Mechanical proofs of the Levi commutator problem

Maria Paola Bonacina *

Department of Computer Science
The University of Iowa
Iowa City, IA 52242-1419, USA
bonacina@cs.uiowa.edu

Abstract. This note presents purely mechanical proofs of the Levi commutator problem in group theory. The problem was solved first by using the theorem prover EQP, developed by William McCune at the Argonne National Laboratory. The fastest proof was found by using Peers-mcd, the Clause-Diffusion parallelization of EQP, developed by the author at the University of Iowa.

1 The Levi commutator problem

The Levi commutator problem is an equational problem in group theory. Given the axioms for a group with product $*$ and identity e

$$e * x \simeq x$$
$$x^{-1} * x \simeq e$$
$$(x * y) * z \simeq x * (y * z)$$

the *commutator* is a binary operator $[\_,\_]$ defined by:

$$[x, y] \simeq x^{-1} * y^{-1} * x * y.$$

The Levi commutator problem consists in proving that

$$x * [y, z] \simeq [y, z] * x \Leftrightarrow [[x, y], z] \simeq [x, [y, z]]$$

that is, $x * [y, z] \simeq [y, z] * x$ holds if an only if the commutator is associative. A textbook proof of this theorem can be found in [10].

In the input to the theorem provers, the group axioms and the commutator definition are written using prefix notation:

```
f(e,x) = x
f(g(x),x) = e
f(f(x,y),z) = f(x,f(y,z))
h(x,y) = f(g(x),f(g(y),f(x,y)))
```

* Supported in part by the National Science Foundation with grant CCR-97-01508.

where f is the product, g is the inverse, and h is the commutator. The theorem is broken into two parts, the implication $\Rightarrow$ and the implication $\Leftarrow$. The $\Rightarrow$ direction assumes

```
f(x,h(y,z)) = f(h(y,z),x)
h(h(a,b),c) != h(a,h(b,c))
```

where the second formula is the negation of the associativity of the commutator (i.e., there exist elements, denoted by the Skolem constants a, b and c, for which h is not associative). The $\Leftarrow$ direction assumes

```
h(h(x,y),z) = h(x,h(y,z))
f(a,h(b,c)) != f(h(b,c),a)
```

where the second formula is the negation of $f(x, h(y, z)) \simeq f(h(y, z), x)$ (i.e., there exist elements, denoted by the Skolem constants a, b and c, for which the property $f(x, h(y, z)) \simeq f(h(y, z), x)$ does not hold).

The $\Rightarrow$ direction is easy: it can be proved by Otter (version 3.0.4) [13] in `auto` mode (i.e., with the strategy chosen automatically by the prover) in 0.07 sec (hence 1 sec of wall-clock time; all run times in this paper are obtained on workstations HP B132L+ with 256M of memory). The $\Leftarrow$ direction is the challenge: a fully automated proof by Otter has not been obtained so far. The rest of this paper presents completely automatic proofs of this problem generated by the provers EQP and Peers-mcd.

To mention a bit of history, a different problem involving the commutator in group theory (proving that $x^3 \simeq e$ implies $h(h(x, y), y) \simeq e$) was a challenge problem in the early days of equational theorem proving [15, 1, 11].

2 The theorem prover EQP

EQP implements contraction-based strategies for equational reasoning (e.g., [5] for a general treatment), with associativity and commutativity (AC) built-in. The following summary does not include features (e.g., reasoning modulo AC) that were not used to solve the Levi commutator problem. A more complete description of EQP can be found in [14, 3].

Contraction-based strategies assume a well-founded ordering $\succ$ on terms, typically a *complete simplification ordering* [9]. In EQP, this ordering is a *recursive path ordering* [7], built from a total *precedence* on the function symbols. The user gives the precedence in the input file, and the prover completes it if it is not total. Function symbols have lexicographic status by default, and multiset status if specified in the input file. An input or generated equation $s \simeq t$ is oriented into $s \rightarrow t$, if $s \succ t$. If $s \# t$ (neither $s \succ t$ nor $t \succ s$), EQP "flips" the equation: it stores it as two pairs $s \simeq t$ and $t \simeq s$.

