planning. The basic idea is to generate proofs of different granularity by postponing the planning process for certain subgoals to a later point. In practice this is done by closing subgoals with *critical* methods without working out the particular subproof. While ordinary methods are *executed* solely via the tactic mechanism of the underlying theorem prover to generate a calculus level proof, critical methods have to be *expanded*, which yields new planning goals. These have to be entirely planned by the proof planner, which might lead, in case of failure, to backtracking in the overall proof plan.

In our examples, we use hierarchical proof planning to hide proofs of the more trivial problems listed in Sec. 2 when they occur as subproblems of more elaborate problems. For instance, the membership problems are immediately closed with a critical method and are only planned during expansion.

[2] Insert for x in $\exists x. \langle x \rangle = G_c$ a term *gen-of*(G_c) with a function *gen-of* that returns the generators of a group and continue the proof with abstract properties of *gen-of*.

4.2 Employing Computer Algebra in Proof Planning

We employ symbolic calculations to guide and simplify the search for proof plans for the group theory problems. There are several computer algebra systems (CAS) that can be connected with the Omega system but, in our examples, we use the GAP package [6]. In this paper, we are not concerned with the technical side of the integration since we exploit previous work, in particular [9] that presents the integration of computer algebra into proof planning and [13] that exemplifies how the correctness of certain limited computations of a large-scale CAS can be guaranteed within the proof planning framework. We concentrate rather on the use of CASs in the context of our examples.

We use symbolic calculations in two ways: (1) To guide the proof planner and prune the search space by computing hints with control rules. (2) To shorten and simplify the proofs by calling GAP within the application of a method. As a side-effect both cases can restrict possible instantiations of meta-variables[3].

We implemented (1) via the control rule `select-instance`. This rule is triggered by the introduction of a meta-variable as a substitute for the actual witness term of an existential variable. Meta-variables are usually introduced via methods or by decomposition of existentially quantified goals. After a meta-variable is introduced the control rule computes a hint with respect to the planning problem that is used as a restriction for this meta-variable. For instance, when showing the existence of an orbit for a point x the control rule supplies a hint as to what that orbit might be (this example is concretely explained in Sec. 5). To obtain suitable hints `select-instance` sends corresponding queries to GAP. If hints can be computed, the meta-variables are instantiated before the proof planning proceeds. However, the instantiations suggested by `select-instance` are treated as a hint by the proof planner; that is, they have to be verified during the subsequent proof planning process. In case the proof attempt fails for a particular instantiation, Omega backtracks and tries to find an appropriate instantiation by crude search. To avoid unnecessary computations and calls to GAP during a proof, `select-instance` keeps a record of already conducted computations and their results. `select-instance` is a generic control rule that has also been used in other contexts and with other CAS (see [11] for example).

The use of calculations for (2) is realised by three methods: two to simplify terms, by computing permutation applications or permutation compositions, and one to solve equations. In the latter case, for instance, if GAP can successfully solve an equation, the respective method is applied and the equational goal in question is closed. These computations are then considered correct for the rest of the proof planning process. However, once the proof plan is executed, GAP's computation is replaced by low level logic derivations to check its correctness. This is done with the help of a small self-tailored CAS that provides detailed information on its computations in order to construct the proof. The construction is achieved by executing Omega tactics, which is again a hierarchical process. In fact, during the execution process tactics can again contain calls to GAP, which

[3] Meta-variables are place-holders for terms whose actual form is computed at a later stage in the proof search. This is an example of middle-out reasoning (see [10]).

have to be in turn verified by execution of the calling tactics themselves. The overall process is extensively described in [13].

5 Planning the Subproblems

In this section, we describe how proof planning is used to formally verify correctness of the queries of Sec. 2. We do not use the GAP answers and certificates to guide the formal proof construction, but instead design the proof plan to work independently. That is, instead of simply implementing tactics for each query to construct the formal verifications, we developed a hierarchy of planning methods to search for a proof plan for each single query and use the constructed certificates when appropriate to guide the search. This has the advantage that, instead of a set of specialised tactics, we can design more general methods and leave the assembly of the proof to the planner. This is a more modular approach and enables the reuse of methods in other domains or when extending our domain to cover more problems. Indeed, our work shows that the proof planner can cope with successive expansions of the domain to new classes of theorems by a well chosen conservative extension of the set of methods.

