ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-202

A Case Study in Automated Theorem Proving: A Difficult Problem about Commutators

by

William McCune

e-mail: mccune@mcs.anl.gov

Mathematics and Computer Science Division

Technical Memorandum No. 202

February 1995

This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Contents

Abstract **1**

1 Introduction **1**

2 The Search for a Proof **2**

2.1 Focus of the Search 4

2.2 Term Ordering 5

2.3 Useful Rewrite Rules 6

2.4 Memory Usage 6

3 A Proof **7**

4 Relation to a Conjecture of Padmanabhan **11**

5 Concluding Remarks **12**

References **12**

A Case Study in Automated Theorem Proving:
A Difficult Problem about Commutators

by

William McCune

Abstract

This paper shows how the automated deduction system OTTER was used to prove the group theory theorem

$$x^3 = e \Rightarrow [[[y, z], u], v] = e,$$

where e is the identity, and $[x, y]$ is the commutator $x'y'xy$. This is a difficult problem for automated provers, and several lengthy searches were run before a proof was found. Problem formulation and search strategy played a key role in the success. I believe that ours is the first automated proof of the theorem.

1 Introduction

The automated theorem prover OTTER [1] is not always easy to use. Version 3.0.0 of OTTER (and later versions) has an autonomous mode that allows the user simply to assert a denial of the theorem or conjecture; the program then formulates a simple strategy and searches for a refutation. The autonomous mode is sufficient for many easy theorems and some moderately difficult theorems, but more difficult theorems usually require some guidance from the user. Very little has been written about the kinds of guidance that can be given to the program and how that guidance is specified. This paper shows how such guidance was provided in one case, a difficult theorem about commutators in group theory.

Let GT stand for an axiomatization of group theory in terms of product, identity, and inverse, for example, $\{ex = x, x'x = e, (xy)z = x(yz)\}$. The focus of this paper is the theorem

$$\left\{ \begin{array}{l} \text{GT} \\ x^3 = e \end{array} \right\} \Rightarrow \{[[[y, z], u], v] = e\},$$

where $[x, y]$ is the commutator $x'y'xy$. (Note the similarity of this theorem to the benchmark theorem, usually called the commutator problem, in which the hypotheses are the same, but the conclusion is $[[y, z], z] = e$. That theorem, once considered a difficult challenge problem for automated theorem provers, is easily proved with OTTER in its autonomous mode and by other theorem provers based on paramodulation and rewriting.)

When I learned of the theorem, I had only it's statement. (In fact, I wasn't sure it was a theorem.) As I tried to get OTTER to prove the theorem, I supplied guidance toward finding a particular *type* of proof and against paths that I thought fruitless, rather than directing OTTER toward finding a particular proof.

This paper is intended for those who already have some familiarity with OTTER.

2 The Search for a Proof

The input file for the first OTTER search was the following.

```
set(knuth_bendix).

lex([e,A,B,C,D,_*_,g(_),h(_,_)]).

clear(print_kept).
clear(print_new_demod).
clear(print_back_demod).

assign(pick_given_ratio, 4).
assign(max_mem, 24000).

list(usable).
x = x.
end_of_list.

list(sos).
e * x = x.
g(x) * x = e.
(x * y) * z = x * (y * z).
h(x,y) = g(x) * (g(y) * (x * y)).
x * (x * x) = e.
h(h(h(A,B),C),D) != e.
end_of_list.
```

The flag `knuth_bendix` specifies a basic search strategy based on Knuth-Bendix completion, including the lexicographic recursive path ordering (LRPO) for orienting equalities and deciding which equalities are to be demodulators (rewrite rules). The command `lex([ ... ])` specifies an ordering on constant and function symbols (smallest first): `*` $\prec$ `h` and `g` $\prec$ `h` so that `h` is immediately eliminated from the denial (and from the search), and `*` $\prec$ `g` so that `g` is eliminated from the search when `g(x)=x*x` is derived.