Similar to Otter, EQP has a default strategy that the user can modify by assigning values to parameters in the input file. The proofs of the Levi commutator problem were obtained by using the default inference system, which includes

paramodulation, simplification, and subsumption. EQP applies paramodulation to the left side of stored rewrite rules or equations. If $s\#t$, EQP considers for paramodulation s in $s \simeq t$ and t in $t \simeq s$. Simplification applies as simplifiers only the rewrite rules, not the equations. EQP inherited from Otter an inference rule called *deletion by weight*, which deletes all generated equations whose weight is larger than a maximum weight (parameter `max-weight`) given in the input file. Normally, the weight of a term is equal to the number of its symbols, and the weight of an equation is the sum of the weights of its sides. This inference rule obviously makes any strategy incomplete, but it is useful in practice.

EQP features two search plans, the *given-clause algorithm* and the *pair algorithm*. The given-clause algorithm is the same of Otter and it is the default search plan. It works with a list of clauses to be selected, called `sos` for historical reasons (the Set of Support strategy of [17]), and a list of clauses already selected, called `usable`, because these clauses can be used for inferences. It selects a given clause from `sos`, makes all expansion inferences between the given clause and the clauses in `usable`, process and appends to `sos` the non-trivial normal forms of all clauses thus generated, moves the given clause from `sos` to `usable`, and repeats.

The pair algorithm works on an *index* of all possible pairs of equations existing in the database (e.g., [6, 16, 12] for indexing techniques). It selects a pair from the index, performs all expansion inferences between the equations in the pair, if at least one of them belongs to `sos`, and repeats. The partition of the equations into `sos` and `usable` is relevant, because no inferences are performed on two equations in `usable`.

Contraction is handled in the same way regardless of whether the given-clause algorithm or pair algorithm is adopted. Each newly generated clause is normalized right after generation (*forward contraction*). If the clause is not deleted, its normal form is applied to contract other clauses (*backward contraction*).

By default, clauses/pairs to be selected are sorted by increasing length, so that the shortest one is picked next. This amounts to *best-first search* with the length of the clause as heuristic evaluation function. The `pick-given-ratio` parameter, also inherited from Otter, allows one to add some *breadth-first search*. If the value of `pick-given-ratio` is k, the search plan selects the oldest candidate every k choices.

3 The proofs by EQP

The first proof for the difficult half of the Levi commutator problem was found by EQP (version 0.9) with the following input (`%` precedes comments):

```
set(lrpo).           % use a lexicographic recursive path ordering
lex([a,b,c,e,f(x,x),g(x),h(x,x)]).           % set the precedence
set(para_pairs).          % use the pair algorithm as search plan
assign(max_mem, 80000).
assign(max_weight, 49).
```

```
assign(pick_given_ratio, 4).
end_of_commands.

list(usable).
f(a,h(b,c)) != f(h(b,c),a).          % Denial of conclusion
end_of_list.

list(sos).
f(e,x) = x.                          % Group axioms
f(g(x),x) = e.
f(f(x,y),z) = f(x,f(y,z)).
h(x,y) = f(g(x),f(g(y),f(x,y))).     % Definition of commutator
% Theorem: commutator is associative implies x*[y,z] = [y,z]*x.
h(h(x,y),z) = h(x,h(y,z)).           % Hypothesis
end_of_list.
```

The precedence $a \prec b \prec c \prec e \prec f \prec g \prec h$ was chosen because it orients the definition of commutator into a rewrite rule $h(x, y) \rightarrow f(g(x), f(g(y), f(x, y)))$. The input equations were put in `sos`, except the denial of the conclusion in `usable`, to obtain a pure forward-reasoning behaviour. Actually, putting the denial of the conclusion in `usable` is irrelevant, because it is simplified by the definition of commutator during the processing of the input, and the resulting normal form is placed in `sos`. Given this input, EQP found a contradiction after 148.96 sec and terminated in 163 sec of wall-clock time, after generating 96,219 equations, 9,657 of which were kept (that is, not deleted by contraction).

In other experiments the three group axioms were moved from `sos` to `usable`. This amounts to a strategy that is more oriented towards backward-reasoning, because it does not perform all the inferences between the axioms. With the input thus modified, a contradiction was found after 127.77 sec, and EQP terminated in 145 sec of wall clock time, after generating 96,846 equations and keeping 9,854 of them.

