# Chapter 2

# Otter and MACE

OTTER [6] is a program that searches for proofs, and MACE [5] is a program that searches for small finite counterexamples. Both apply to statements in first-order logic with equality. OTTER is more powerful and flexible at its task than MACE is at its task (consequently, OTTER is more difficult to use). When searching for a proof with OTTER the user typically formulates a search strategy and makes several attempts, modifying the strategy along the way. When searching for a counterexample with MACE, the user simply supplies a statement of the conjecture; if the conjecture is not too complex, and if there exist small finite models, then MACE will find them. For many of our conjectures (quasigroup problems in particular), we know that counterexamples, if they exist, must be infinite; MACE is useless in such cases. But the two programs nicely complement one another in many other cases.

Version 3.0.?? of OTTER and version 1.0.?? of MACE were used for the experiments presented in this document. Both programs are in the public domain and are available by anonymous FTP. (All of the input files are also available by FTP.) See ftp://info.mcs.anl.gov/pub/Otter/README for information on obtaining the programs. The primary documentation for the programs are [6] and [5]; these are included with the programs when obtained by FTP.

The following descriptions of the two programs are informal. See the manuals and [13] for more formal and detailed presentations. First, we define some terms of automated theorem proving.

## 2.1 Definitions

These definitions are biased toward OTTER and equational theorem proving.

A *term* is either a variable, a constant, or the application of an $n$-ary function symbol to $n$ terms.

An *atom* is the application of an $n$-ary predicate symbol to $n$ terms. Nearly all atoms in this work will be equalities.

A *literal* is either an atom or the negation of an atom.

A *clause* is a disjunction of literals. The variables in a clause are implicitly universally quantified.

A *unit clause* contains exactly one literal. Most of the clauses in this work are unit equality clauses, that is, equations.

The *empty clause* contains no literals and represents a contradiction.

A *Horn clause* has at most one positive literal. All of the clauses associated with this work are Horn clauses.

Two terms or atoms *unify* if there is a substitution of terms for variables that makes the terms or atoms identical. When we speak of a *unifying substitution*, we refer to the most general unifying substitution.

An *equational deduction rule* is a non-unit clause containing some number, say $n$, of negative equalities (the antecedents) and one positive equality (the consequent). It is frequently written as an implication,

$$s_1 = t_1 \ \& \ \cdots \ \& \ s_n = t_n \ \rightarrow \ s = t,$$

and is typically used by OTTER with the inference rule hyperresolution.

The inference rule *hyperresolution* (for equational deduction rules) takes an equational deduction rule with $n$ antecedents and $n$ equality unit clauses, and derives, if possible, the appropriate instance of the consequent.

*Paramodulation* (for unit equality clauses) is an inference rule that combines variable instantiation (by unification) and equality substitution into one step. Consider $t = s$ and let $P[t']$ be a positive or negative equality containing a term $t'$. If $t$ and $t'$ unify, with unifying substitution $\sigma$, then paramodulation *from* $t = s$ *into* $P[t']$ yields the paramodulant $\sigma(P[s])$. For example, from $uu' = e$ into $x(x'y) = y$, we can derive $xe = x''$ by unifying $uu'$ and $x'y$.

*Unit conflict* is a trivial inference rule the derives the empty clause (a contradiction) from two unifying unit clauses of opposite sign. All OTTER proofs are by contradiction.

A *demodulator* is a positive equality unit, $t_1 = t_2$, that is used as a rewrite rule to rewrite instances of $t_1$ to the corresponding instances of $t_2$.

*Demodulation* is the process of repeatedly rewriting a clause with a set of demodulators until no more rewriting steps can be applied. A desirable property of a set of demodulators is that demodulation with the set terminates on all terms to which it is applied.

*Forward demodulation* is demodulation applied to newly inferred clauses. OTTER always performs forward demodulation when demodulators are present.

*Back demodulation* occurs when a newly inferred equality is added to the set of demodulators. All clauses that can be rewritten with the new demodulator are demodulated with the (extended) set of demodulators. Back demodulation is optional in OTTER.