Specifically, we use the following methods in the proofs:
6 basic methods that correspond to (generalised) natural deduction rules for quantifiers and connectives, methods for equality, and for definition expansion.
5 methods from set theory, most of them dealing with concrete sets. For example, In-Set will justify a line of the form $x \in \{x_1, \ldots, x_n\}$ when x is equal to one of the elements x_i and Foralli-Finite-Sort will reduce a goal of the form $\forall x{:}\{x_1,\ldots,x_n\}.\, P(x)$ (i.e. x is quantified over a finite domain) to the conjunction $P(x_1) \wedge \ldots \wedge P(x_n)$.
3 methods using GAP for the justification of a step, described in Sec. 4.2. Example: the method Eval-Permutation replaces the application of a permutation to a point by the result of this application computed with GAP.
5 methods from permutation group theory dealing with simple properties of permutations, generating sets, etc. Example: the goal that a permutation $a_1^{n_1} a_2^{n_2} \ldots, a_m^{n_m}$ is an element of $\langle A \rangle$ is reduced to the subgoals $a_i \in A$ for each $i \in 1, \ldots, n$ by the method Perm-by-Generators.
6 methods for computational objects containing the introduction of meta-variables. For example, the application of Schreier's lemma, where the orbit and the coset representation are introduced as meta-variables.
6 critical methods, for recurring proof obligations, that implement the hierarchical proof planning approach described in Sec. 4.1.

For most of the queries the constructed proof plans are similar to the proofs sketched in Sec. 2. We now describe the actual planning process for the proof of Example 4 in Sec. 2.3 concerning the orbit of 1 under the action of the Mathieu group $M{=}\langle$(1,10)(2,8)(3,11)(5,7), (1,4,7,6)(2,11,10,9)$\rangle$. A subpart of the resulting proof is given in Fig. 1. As mentioned in Sec. 3 the problem of computing the concrete set which is the orbit is formalised via existential quantification given in line *Thm.* The first method applied introduces a meta-variable M_O for the concrete set which is instantiated by `select-instance`. The hint from GAP

L_{22}.	$\forall Y{:}\{1,\ldots,11\}.(a_2@Y)\in\{1,\ldots,11\}$	(Orbit-Closed)
L_{21}.	$\forall Y{:}\{1,\ldots,11\}.(a_2@Y)\in\{1,\ldots,11\}$	(Orbit-Closed)
L_{20}.	$\forall Y{:}\{1,\ldots,11\}.(a_1@Y)\in\{1,\ldots,11\}\wedge$ $\forall Y{:}\{1,\ldots,11\}.(a_2@Y)\in\{1,\ldots,11\}$	(Andi* L_{21}, L_{22})
L_{19}.	$\forall Z{:}\{a_1,a_2\}, Y{:}\{1,\ldots,11\}.(Z@Y)\in\{1,\ldots,11\}$	(Foralli-Finite-Sort L_{20})
L_{18}.	$1\in\{1,10,11,2,3,4,5,6,7,8,9\}$	(In-Set)
L_{17}.	$\forall X{:}Orbit(\langle\{a_1,a_2\}\rangle,@,1).X\in\{1,2,3,4,5,6,7,8,9,10,11\}$	(Fixpoint L_{18}, L_{19})
L_3.	$Orbit(\langle\{a_1,a_2\}\rangle,@,1)\subseteq\{1,2,3,4,5,6,7,8,9,10,11\}$	(Defnexp L_{17})
L_{16}.	$11\in Orbit(\langle\{a_1,a_2\}\rangle,@,1)$ $\ldots$	(In-Orbit)
L_6.	$1\in Orbit(\langle\{a_1,a_2\}\rangle,@,1)$	(In-Orbit)
L_5.	$1\in Orbit(\langle\{a_1,a_2\}\rangle,@,1)\wedge\ldots\wedge 11\in Orbit(\langle\{a_1,a_2\}\rangle,@,1)$	(Andi* $L_6,\ldots,L_{16}$)
L_4.	$\forall X{:}\{1,2,3,4,5,6,7,8,9,10,11\}.X\in Orbit(\langle\{a_1,a_2\}\rangle,@,1)$	(Foralli-Finite-Sort L_5)
L_2.	$\{1,2,3,4,5,6,7,8,9,10,11\}\subseteq Orbit(\langle\{a_1,a_2\}\rangle,@,1)$	(Defnexp L_4)
L_1.	$M_O = Orbit(\langle\{a_1,a_2\}\rangle,@,1)$	(Subset-Equal L_2, L_3)
Thm.	$\exists O.O = Orbit(\langle\{a_1,a_2\}\rangle,@,1)$	(Existsi L_1)