3.1 Tuning the parameters of the system to find a proof

Several parameters control a search by EQP. The parameter `max-mem` represents the maximum number of kilobytes that the system is allowed to allocate. Typically, one would start with a relatively low value, e.g., $20,000$, and raise it if the prover runs out of memory. The value $80,000$ is high for this parameter: it was chosen fairly early in the experimentation, in order to be sure to have plenty of memory. Not too surprisingly, it turned out to be unnecessarily high: the two proofs found with `max-weight` equal to 49 used $22,949$ and $23,437$ dynamically allocated kilobytes, respectively.

However, EQP runs out of memory with `max-mem` $80,000$, if no `max-weight` is set. For this parameter, there are two rules of thumb. If the prover runs out of memory, then decrease `max-weight`. If the prover terminates without finding a proof (EQP prints the message `no more inferences to make`), then increase

max-weight. The latter rule is based on the consideration that since deletion by weight is the only (with the default inference system) or the main source of incompleteness, the prover terminated without finding a proof because too many equations were deleted by weight. Assuming that the Levi commutator problem was very difficult, max-weight was initialized to 20, which is typically a low value for this parameter. Indeed, EQP terminated without finding a proof, regardless of whether the group axioms were in usable or sos. With max-weight equal to 35, EQP terminated without finding a proof, if the group axioms were in usable, and ran out of memory, if the group axioms were in sos. For max-weight equal to 36, 40, and 48, EQP ran out of memory, regardless of whether the group axioms were in usable or sos. Finally, EQP succeeded with max-weight 49. This behaviour of the max-weight parameter contradicts the first rule of thumb given above, because the prover was running out of memory and yet it was necessary to increase max-weight. A possible explanation is that a max-weight between 36 and 48 may cause EQP to discard by weight some important simplifier, so that it runs out of memory even if max-weight is smaller.

The parameter pick-given-ratio was assigned value 4 as a first guess, because it had worked well before; since a proof was found in a reasonable time with this value, no other values were tried.

3.2 Tuning the parameters of the system to improve performance

The above observation that it was necessary to increase max-weight in order to find a proof suggested to investigate what happens with max-weight greater than 49. Given the same input listed above, but with max-weight equal to 60, EQP generated an empty clause after only 60.28 sec, and terminated in 64 sec of wall-clock time, having generated 32,553 equations and kept 4,491 of them.

On the other hand, with max-weight 60, and the group axioms in usable, EQP did worse, finding a contradiction at 155.03 sec, and reporting 176 sec of wall-clock time, 75,534 equations generated, and 9,490 equations kept. A simple inspection of the proofs shows that rewrite rules from the canonical rewrite system for groups are included in both proofs: if the group axioms are in sos, these rewrite rules are derived right away, whereas they are derived later, if the group axioms are in usable. This may account for the different behaviour. Therefore, the pure forward-reasoning strategy with the more generous max-weight was the overall winner.

3.3 Comparison with a guided proof by Otter

Otter can be guided to find a proof for this problem by using an ad-hoc strategy: the precedence is lex([a,b,c,e,h(x,x),f(x,x),g(x)]), which orients the commutator definition backward ($f(g(x), f(g(y), f(x, y))) \rightarrow h(x, y)$); the max-weight is 20 (much lower than those afforded by the purely mechanical proofs); and the pick-given-ratio is 4 (same as in the purely mechanical proofs). The group axioms are in usable and everything else is in sos. Most important, the input file needs to include a list of "hints" in the form:

```
weight_list(purge_gen).
weight(h($(0),f($(0),h($(0),$(0)))), 100).
...
end_of_list.
```

The `weight` instruction above tells Otter to assign weight 100 to any term that matches the pattern $h(x_1, f(x_2, h(x_3, x_4)))$. Because `max-weight` is 20, all generated equations containing terms matching those in the `purge-gen` list are deleted. Since these patterns are provided by the user, this proof is not completely automatic.