The commands `clear(print_*)` disable some of OTTER's output; their purpose here is to save disk space. The command `assign(pick_given_ratio, 4)` specifies a ratio of 4:1 for selection of given clauses (clauses with which to make inferences): for each four clauses that are selected because they have the lowest weight, one clause is selected because it has been available for the longest time (that is, best-first:breadth-first search). The command `assign(max_mem, 24000)` limits memory usage to about 24 megabytes.

The clauses in `list(sos)` are the axioms for group theory, the definition of the commutator function `h(x,y)`, the special hypotheses $x^3 = e$, and the denial of the conclusion (`A`, `B`, `C`, and `D`, are Skolem constants, that is, elements for which the theorem fails to hold).

With this input file, OTTER quickly rewrites the denial, as expected, into

```
ABAABBCABAABBABAABBCCDABAABBCABAABBABAABBCCABAABBCABAABBABAABBCCDD!=e
```

(the product symbol is not shown, and right association is assumed) which has weight 133 (the default weight, which applies if no weight templates occur in the input, is a count of the number of constant, variable, function, and predicate symbols).

A scan of the output file indicated at least four problems with the search.

Focus of the Search. The high weight of the negative clauses delays their participation in the search. When new equalities are made into demodulators, all possible rewriting is performed, but more seems to be needed. In particular, an equality such as $xxyy = yxyx$ cannot be an ordinary rewrite rule (with LRPO or with RPO), so it must be applied with paramodulation. In order to apply it to another clause, the other clause must have been selected as given clause; negative clauses are rarely selected as given clauses, however, so many important inferences are delayed too long.

What OTTER clearly needs to address this problem is a better control mechanism that can be tailored to bidirectional search. The output file has two types of clause: (1) right-associated negative ground equalities (originating from the rewritten denial shown above) with product and Skolem constants on the left and e on the right, and (2) right-associated positive equalities in product and variables. We wish to reason forward, applying the positive equalities to positive equalities, and to reason backward, applying positive equalities to negative equalities. However, with OTTER's limited methods for selecting the given clause, we must usually focus on one or the other.

The Term Ordering. LRPO does not make enough equalities into rewrite rules. If we were to use RPO instead (i.e., give * multiset status), many of the equalities that fail to become rewrite rules under LRPO, for example, $xxyyxx = yxy$ (right association), would become useful rewrite rules. However, we wish to retain associativity, $(xy)z = x(yz)$, as a rewrite rule, and it cannot be so under RPO.

Useful Rewrite Rules. Since we wish to keep everything right associated, many equalities and rewrite rules do not apply where we wish them to. Consider, for example, the rewritten denial shown above and the equality $xxyy = yxyx$. We would like the equality to apply (by paramodulation) at 12 different places, but as things are, it applies only at the end.

Memory Usage. The available 24 megabytes was consumed within 37 minutes (on a SPARC 2), and the search stopped. At that point 11,195 clauses had been retained, 848 of those had become rewrite rules, and 119 clauses had been given (selected as the focus of attention). The vast majority of retained clauses were simply sitting in the `sos` list, wasting memory. (The only `sos` clauses that participate in the search are those that are also rewrite rules. Given clauses are selected from the `sos` list and moved to the `usable` list.) The standard solution is to set a maximum on the weight of retained clauses, but this becomes difficult because of our requirement for bidirectional search.

The next few sections describe some experiments designed to address the preceding problems.

2.1 Focus of the Search

I thought that there might exist the following type of bidirectional proof. Equalities are derived from $\{(xy)z = x(yz), ex = x, xxx = e\}$. The balanced equalities (both sides having the same weight) paramodulate into negative clauses, and rewrite equalities (i.e, the left side heavier) rewrite negative equalities, eventually deriving $e \neq e$.

To search for that type of proof, both positive and negative equalities must be selected as given clauses. The following approaches were considered.

- Simply use `assign(pick_given_ratio, 1)`. Since the negative clauses are much larger than the positive ones, half of the given clauses are the shortest available clauses (which are all positive) and the other half is a mixture of positive and negative clauses (oldest first). This approach was abandoned because it places too much emphasis on positive clauses, and no preference is given to *short* negative clauses.