$a_1 = \{(1,10),(2,8),(11,3),(5,7)\}$, $a_2 = \{(1,4,7,6),(10,9,2,11)\}$
Binding for meta-variable: $M_O \leftarrow \{1,\ldots,11\}$

Fig. 1. Orbit Proof

L_{26}.	$11 = 11$	(Reflex)
L_{25}.	$11 = (M_P@1)$	(Eval-Permutation L_{26})
L_{24}.	$M_P\in\langle\{a_1,a_2\}\rangle$	(In-Group)
L_{23}.	$\exists P{:}\langle\{a_1,a_2\}\rangle.11 = (P@1)$	(Existsi-In-Sort L_{24}, L_{25})
L_{16}.	$11\in Orbit(\langle\{a_1,a_2\}\rangle,@,1)$	(Defnexp L_{23})

Binding for meta-variable: $M_P \leftarrow \{(1,11,3,2,8,9,10,6,7,5,4)\}$

Fig. 2. Expansions of In-Orbit

is the set $\{1,2,3,4,5,6,7,8,9,10,11\}$ which is then shown to be equal to the orbit by double inclusion. The first direction, given in L_2, is to show that all the points of the computed set are included in the orbit. The reverse inclusion (L_3) is closed by a fixed-point argument. It suffices to show that 1 is in the set, and the set is invariant for the generators of G.

Lines $L_6,\ldots,L_{16}$ and L_{21}, L_{22} are justified by critical methods. The expansion of the critical method In-Orbit in line L_{16} is displayed in Fig. 2. The witness permutation, which maps 1 to 11 is again introduced as meta-variable M_P and instantiated with $(1,11,3,2,8,9,10,6,7,5,4)$ by a call of GAP. Line L_{24} contains the critical method In-Group, which justifies proof lines that correspond to the membership query of Sec. 2.1. The expansion of this method is given in Fig. 3, where the permutation is rewritten as the product ${a_2}^3 a_1$ of the generators a_1 and a_2. The method Equal-With-Gap calls GAP to justify equality of permutations. The proof for our example has 22 lines on the most abstract level, the expansion of all critical methods leads to a proof with 166 lines.

Further expansion results in a proof of over 10000 lines. This expansion of methods is achieved by tactic applications which eventually give a calculus level proof. For example, the equality in line L_{27} is established by checking that two permutations have the same functional behaviour. Since permutations act on a infinite set of letters, i.e. the natural numbers, checking equality formally is not straightforward. The important idea is that a permutation π only acts

L_{30}.	$a_1 \in \{a_1, a_2\}$	(In-Set)
L_{29}.	$a_2 \in \{a_1, a_2\}$	(In-Set)
L_{28}.	$({a_2}^3 * a_1) \in \langle \{a_1, a_2\} \rangle$	(Product-Of-Generators L_{29}, L_{30})
L_{27}.	$\{(1,11,3,2,8,9,10,6,7,5,4)\} = ({a_2}^3 * a_1)$	(Equal-With-GAP)
L_{24}.	$\{(1,11,3,2,8,9,10,6,7,5,4)\} \in \langle \{a_1, a_2\} \rangle$	(Re-Represent-With-Generators L_{27}, L_{28})

Fig. 3. Expansions of In-Group

nontrivially on a finite support set $\mathrm{supp}(\pi)$, i.e. on all the letters actually occurring in its cycles, and fixes the rest of $\mathbb{N}$. This fact is introduced as a theorem. In our example, in order to verify that $\forall x{:}\mathbb{N}.\, (1,7)(2,10)(4,6)(9,11)x = ((1,4,7,6)(2,11,10,9))^2\, x$ it suffices to show the equality for all $x \in \{1, 2, \ldots, 11\}$.