Given this input, Otter found a contradiction at 316.08 sec, and terminated in 316.11 sec of user CPU time (425 sec of wall-clock time), after generating 871,524 equations, only 6,806 of which were kept. Note that the number of generated equations is one order of magnitude higher than in the EQP proofs.

It seems that the search plan with the pair algorithm may have played an important role in allowing EQP to find a completely mechanical proof: Otter does not have the pair algorithm, and EQP could not find a proof with the given-clause algorithm, regardless of whether the group axioms were in `usable` or in `sos`, and regardless of `max-weight` (e.g., it went out of memory with `max-weight` 49 and `max-weight` 60).

3.4 Presentation of the proofs

In forward-reasoning strategies the generated proof is made of the ancestors of the empty clause. In EQP, each clause is stored with its identifier and its "justification," that is, the name of the inference rule that generated it, and the identifiers of its parents. As soon as an empty clause is generated, the prover reconstructs the proof by listing first the empty clause, then its parents, then the parents of each parent and so on, until it reaches input clauses. Then, this list of clauses is printed with the input clauses first and the empty clause last. Each clause is printed on a separate line preceded by its identifier, weight, and justification. For instance, the 64 sec proof begins as follows:

```
1 (wt=11) [flip(1)] -(f(h(b,c),a) = f(a,h(b,c))).
2 (wt=5) [] f(e,x) = x.
3 (wt=6) [] f(g(x),x) = e.
4 (wt=11) [] f(f(x,y),z) = f(x,f(y,z)).
5 (wt=13) [] h(x,y) = f(g(x),f(g(y),f(x,y))).
6 (wt=23) [back_demod(1),demod([5,4,4,4,5])]
-(f(g(b),f(g(c),f(b,f(c,a)))) = f(a,f(g(b),f(g(c),f(b,c))))).
7 (wt=51) [demod([5,5,4,4,4,5,5])]
f(g(f(g(x),f(g(y),f(x,y)))),f(g(z),f(g(x),f(g(y),f(x,f(y,z)))))) =
f(g(x),f(g(f(g(y),f(g(z),f(y,z)))),f(x,f(g(y),f(g(z),f(y,z)))))).
8 (wt=51) [flip(7)]
f(g(x),f(g(f(g(y),f(g(z),f(y,z)))),f(x,f(g(y),f(g(z),f(y,z)))))) =
f(g(f(g(x),f(g(y),f(x,y)))),f(g(z),f(g(x),f(g(y),f(x,f(y,z)))))).
```

```
9 (wt=8) [para(3,4),demod([2]),flip(1)] f(g(x),f(x,y)) = y.
10 (wt=6) [para(2,9)] f(g(e),x) = x.
...
```

Input equations have empty justification, except equation 1 which is the result of flipping the denial of the conclusion. (Note that negation is denoted by the infix operator `!=` in the input, by the prefix operator `-` in the ouput.) Equation 6 is the result of simplifying 1 by equation 5, then 4, applied three times, and then 5 again. Because 1 was an existing equation, this is backward-simplification, as indicated by the code `back-demod`. Equation 7 is the result of normalizing associativity of the commutator, and equation 8 is the result of orienting 7. Equation 9 is generated by paramodulating 3 into 4, simplifying the result by 2 (forward simplification) and orienting it. Paramodulating 2 into 9 generates equation 10. The entire proof is made of 215 equations.

This proof was the fastest but not the shortest: the first proof by EQP (group axioms in `sos`, `max-weight` 49) contains 123 equations; the second proof (group axioms in `usable`, `max-weight` 49) has 193 equations; and the slowest one (group axioms in `usable`, `max-weight` 60) has 281. The guided proof by Otter includes 138 equations. Therefore, the pure forward-reasoning strategy with the more conservative `max-weight` generates the shortest proof. This instance confirms that proof length is not a suitable measure of complexity for theorem proving, because a shorter proof may take longer time.

4 The distributed theorem prover Peers-mcd

Peers-mcd is the parallelization of EQP, according to the Modified Clause-Diffusion method for distributed deduction. A Clause-Diffusion strategy launches concurrent, asynchronous, deductive processes to search in parallel the space of the problem. Each process executes a theorem-proving strategy, develops its own derivation, and builds its own database of clauses. The search space is subdivided among the processes, which cooperate to find a proof, and communicate by broadcasting the clauses they generate. As soon as one of them succeeds, all processes halt.