*Skolemization* is a process of eliminating existentially quantified variables from a formula by replacing them with *Skolem constants* and *Skolem functions*. The resulting formula is equiconsistent with the original formula. In this work, Skolemization is used mainly to assert the denials of conclusions. For example, when proving commutativity, the conclusion is $\forall x \forall y (xy = yx)$, its negation is

$\exists x \exists y(xy \neq yx)$, and the Skolemized negation is $AB \neq BA$, where $A$ and $B$ are new symbols, that is, Skolem constants. In other words, to deny the conclusion, we simply assert that there are two elements, $A$ and $B$, that do not commute.

*Knuth-Bendix completion* is a method for attempting to transform a set of equalities into a *complete set of reductions*, that is, a set of rewrite rules that produces canonical forms. Although we are not concerned in this work with complete sets of reductions, the mechanics of some of our search strategies are similar to Knuth-Bendix completion. See Sec. 2.2.3.

## 2.2 Otter

OTTER is a resolution/paramodulation theorem-proving program that applies to statements in first-order logic with equality. OTTER operates on clauses, so if the statements supplied by the user are not clauses, OTTER immediately translates them with Skolemization (to eliminate existential quantifiers) and conjunctive normal form (CNF) conversion. This work focuses mainly on equational logic, and equations are clauses already, so CNF conversion is not necessary. We search for proofs by contradiction, and Skolemization is used here mainly to deny the conclusion of the conjecture.

OTTER is not interactive; the user prepares an input file which contains a denial of the conjecture and a specification of a search strategy, then runs OTTER and receives an output file. (OTTER has a primitive interactive mode, but it was not used for this work.) If the output file does not contain a proof, the user examines the output, modifies the input, and tries again for a proof. Most trivial theorems are proved within the first few attempts, and most difficult theorems require some iteration, resulting in the development of a specialized search strategy. For this work, we did not use OTTER as a proof checker; that is, the strategies were not designed to lead OTTER directly to particular proofs. Rather, the strategies were designed to guide the program toward a particular *type* of proof or to avoid search paths that seem to be worthless.

Because of the iterative way in which OTTER is typically used, we do not think of it as an "automatic" theorem prover. However, OTTER does have a mode, the *autonomous mode*, that allows the user to supply simply a denial of the conjecture; the program does a simple syntactic analysis of the conjecture, decides on a simple strategy, and searches for a proof. Although not used in this work, the autonomous mode can be used by OTTER to prove many of the easy theorems we present.

The real, practical strength of OTTER is the ability to quickly explore large search spaces. Strategies are used mainly to restrict the search rather than to carefully guide it, resulting in a search less focused than with some of the other automated theorem provers, which use heuristics to carefully plan and control the search for a proof.

The theorems we present fall into two main classes: ordinary equational theorems, and equational theorems with the rule $=(gL)\Rightarrow$. In some cases we also add deduction rules such as cancellation or various closure conditions

(Sec ??). The strategies we use for ordinary equational theorems (both with and without deduction rules) are simpler, more uniform, and better understood than our strategies for $=\!\!(gL)\!\!\Rightarrow$ ; before going into detail about particular strategies in Sections 2.2.3 and 2.2.4, we present some general features of OTTER.

### 2.2.1 The Main Loop

OTTER maintains three primary lists of clauses.

`usable.` These clauses are able to participate in the search through application of inference rules.

`sos` (set of support). These clauses are waiting to participate in the search through application of inference rules. A subset (possibly empty) of `sos` occurs also in list `demodulators`; members of that subset can participate in the search through rewriting (demodulation).

`demodulators.` All of these clauses are equalities. They are applied as rewrite rules to all inferred clauses.

In the input file, the user typically partitions the statement of the conjecture into `usable` and `sos`; if any equalities are present, they may be inserted into `demodulators` also. OTTER's highest-level operation is a simple loop:

**Repeat**
1. Select a clause, the *given clause*, from `sos`.
2. Move the given clause from `sos` to `usable`.
3. Draw inferences from the given clause using other clauses in `usable`.
   With each inferred clause, rewrite it with `demodulators` and
   decide if it should be retained; if retained, append it so `sos`,
   and if it should be a rewrite rule, append it to `demodulators`.

**Until** a proof is found or a resource is exhausted.

The five main determinants of the search are (1) the initial partition into `usable` and `sos`, (2) the method for selecting the given clause, (3) the inference rules that are applied, (4) the rules for retention of inferred clauses, and (5) the rules for deciding whether inferred equalities are to be rewrite rules. These are considered to be part of the search strategy and are covered in some detail in the following subsection.