The formal certificates produced by proof planning are aligned with the argumentation given in the informal certificates, except for the queries for the stabiliser, nonmembership and order. In the case of the stabiliser, we do not use the Schreier trees as coset representation (see Sec. 3) and we use a formulation of Schreier's lemma for stabilisers, not arbitrary subgroups.

For the nonmembership proof we avoid the explicit introduction and verification of a concrete base—instead we added the following two methods. The first method reduces the goal $g \notin G$ to the new goal that bg is not in the orbit bG for some point b, if the planner succeeds we have a proof. In the other case, we try to find a permutation that is not an element of the stabiliser of G for a point b, but fixes b. This is done by the second method which introduces the new subgoals $M_P \in G$, $bM_P = bg$ and gM_P^{-1} is not in the stabiliser G_b for some point b where M_P is a meta-variable for a permutation. The stabiliser is also introduced as meta-variable M_S which has to be instantiated by a generating set, so the initial nonmembership goal is reduced to a new nonmembership goal $gM_P^{-1} \in M_S$ and the two methods can be applied again. For b the first element of the stabiliser base is chosen by a call to GAP within the methods. Since the correctness of the two methods is independent of the choice of b, we avoid additional proof obligations for the stabiliser base. In fact elements of the base only have to be introduced until the first method can be applied successfully.

For the order problem (see Sec. 2.8), we refrained from implementing the order lemma as a single, highly specialised, step. The necessary concrete sets of orbits $x_i G^{(i-1)}$ and stabilisers $G^{(i-1)}$ all depend on each other, which cannot be modelled with independent meta-variables. Instead, we opted for a more modular interplay of methods and computed hints by adding a method that applies the lemma $|G| = |bG||G_b|$ for the orbit bG and stabiliser G_b of a point b. The necessary order for the meta-variable instantiations is established by successive application of this method. The method uses the first base element of G for b and introduces two meta-variables for the stabiliser and the orbit, that have to be instantiated by concrete sets via the hint system. The method is applied successively until the stabiliser has order 1.

6 Evaluation

In order to test the robustness of our approach, we proved 1600 problems with Omega. We chose randomly a permutation and a generating set consisting of 2 or

4 permutations from the symmetric groups of 5 and 8 points. Then either proof obligation of the membership or nonmembership of this permutation was given to the proof planner. Whereas the proof of membership is relatively simple, the nonmembership contains the orbit and stabiliser queries as subgoals.

Generating set	Membership		Nonmembership				Average Order
			Unexpanded		Expanded		
	Length	Time	Length	Time	Length	Time	
2 Perm. of Sym_5	4.9	4.4	68.8	127.6	198.9	323.6	58.8
4 Perm. of Sym_5	6.1	10.4	88.9	279.1	360.5	561.8	112.6
2 Perm. of Sym_8	5.0	197.0	160.1	282.9	754.6	1347.8	25217.2
4 Perm. of Sym_8	6.9	203.2	233.7	369.1	1313.0	2274.5	37389.8

The table contains the average proof length of the constructed proofs and average runtime given in seconds. In the case of nonmembership we have two columns: one for unexpanded proofs, in which critical methods are contained as justification, and for expanded proofs, in which all critical methods were expanded to subproofs containing only non-critical methods. The last column shows the average order of the generated groups. The length of the membership proof depends on the number of different elements of the generating set that appear in the word for the permutation being tested. The length of nonmembership proofs depends on the applicability of the two additional methods for nonmembership queries described in Sec. 5. In the best case, only one of them needs to be applied to finish the proof successfully. The shortest proof for "2 Permutations of Sym_8" contains 18 lines.

The average runtimes include the socket communication between the GAP and Omega, and are therefore not quite accurate because they depend on the CPU load and the network traffic. We tried to compensate these effects by the large number of problems and multiple runs at different times. For the nonmembership proofs there is a correspondence between proof length and runtime. For membership proofs there is an increase in the runtime between permutations from Sym_5 and Sym_8, whereas the proof length is nearly constant. This is due to the fact that the witness terms are larger for Sym_8 and have to be communicated and parsed by GAP and Omega.