- Adjust the weights of clauses, making the positive clauses heavier and the negative clauses lighter. This can be accomplished by including the following weight list in the input file.

```
weight_list(pick_and_purge).
weight(x,4).  % applies to all variables
weight(A,0).
weight(B,0).
weight(C,0).
weight(D,0).
end_of_list.
```

  Several weights for variables were tried before deciding to use 4. With weight 4 for variables, along with `assign(pick_given_ratio, 4)`, the search starts out mostly positive, but as the retained positive clauses become larger, the focus changes to negative clauses (which become shorter), with positive clauses entering occasionally because of a ratio of 4. This approach seemed promising.

- Separate the search into positive and negative parts. This involves making two OTTER runs. In the first run, the focus is exclusively on positive clauses; after some time, the run is stopped, and the positive clauses that had been given are collected and used as input for the second run. The second run is a search for a proof, in which the focus is exclusively on negative clauses, using the (fixed) set of positive clauses from the first run for paramodulation and rewriting. This approach was abandoned after several failures.

2.2 Term Ordering

To address this problem, we use OTTER's ad hoc term ordering to orient equalities and to decide which equalities are to be rewrite rules. The default ad hoc ordering says simply that for terms, $t_1 > t_2$ if t_1 has more symbols than t_2. Equalities are oriented, when possible, as *heavy=light*, and positive equalities whose left sides are heavier are made into rewrite rules. With this method, when associativity is a rewrite rule, and when the terms being rewritten are built from the binary function symbol, constants, and variables, rewriting will always terminate. This method was used for the rest of the experiments in this study; we can specify it with the command `clear(lrpo)`, placed after the command `set(knuth_bendix)`.

This solution does cause a secondary problem, however. The definition of commutator, `g(x)*g(y)*x*y = h(x,y)`, and equality `x*x = g(x)`, which is derived at the beginning of the search, will be oriented as shown; both are the wrong way for the type of proof we are seeking. The solution is simply to input the following list.

```
list(demodulators).
h(x,y) = g(x)*g(y)*x*y.
g(x) = x*x.
end_of_list.
```

(With the ad hoc ordering, input demodulators are not flipped.) This input list causes the denial to be rewritten on input into the form shown above; neither h nor g will appear thereafter in the search.

2.3 Useful Rewrite Rules

Recall that the equality $xxyy = yxyx$ applies only at the end of the rewritten denial. However, the trivial consequence $xxyyz = yxyxz$ applies at the other places of concern. Also, if we reformulate the rewritten denial from $t \neq e$ into $t * E \neq E$, where E is a new constant, the original equality $xxyy = yxyx$ is no longer needed. (The reformulated denial corresponds to the conclusion $[[[y,z],u],v] * w = w$, which clearly leads to an equivalent theorem.) This approach applies to both rewrite and nonrewrite (paramodulation) equalities.

Let us borrow from associative-commutative terminology and call $xxyyz = yxyxz$ the *extension* of $xxyy = yxyx$. Paramodulating an equality into associativity, then rewriting with associativity, produces the extension; hence, many of the extensions appear automatically. However, we don't need any nonextended equalities, and we can avoid them by simply starting with extended equalities only, because paramodulation of two extended equalities always produces an extended equality. In this case, we start with $xxxy = y$ instead of $xxx = e$. In addition, this approach eliminates the identity e from the search.

2.4 Memory Usage

The easiest way, and one of the most useful, to address the memory problem is to limit the size of kept clauses with the parameter max_weight. At this point, the weighting scheme of assigning variables weight 4 and Skolem constants weight 0 was being used. The rewritten denial has weight 67, and I was aiming for a proof in which the negative clauses "become smaller". I had no idea how big positive clauses would have to be; after several preliminary runs, I made a guess of weight 104, which allows positive clauses with up to 21 occurrences of variables. Assigning a weight limit obviously makes the search less complete, but it is frequently necessary in practice. If the search fails, one can easily raise the limit and try again.

Another way a lot of memory was saved was to adjust the indexing parameters. This requires considerable knowledge of the indexing method, and I'll present it in some detail, so that it might be more accessible to others. Indexing is used in five ways for this type of search.