### 2.2.2 General Strategies

As mentioned above, use of OTTER frequently involves iteration through a sequence of searches, and part of the iteration is toward achieving a *well-behaved* search. The notion of a well-behaved search is not precise, and recognizing one is largely a matter of experience in finding proofs with OTTER. However, we can list here some ways in which searches can be ill behaved. One determines the behavior of a search by examining the output file.

- The `sos` list grows to rapidly. This wastes time and memory, because most `sos` clauses will never participate in the search.
- The program focuses on many trivial variants of clauses, because they are smaller than other, useful clauses.
- The program is not using, or not using to the desired extent, some of the inference rules that are enabled.
- A bidirectional search is desired, but the given clauses that are selected force the search to be in one direction only.
- The program is wasting time making checks or performing other operations that have little or no effect on the search.

To adjust the behavior of a search, one changes the specification of one or more strategies. The rest of this subsection contains summaries of OTTER's most important general strategies.

### Set of Support

One of the earliest strategies for automated theorem proving is the set of support strategy [14], which is designed to prevent lines of reasoning. The set of support requires that the input clauses be partitioned into two sets, $S$ (clauses with support) and $T$ (those without support); the strategy requires that all lines of reasoning start with a member of $S$. That is, no inference is drawn entirely from clauses in $T$.

OTTER's main loop can be viewed as a simple implementation of the set of support strategy. The initial `sos` list is $S$, and the initial `usable` list is $T$. Because the clauses in the initial `usable` list never occur in the `sos` list, and because each inference is started with the given clause, which is selected from the `sos list`, no inference is made in which all of the participants are in the initial `usable` list. If the user wishes to do without the set of support strategy, then he or she simply places all input clauses in the `sos` list.

The primary motivation for the set of support strategy is to focus the reasoning on the problem at hand instead of on the general theory. Many theorems formulate naturally as axioms of a theory, special hypotheses, and a conclusion. The standard recommendation for partitioning the input into $S$ (supported clauses) and $T$ (non-supported clauses) is to place the axioms of the theory into $T$ and clauses corresponding to the special hypotheses and denial of the conclusion into $S$. The result is that all lines of reasoning start with the hypotheses and the conclusion.

However, our experience has shown that the set of support strategy is not particularly effective for two classes of theory: (1) equational and nearly-equational theories, and (2) theories with few axioms and few concepts. The theories on which we focus in this work fall into both classes, so for the most part, we do without the set of support strategy.

### Selecting the Given Clause

Selection of the given clause has a great effect on the character and outcome of the search; it is the "next path to explore". Since new clauses are appended to the `sos` list, selecting the first `sos` clause leads to breadth-first search (occasionally useful), and taking the last leads to depth-first search (rarely useful). OTTER's default method, however, is to select the best clause in the `sos` list as the given clause. For example, if a simple identity is derived, it is usually best to use it right away. The default measure of "best" , and the one used for this work, is simply the length of the clause; if the user has other ideas about what is best, he or she can use OTTER's weighting mechanism to specify rules for assigning weights to clauses.

For most of the theorems and conjectures in this work, we use a selection method that is a combination of best-first and breadth-first. One of OTTER's parameters, `pick_given_ratio`, can be used to specify the ratio; a value of $n$ means $n$ parts best-first to 1 part breadth-first; that is, through $n$ iterations of the main loop, the best clause is taken, then in the next iteration, the first is used, and so on. With this method, OTTER focuses mainly on the small clauses, while occasionally allowing a larger, richer clause to participate in the search, to "mix things up".

### Demodulation (Rewriting)

The list `demodulators` contains a set of directed equalities that are used as rewrite rules. When a new clause is inferred, one of the first processes applied to it is forward demodulation, in which the clause is rewritten, as much as possible, with the members of `demodulators`. The primary motivation for demodulation is to reduce redundancy by canonicalizing clauses. For example, if the theory has a left identity, $ex = x$, then it is usually wasteful to retain both $(xy)z = x(yz)$ and $(xy)(ez) = x(yz)$; with $ex = x$ as a demodulator, the second will be demodulated to an equality identical to the first and will be deleted.

Equalities can become demodulators in three ways: (1) demodulators can be specified in the input file, (2) input equalities in `sos` or `usable` can be copied into `demodulators`, and (3) derived equalities can be copied into `demodulators`. Alternatives (2) and (3), which apply only if certain flags are set, can also cause the new demodulator to rewrite previously retained clauses, through back demodulation.