When Peers-mcd is invoked on a problem, it starts n processes, if n is the number of processes specified on the command line. Each process runs on a different workstation, and all processes execute the same code, which incorporates the code of EQP and uses MPI for message passing [8]. The workstations involved are specified by the user in a "procgroup" file. Process p_0 reads the input file and broadcasts to the other processes the input equations and parameters. Then, n processes $p_0, \ldots, p_{n-1}$ execute the strategy on the given problem.

A key issue for this type of distributed prover is how to subdivide the search space. Since the search space is infinite and unknown, the search plan of the strategy induces a subdivision of the search space by subdividing the clauses that generate it. Thus, the subdivision is built dynamically during the search. The component of the search plan that determines the subdivision is the *subdivision criterion*, implemented as an *allocation algorithm* to assign clauses to

processes. Since decisions are based on partial knowledge of the search space, the subdivision criteria are heuristic in nature.

Peers-mcd implements several subdivision criteria, including *ancestor-graph oriented* (AGO) criteria [3]. The general idea of these criteria is to use information in the *ancestor-graphs* of clauses to assign them to processes, in such a way to prevent the parallel searches from overlapping too much, at least in an intuitive sense.

The assignment of equations to processes determines the subdivision of the search space as follows. For paramodulation, a process executes a paramodulation step only if it owns the equation paramodulated into. For backward-contraction, Modified Clause-Diffusion distinguishes between deletion, such as in subsumption, and replacement, such as in simplification. The former is unrestricted. For the latter, if ψ can be simplified, only the owner of ψ generates its normal form. The other processes merely delete ψ.

Complete descriptions of Clause-Diffusion, Modified Clause-Diffusion and Peers-mcd can be found in [4], [2] and [3], respectively.

5 The proofs by Peers-mcd

In Peers-mcd, the user can select the subdivision criterion by assigning a value to the parameter `decide-owner-strat`. Therefore, Peers-mcd was given the same input as EQP, with one additional line:

```
assign(decide_owner_strat, 4).    % subdivision crit. para-parents
```

The criterion `para-parents` is an AGO criterion of type *parents*. These criteria have the property of assigning equations with the same parents to the same process. The intuition is that equations with the same parents are in the same neighborhood in the seach space: giving them to different processes might bring those processes to search in the same area, with consequent overlap and waste of resources. In order to make sure that equations with the same parents belong to the same process, it is sufficient to make the allocation algorithm a function of the identifiers of the parents: if the identifiers are the same, the destination process is the same. Different criteria may be defined depending on the notion of parents and the chosen function. The criterion `para-parents` only considers paramodulation parents, and uses addition modulo the number of processes. Thus, if φ was generated by paramodulating ψ_1 into ψ_2, or vice versa, φ is assigned to process p_k, where $k = id(\psi_1) + id(\psi_2)\ mod\ n$. If φ was generated by flipping φ', φ is assigned to the same process that owns φ'. If φ was generated by backward-contraction, or was an input equation, it is assigned to p_0.

Peers-mcd was tried first with `max-weight` 49. With this `max-weight`, EQP did better with the three group axioms in `usable`, finding a contradiction in 127.77 sec, and terminating after 145 sec of wall-clock time. Therefore, Peers-mcd also started with the axioms in `usable`. Using two processes, Peers-mcd found a contradiction after 71.43 sec, and terminated in 88 sec of wall-clock time, which represents a speed-up of 1.65 (ratio of wall-clock times) and efficiency 0.82. Also,

EQP generated 96,846 equations, keeping 9,854, whereas Peers-mcd generated only 38,126 equations, and kept 7,348 of them (these numbers for Peers-mcd are the sums of the numbers for the two processes). The proof by Peers-mcd is made of 123 equations, and is therefore shorter than the proof found by EQP with the same input (193 equations).