- Paramodulation. This uses FPA/path indexing to find unifiable terms.
- Forward demodulation. This uses discrimination indexing to find demodulators.
- Forward subsumption. This uses discrimination indexing to find subsuming clauses.
- Back demodulation. This uses FPA/path indexing to find terms to demodulate.
- Back subsumption. This uses FPA/path indexing to find clauses to subsume.

OTTER's discrimination indexing is not adjustable, but we can limit the indexing depth for FPA/path indexing. In fact, because of the structure of the terms in these searches, FPA/path indexing filters out little or nothing, so disabling it saves vast amounts of memory (because terms are so deep) and a little bit of time. FPA/path indexing works by filtering out terms that fail to unify because of direct term structure (i.e., symbol clash). But our equalities are built from nothing more than variables and product, and our negative equalities are built from constants and product and are right associated. Consider paramodulation between two extended equalities; it cannot fail, so indexing can filter out nothing. Consider paramodulation between an extended equality and a right-associated ground equality; the only way it can fail is by indirect symbol clash, which cannot be filtered out by FPA/path indexing. Therefore, OTTER's indexing is useless for paramodulation.

A similar analysis shows that OTTER's indexing is useless for back demodulation and for back subsumption on positive clauses. But back subsumption on negative clauses does benefit from FPA/path indexing; in fact it is a *perfect* filter, because all of our negative clauses are ground. However, memory was judged to be a serious problem, and back subsumption is not called often because we keep relatively few clauses, so we simply disabled all FPA/path indexing (by setting the parameters `fpa_terms` and `fpa_literals` to 0).

3 A Proof

The following input file led to the first proof.

```
set(knuth_bendix).

lex([e,A,B,C,D,E,*(_,_),g(_),h(_,_)]).
```

```
clear(lrpo).

clear(print_kept).
clear(print_new_demod).
clear(print_back_demod).
clear(detailed_history).

assign(pick_given_ratio, 4).
assign(max_weight, 105).
assign(max_mem, 24000).

assign(fpa_literals, 0).
assign(fpa_terms, 0).

list(usable).
x = x.
end_of_list.

list(sos).
x*x*x*y = y.
(x*y)*z = x*y*z.
h(h(h(A,B),C),D)*E != E.
end_of_list.

list(demodulators).
h(x,y) = g(x)*g(y)*x*y.
g(x) = x*x.
(x*y=x*z) = (y=z).
end_of_list.

weight_list(pick_and_purge).
weight(x, 4).
weight(A, 0).
weight(B, 0).
weight(C, 0).
weight(D, 0).
weight(E, 0).
end_of_list.
```

The third clause in `list(demodulators)` applies left cancellation as a rewrite rule. It is used once (clause 129) in the proof below, but it is not necessary; other proofs have been found without it.

The following proof was produced with the preceding input file; OTTER took about 12 hours and used about 12 megabytes of memory on a SPARC 2 to find it.

Proof (found by Otter 3.0.3+ on altair.mcs.anl.gov at 43268.49 seconds).