### Retaining Inferred Clauses

After a newly inferred clause is demodulated and possibly rewritten in other ways, OTTER must decide whether it is to be retained. Forward subsumption is nearly always applied; that is, if the new clause is subsumed by any of OTTER's existing clauses, then it is discarded. Also, the user can assign limits on several measures, including number of variables, number of literals, and the weight of clause. A limit on the weight of retained clauses is nearly always used for difficult theorems.

As when selecting the given clause, the default weight (which we use unless stated otherwise) is the length of new clause. If there is no limit on the weight of retained clauses, or if the limit is too high, then the `sos` list will grow very rapidly; most of the `sos` clauses will simply be wasting space, with little chance of participating in the search. Also, if back demodulation is enabled, when newly derived equalities are adjoined as demodulators, a lot of time will be wasted attempting to rewrite all of the `sos` clauses. If the limit is too low, then the `sos` list will be exhausted and the search will terminate. The typical strategy, and the one used in this work, is to start with a low limit and make several test runs, adjusting the limit, until a well-behaved search is achieved. (This iterative process explains the apparently arbitrary weight limits specified in many of the input files associated with this work.)

### Forward, Backward, and Bidirectional Searches

Most of the theorems and conjectures in this work fit the template

$$E \cup D \Rightarrow C,$$

where $E$ and $C$ are sets of equalities, $D$ is a set of deduction rules, and "$\Rightarrow$" may include an extension to the theory.

In a forward search, one reasons from $E \cup D$, and derives additional equalities, succeeding if all members of $C$ are derived. An easy (but possibly risky) way to have OTTER conduct a forward search is to place the denials of the conclusions $C$ in an auxiliary list called `passive`. Clauses in `passive` do not participate actively in the search, but if an equality that conflicts with a member of `passive`, then a proof will be reported and printed. This method is risky, however, if the members of $C$ are complex enough to be rewritten by a demodulator. For example, suppose $e$ is a constant, the denial of $(ex)x = yy$ is in `passive`, and the equality $ex = x$ is derived and adjoined as a demodulator. If $(ex)x = yy$ is derived later, it will be demodulated to $xx = yy$ before the unit conflict check with passive, and the proof will be missed. Therefore, only non-simplifiable clauses should be placed in `passive`.

In a backward search, one reasons from the denial of $C$ (assume $C$ is a singleton), using members of $E$ and $D$ to derive negated equalities, until a contradiction with a member of $E$ is derived. To have OTTER conduct a backward search one can simply start the search with $E \cup D$ in list `usable` and the denial of $C$ in `sos`. All derived clauses will be negative. (If $D$ is nonempty, then they will have to be used to derive negative clauses with the inference rule UR-resolution [13].) *However, purely backward searches are ineffective for nearly all theories in which equality plays a major role*; deriving and using equalities is simply too important.

In a bidirectional search, one reasons both forward and backward. To achieve a bidirectional search with OTTER, one can place a subset of $E$ in the list `sos`; both positive and negative equalities can be derived. However, it can be difficult to achieve the desired balance between forward and backward search, because

OTTER uses a single method for selecting the given clause, and both positive and negative clauses must be selected (see Sec. ?? for an example).

Most of the searches and most of the proofs in this work are either forward or mostly-forward bidirectional. If $C$ was a simple singleton equality, its denial was usually placed in the list `passive`, resulting in a forward search (proof); otherwise, its denial was placed in list `usable`, resulting in a forward or bidirectional search (proof).

**Problem Formulation**

Although it is arguable whether problem formulation is a search strategy, from a practical view, it falls in with the other strategies presented in this section.

One usually has choices in the statement of the theorem or conjecture, and OTTER can be sensitive to these choices. We list three examples. First, given an equational theory $E$, which axiomatization of $E$ should be used? We usually find it best to use an axiomatization that includes simple identities; in some cases we prefer to use single axioms.