For comparison we repeated the test for some well-known groups. The groups were formalised by generating sets containing two permutations and we chose randomly 10 permutations that are members of the group, and 10 permutations which are not. The results are summerised in the following table:

Group generated by two permutations	Membership		Nonmembership				Order
			Unexpanded		Expanded		
	Length	Time	Length	Time	Length	Time	
Alternating group A_5	4.9	4.6	125.4	268.9	432.1	657.3	60
Alternating group A_8	5.0	39.5	238.9	413.2	1181.0	2083.2	20160
Mathieu group M_{11}	5.0	60.4	251.6	477.2	1641.4	2535.1	7920

7 Conclusion

There are various accounts of experiments combining computer algebra and theorem proving in the literature (see [8] for just a few). They generally deal with the technical and architectural aspects of integration, as well as with correctness issues. In particular, the skeptical approach we follow (i.e., CAS computations are only trusted if they can be fully justified with calculus level proofs in a proof checker) was introduced in [7] in the context of interactive tactical theorem proving. This skeptical approach has been automated in the context of proof planning [9] and extended to embed certain complex symbolic computations from arbitrary CASs into proof planning [13]. The work presented in this paper draws on experience of previous case studies, in particular [2] and [11].

In order to simply check the correctness of the GAP computations, it would suffice to provide a set of tactics that construct the corresponding formal proof for each computation as demonstrated in [9,2], for instance. However, the intention of our proof planning approach is to have an extensible and robust machinery that can react flexibly to failure by avoiding a tight coupling between computer algebra algorithms and tactics and that can be reused for other more complex problems. For example, this work can now serve as the basis for an approach to computational problems in graph theory, like showing two graphs are not isomorphic. Moreover, one goal of our work was to demonstrate that the information that is necessary to convince a human mathematician of the plausibility of a result is not only sufficient to certify the correctness formally but also to plan a corresponding proof problem independent of the actual computation. Our experiments show that this is possible and that the technique can be successfully applied to problems involving large and complex structures, as we can check mathematical data of a magnitude that is likely beyond the range of traditional theorem proving systems.

In our experiments Omega could construct proof plans for all examined problems. This is not surprising, since the design of the methods was aimed at covering all possible cases. But since there is no formal way of showing completeness at the method level this property can only be verified experimentally. Thus the proof planner should only fail if GAP provides an incorrect result. Even then the planner could theoretically find the correct instantiations by crude search, although this is not feasible in practice.

References

1. A. Bundy. The Use of Explicit Plans to Guide Inductive Proofs. In *Proc. of CADE–9*, LNCS 310, Springer, 1988.
2. O. Caprotti and M. Oostdijk. Formal and efficient primality proofs by use of computer algebra oracles. *J. of Symbolic Computation*, 32(1/2), 2001.
3. A. Cohen and S. Murray. An automated proof theory approach to computation with permutation groups. Tech. report, Dept. of Mathematics, TU Eindhoven, 2002.
4. J. Siekmann et al. Proof development with Omega. In *Proc. of CADE–18*, LNAI 2392. Springer, 2002.

5. A. Fiedler. *P.rex*: An interactive proof explainer. In *Proc. of IJCAR-2001*, LNAI 2083. Springer, 2001.
6. The GAP Group, Aachen, St Andrews. *GAP – Groups, Algorithms, and Programming, Version 4*, 1998. `http://www-gap.dcs.st-and.ac.uk/~gap`.
7. J. Harrison and L. Théry. A Skeptic's Approach to Combining HOL and Maple. *J. of Automated Reasoning*, 21(3), 1998.
8. D. Kapur and D. Wang, editors. *J. of Automated Reasoning— Special Issue on the Integration of Deduction and Symbolic Computation Systems*, volume 21(3). 1998.
9. M. Kerber, M. Kohlhase, and V. Sorge. Integrating Computer Algebra Into Proof Planning. *J. of Automated Reasoning*, 21(3), 1998.
10. I. Kraan, D. Basin, and A. Bundy. Middle-Out Reasoning for Logic Program Synthesis. Technical Report MPI-I-93-214, Max-Planck-Institut, Germany, 1993.
11. A. Meier, M. Pollet, and V. Sorge. Comparing Approaches to the Exploration of the Domain of Residue Classes. *J. of Symbolic Computation*, 34(4), 2002.
12. E. Melis and J. Siekmann. Knowledge-Based Proof Planning. *Artificial Intelligence*, 115(1), 1999.
13. V. Sorge. Non-Trivial Symbolic Computations in Proof Planning. In *Proc. of FROCOS 2000*, LNCS 1794. Springer, 2000.