1	$h(x,y) = g(x)g(y)xy$	
2	$g(x) = xx$	
3	$(xy = xz) = (y = z)$	
4	$x = x$	
6,5	$xxxy = y$	
8,7	$(xy)z = xyz$	
9	$h(h(h(A,B),C),D)E \neq E$	
10	$AABBABAABBABCCAABBABCAABBABAABBABCCAABBABCD$ $DAABBABAABBABCCAABBABCDE \neq E$	[copy,9 :1,2,2,8,8,1,2,8,8,8, 8,8,2,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,1,2,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8, 2,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8, 8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8, 8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8]
11	$xyxyxyz = z$	[$5 \to 7$:8,8, flip]
17,16	$xyxyxz = yyz$	[$5 \to 11$]
18	$xyxyzzu = yyzxzxu$	[$16 \to 16$]
21	$xyxyz = yyxxz$	[$5 \to 16$]
23,22	$xyzxyzxu = yzyzu$	[$7 \to 16$:8,8,8]
25	$xxyyz = yxyxz$	[flip 21]
26	$xyxzzyyu = yyxxzyzu$	[$21 \to 21$]
29	$xyzxyzu = yzyzxxu$	[$7 \to 21$:8,8,8]
31,30	$xxyyxxz = yxyz$	[$5 \to 21$, flip]
32	$xyzxyzu = zzxyxyu$	[$7 \to 21$:8,8,8]
34	$xyxyzzu = zxyzxyu$	[flip 29]
35	$xxyzyzu = yzxyzxu$	[flip 32]
40	$AABABBCCAABBABCAABBABAABBABCCAABBABCDDAABBA$ $BAABBABCCAABBABCDE \neq E$	[$21 \to 10$:23,17]
41	$xxyzzyyu = yxyxzyzu$	[$21 \to 25$]
42	$xyxyzxzxu = yyxzzu$	[$16 \to 25$, flip]
44	$xyxyzxzu = yyxzzxxu$	[flip 41]
55	$xyxzyzyu = yyxxyzzu$	[$25 \to 21$]
56	$xxyyxzzu = yxyzxzxu$	[flip 55]
58	$xxyyxzzxxu = yxyzxzu$	[$21 \to 30$]
63,62	$xxyzyzxxu = yzxyzu$	[$7 \to 30$:8,8,8]
81	$xyxyzuzuv = yyuxuxzzv$	[$25 \to 18$]
82	$xyzxyzuuv = yzyzuxuxv$	[$7 \to 18$:8,8,8]
83	$xyzxyzuuv = zzuxyuxyv$	[$7 \to 18$:8,8,8]
89	$xxyzuyzuv = zuxzuxyyv$	[flip 83]
92	$xyxyzzuxuxv = yyzxzxuuv$	[$18 \to 16$]
111	$xyxxyzxzu = yxyxzzxxu$	[$30 \to 22$]
119	$xyzxyxxu = yzyzzxzu$	[$16 \to 22$]
121,120	$xyzuxyzuxv = yzuyzuv$	[$7 \to 22$:8,8,8]
122	$xyxxyz = yxyxxz$	[$5 \to 22$]
124,123	$xyyxxyxz = yxyyz$	[$30 \to 22$:6]
129	$xyxyyz = yxyyxz$	[$16 \to 22$:3]
131	$xyxyzzyyu = yxyyxzyzu$	[flip 111]