Then, the experiment was repeated with `max-weight` 60 and the group axioms in `sos`. This was the best configuration for EQP, which produced a contradiction in 60.28 sec, and halted in 64 sec of wall-clock time. With this input and two processes, Peers-mcd generated a contradiction in 22.51 sec, and halted in 27 sec of wall-clock time, yielding a super-linear speed-up of 2.37, with efficiency 1.18. The number of generated equations was $18,374$ ($2,831$ kept), significantly smaller than the $32,553$ ($4,491$ kept) reported by EQP. The distributed prover also found the shortest proof, made of only 88 equations, while the proof by EQP with this input included 215 equations, and the shortest of all EQP proofs listed 123 equations. Using more processes or other subdivision criteria did not improve the performance.

Both distributed proofs were found by process p_0. This may be related to the fact that criterion `para-parents` privileges p_0 by assigning it the equations generated by backward-contraction. However, this is no general rule (that is, using `para-parents` does not imply that p_0 finds the proof). Furthermore, both proofs contained equations of all kinds, generated by p_0 and belonging to p_0, generated by p_0 and belonging to p_1, generated by p_1 and belonging to p_0, and generated by p_1 and belonging to p_1, which shows that both processes contributed to the proofs. Actually, in distributed forward-reasoning strategies, a process may help another one also by doing work that is irrelevant for the proof, since the proof is such a small fraction of the generated search space.

In summary, for both inputs that were most successful for EQP, there is a configuration of Peers-mcd that can do better, and with super-linear speed-up in the best case.

Acknowledgements Thanks to William McCune for providing the input file for the guided Otter proof, after the first EQP proof was found.

References

1. W. W. Bledsoe. Non-resolution theorem proving. *Artificial Intelligence*, 9:1–35, 1977. Section written by D. Lankford.
2. M. P. Bonacina. On the reconstruction of proofs in distributed theorem proving: a modified Clause-Diffusion method. *J. of Symbolic Computation*, 21(4–6):507–522, 1996.
3. M. P. Bonacina. Experiments with subdivision of search in distributed theorem proving. In M. Hitz and E. Kaltofen, editors, *Proc. of PASCO-97*, pages 88–100. ACM Press, 1997.
4. M. P. Bonacina and J. Hsiang. The Clause-Diffusion methodology for distributed deduction. *Fundamenta Informaticae*, 24:177–207, 1995.

5. M. P. Bonacina and J. Hsiang. Towards a foundation of completion procedures as semidecision procedures. *Theoretical Computer Science*, 146:199–242, 1995.
6. J. D. Christian. Fast Knuth-Bendix completion: summary. In N. Dershowitz, editor, *Proc. of the 3rd RTA*, volume 355 of *LNCS*, pages 551–555. Springer Verlag, 1989.
7. N. Dershowitz. Termination of rewriting. *J. of Symbolic Computation*, 3(1 & 2):69–116, 1987.
8. W. Gropp and E. Lusk. User's guide for mpich, a portable implementation of MPI. Technical Report 96/6, MCS Division, Argonne Nat. Lab., 1996.
9. J. Hsiang and M. Rusinowitch. On word problems in equational theories. In T. Ottman, editor, *Proc. of the 14th ICALP*, volume 267 of *LNCS*, pages 54–71. Springer Verlag, 1987.
10. A. G. Kurosh. *The Theory of Groups*. Chelsea, New York, 1955.
11. E. L. Lusk and R. A. Overbeek. Reasoning about equality. *J. of Automated Reasoning*, 1:209–228, 1985.
12. W. McCune. Experiments with discrimination tree indexing and path indexing for term retrieval. *J. of Automated Reasoning*, 9(2):147–167, 1992.
13. W. McCune. Otter 3.0 reference manual and guide. Technical Report 94/6, MCS Division, Argonne Nat. Lab., 1994.
14. W. McCune. 33 Basic test problems: a practical evaluation of some paramodulation strategies. In R. Veroff, editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, pages 71–114. MIT Press, 1997.
15. A. J. Nevins. A human oriented logic for automatic theorem proving. *J. ACM*, 21:606–621, 1974.
16. M. E. Stickel. The path-indexing method for indexing terms. Technical Report 473, SRI International, 1989.
17. L. Wos, D. Carson, and G. Robinson. Efficiency and completeness of the set of support strategy in theorem proving. *Journal of the ACM*, 12:536–541, 1965.