134	$AABABBCCAABBABCBABBCACBBABCDDBABBCACBBABC$	
	$DE \neq E$	[40 :124,31,31,124,31,31]
135	$xyxyyzyu = zxyzxzzu$	[flip 119]
139	$xyxxyyxz = yyxyz$	[22 $\rightarrow$ 30 :6, flip]
154,153	$xxyyzuxzuxv = yxyzuzuv$	[22 $\rightarrow$ 21, flip]
194	$ABBAABCCAABBABCBABBCACBBABCDDBABBCACBBABC$	
	$DE \neq E$	[21 $\rightarrow$ 134]
214	$xyzuxyzuv = yzuyzuxxv$	[7 $\rightarrow$ 29 :8,8,8]
216,215	$xyxyzzyyu = zxyzxu$	[5 $\rightarrow$ 29, flip]
226	$xyyxz = yxxyz$	[5 $\rightarrow$ 29 :6]
238,237	$xyxxyzxzu = zyxzyu$	[131 :216, flip]
251	$xxyxyyz = yxyyxxz$	[29 $\rightarrow$ 30 :6]
266	$xyxxyyz = yyxyxxz$	[flip 251]
281	$xyzuxyzuv = uuxyzxyzv$	[7 $\rightarrow$ 32 :8,8,8]
311,310	$xyyxyzzyyu = yxxzyzu$	[30 $\rightarrow$ 226, flip]
319	$xyzzxyu = zxyxyzu$	[7 $\rightarrow$ 226 :8,8,8]
320	$xyxyyxz = yxxyxyz$	[226 $\rightarrow$ 226]
321	$xyzzyxyxu = yxxyzyxzu$	[32 $\rightarrow$ 226]
324	$xyyxzxyzxu = yxyzyzu$	[22 $\rightarrow$ 226, flip]
328	$xyzyzxu = yzxxyzu$	[7 $\rightarrow$ 226 :8,8,8]
339	$xyyxyxz = yxyxxyz$	[flip 320]
340	$xyyxzxyzu = yxzzxyxyu$	[flip 321]
348	$ABBAABCCABAABBCBABBCACBBABCDDBABBCACBBABC$	
	$DE \neq E$	[226 $\rightarrow$ 194]
352,351	$xyxxyxz = yxyz$	[226 $\rightarrow$ 30]
409	$ABBAABCCABBABACBABBCACBBABCDDBABBCACBBABC$	
	$DE \neq E$	[25 $\rightarrow$ 348]
418	$xyzxyzyyu = zzxyxu$	[26 $\rightarrow$ 34 :6, flip]
520	$xxyxxz = yyxyyz$	[226 $\rightarrow$ 42 :31]
534,533	$xyxyyxzzxxu = yxyyzxzu$	[44 $\rightarrow$ 32 :6]
576,575	$xxyyxyzxzu = yzyxzyu$	[58 $\rightarrow$ 32 :6,216]
578,577	$xyxxyyzxzu = yzyxzyu$	[58 $\rightarrow$ 29 :576,311, flip]
596,595	$xxyzuyzuxxv = yzuxyzuv$	[7 $\rightarrow$ 62 :8,8,8]
644,643	$xyxzxyzxu = yxzyzu$	[81 $\rightarrow$ 82 :63,17,6]
746	$xxyyxzxuuv = zyxzyuzuzv$	[92 $\rightarrow$ 62]
747	$xyzxyuxuxv = zzyyzxzuuv$	[flip 746]
806	$xyzxzuyuyv = yxyzyxzuuv$	[83 $\rightarrow$ 351 :154]
810,809	$xyzxxyzxu = yzxyzu$	[7 $\rightarrow$ 351 :8,8,8]
813	$xyxzxyzuuv = yxzyzuxuxv$	[flip 806]
918,917	$xyxyzyzzyu = zxyzxzu$	[129 $\rightarrow$ 129 :238]
1121	$xyzuyxzuyv = yyxyzuxzuv$	[89 $\rightarrow$ 139 :596]
1277,1276	$xyzyzxyzyu = yzxxzzu$	[16 $\rightarrow$ 319, flip]
1308	$ABBACBABACABABCDDBABBCACBBABCDE \neq E$	
		[319 $\rightarrow$ 409 :1277,534,121,352]
1372	$ABBACBABACBBAACDDBABBCACBBABCDE \neq E$	[21 $\rightarrow$ 1308]
1378	$ABBACAABBCBBAACDDBABBCACBBABCDE \neq E$	[21 $\rightarrow$ 1372]
1404	$ABBACAACCBCCAACDDBABBCACBBABCDE \neq E$	[520 $\rightarrow$ 1378]
1416	$ABBACAACCBACADCDCBABBCACBBABCDE \neq E$	[56 $\rightarrow$ 1404]
1453	$ABBCACAACBACADCDCBABBCACBBABCDE \neq E$	[122 $\rightarrow$ 1416]
1551,1550	$xyzuvxyzuvxw = yzuvyzuvw$	[7 $\rightarrow$ 120 :8,8,8]
1645	$xyxxzuxzuv = yyzuyxzuyv$	[120 $\rightarrow$ 16]

1691	$ACABCABACBACADCDCBABBCACBBABCDE \neq E$	[35 → 1453]
1910	$ACABCACCBABAADCDCBABBCACBBABCDE \neq E$	[32 → 1691]
2159,2158	$xyyxzxxyzu = yyzyzxu$	[214 → 226 :23, flip]
3080	$ACABCACCABAABDCDCBABBCACBBABCDE \neq E$	[129 → 1910]
3655,3654	$xyxzxzzyxu = yzxzzyu$	[328 → 324 :17,31, flip]
3880,3879	$xyzzyxyzzu = yyzxyxu$	[339 → 340 :918,6,17, flip]
4469,4468	$xyzxzzxyxu = yzxzzyu$	[418 → 281 :17,6]
4483	$ACBCACCBABDCDCBABBCACBBABCDE \neq E$	[3080 :4469]
4642	$ACBCACCBADCBBDCABBCACBBABCDE \neq E$	[328 → 4483]
4669	$ACBCACCBADCBBDBCACABCBBABCDE \neq E$	[319 → 4642]
4727	$ACBCACCBADCBBDBAACCBCBBABCDE \neq E$	[21 → 4669]
4749	$ACBCACCBADCBBDBAACACBACAACDE \neq E$	[135 → 4727]
4772	$ACBCACCBADCBBDBAACACBAACCADE \neq E$	[226 → 4749]
4795	$ACBCACCBADCBBDBACCAABAACCADE \neq E$	[21 → 4772]
4830	$ACBCACCBADCBBDBACCBBABBCCADE \neq E$	[520 → 4795]
4850	$ACBCACCBADCBBDBABCBCABBCCADE \neq E$	[25 → 4830]
4857	$ACBCACCBADCBBDBBCAABCBBCCADE \neq E$	[328 → 4850]
4867	$ACBCACCBADCBBDBBCAACCBCBBADE \neq E$	[266 → 4857]
4879	$ACBCACCBADCBBDCBCBACABCBBADE \neq E$	[41 → 4867]
4894	$ACBCACCBABDCDCBBCBACABCBBADE \neq E$	[319 → 4879]
4912	$ACBCACCBABBDCBDCCBACABCBBADE \neq E$	[34 → 4894]
9687,9686	$xyxzyxyzxzu = yzzyzyxu$	[1121 → 813 :644,3655,2159]
15798	$ACBCACCAADCABDABBABACBADE \neq E$	[1645 → 4912 :9687]
15805	$E \neq E$	[747 → 15798 :578,3880,6,1551,810,23,17,6]
15806	□	[15805,4]

4 Relation to a Conjecture of Padmanabhan

Conjecture (R. Padmanabhan [2]). *Let $A = \{a_1, a_2, \cdots, a_n\}$ and $\{a\}$ be identities in the language of one binary operation. If $A \Rightarrow a$ in group theory, then $A \Rightarrow a$ in cancellative semigroups (CS) as well.*

The proof in the preceding section supports the conjecture. For cancellative semigroups, the statement corresponding to the focal theorem of this paper is

$$\{\mathrm{CS}, x' = xx, xxx = yyy\} \Rightarrow \{[[[x,y],z],w] = uuu\}. \tag{1}$$

What Otter actually proved is

$$\{\mathrm{CS}, x' = xx, xxxy = y\} \Rightarrow \{[[[x,y],z],w]u = u\}. \tag{2}$$

Statement 1 follows easily from 2, because $\{\mathrm{CS}, xxx = yyy\} \Rightarrow \{xxxy = y\}$.

5 Concluding Remarks

For these experiments, I ran about 20 OTTER searches, modifying the formulation and search strategy for each based on results of the previous searches. In providing the guidance, I used only fairly well-understood and fairly well-defined knowledge about OTTER and search strategies, rather than knowledge about a particular proof or general knowledge of mathematics; therefore there is hope that some of the methods described in this paper can be automated. Such automation would be an advance toward the goal of self-analytical theorem provers, advocated by Larry Wos [3].

OTTER clearly needs better features for control of bidirectional search. We were able to achieve an effective bidirectional search for this problem by adjusting the weights: the first part of the search focused on positive clauses, then shorter negative clauses were derived, then the second part of the search focused on negative clauses. But few bidirectional searches have such a smooth and natural transition. In general, we need a true dual-focus (or more-part focus) search.

Finally, perhaps the strategies used for this problem can be shown to be complete for a useful class of problems.

References

[1] W. McCune. OTTER 3.0 Reference Manual and Guide. Tech. Report ANL-94/6, Argonne National Laboratory, Argonne, Ill., 1994.

[2] R. Padmanabhan, Electronic mail to W. McCune, May 7, 1993.

[3] L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning: Introduction and Applications,* revised edition. McGraw-Hill, New York, 1992.