Second, should dependent identities be included? Consider group theory in terms of product, inverse, and identity $e$. Three logically equivalent alternatives are the following.

$$\{ex = x,\ x'x = x,\ (xy)z = x(yz)\}$$
$$\{ex = x,\ xe = e,\ x'x = x,\ xx' = e,\ (xy)z = x(yz)\}$$
$$\{ex = x,\ xe = e,\ x'x = x,\ xx' = e,\ (xy)z = x(yz),\ e' = e,\ x'' = x,\ (xy)' = y'x',\ x(x'y) = y,\ x'(xy) = y\}$$

In addition, should cancellation be included as a deduction rule? We don't have easy answers; it usually depends on the situation and requires some experimentation to achieve a well-behaved search.

Finally, consider the following two forms of the deduction rule for left cancellation:

$$xy = xz \ \rightarrow\ y = z$$
$$xy = u \ \&\ xz = u \ \rightarrow\ y = z$$

The first form, although more natural, is less effective than the second. Both rules apply to the identity $ab = ac$ (assuming the presence of $x = x$ for use with the second rule), but only the second rule applies to the pair $\{ab = d, ac = d\}$.

### 2.2.3 Equational Theorems

Many of our proof searches in equational theories use strategies that are similar to the Knuth-Bendix completion method [2]. Because the method is well known, we review it here and state how it differs from our OTTER strategies.

**Knuth-Bendix completion**

*Knuth-Bendix completion* is a method for attempting to transform a set of equalities $E$ in to an equivalent set of rewrite rules $R$ such that rewriting with $R$ terminates on all terms and produces a canonical form. Rewriting with $R$ is a decision procedure for equality of terms in the theory of $E$. (Thus, the existence of $R$ is undecidable.) The set $R$ is called a *complete set of reductions*. The method uses an ordering, say $\succ$, on terms to orient equalities into rewrite rules and to guarantee that rewriting terminates; each member of $R$ satisfies *left-side* $\succ$ *right-side*. The following summarizes the method.

1. All equalities, whether members of $E$ or derived, must be orientable into rules as *left-side* $\succ$ *right-side*; otherwise the method fails.
2. New equalities are inferred by paramodulation from left sides of rules into right sides of rules, then simplified with the current set of rules.
3. When a new (simplified) equality is derived and oriented into a rule, it is used to rewrite all known rules.
4. The method succeeds if no new paramodulants (left into left) can be derived.

If the method fails, then a different term ordering may lead to success.

(Some history: The roots of our current equational rules strategies are in the methods used by Larry Wos and Ross Overbeek in the late 1960's and early 1970's. Those methods were developed independently from the Knuth-Bendix method (at about the same time) and were aimed at proving theorems rather than at finding a complete set of reductions.)

The OTTER strategy we typically use for equational theorems is quite similar to the Knuth-Bendix method; the main differences are (1) our goal is to prove theorems, (2) instead of failing when a non-orientable equality is derived, we continue, using if for paramodulation, but not for rewriting, (3) we typically impose a limit on the size of retained clauses, and (4) we can paramodulate into negative clauses as well as positive ones, resulting in a bidirectional search.

### 2.2.4 Equational Theorems with the Rule $=\!\!(gL)\!\!\Rightarrow$

This section contains a description of OTTER's implementation of the derivation rule $=\!\!(gL)\!\!\Rightarrow$ and some practical information on its use. The rule $=\!\!(gL)\!\!\Rightarrow$ is formally presented in Sec. ??. This section can be skipped during the first reading of this document.

From the operational point of view of OTTER, $=\!\!(gL)\!\!\Rightarrow$ is implemented in two ways: as an inference rule and as a rewrite rule. Application $=\!\!(gL)\!\!\Rightarrow$ requires an equality to have identical terms in two positions. When used as an inference rule, the identical terms occur implicitly through the use of unification (in analogy to the inference rules resolution and paramodulation). Let $F[a_1, x]$ represent a term in that contains a subterm $a_1$ at a particular position, with $x$

representing everything else in the term. Suppose we have $F[a_1, x] = F[a_2, y]$, (i.e., $a_1$, and $a_2$ are in corresponding positions and occur in the same nest of symbols), with $a_1$ and $a_2$ unifiable. By $=\!\!(gL)\!\!\Rightarrow$ we infer $F[z, x'] = F[z, y']$, where $z$ is a new variable, and $x'$ and $y'$ are the appropriate instances of $x$ and $y$. For example, from

$$f(f(x, y), f(z, f(x, z))) = f(u, f(y, u)),$$

we can derive

$$f(f(x, y), f(z, w)) = f(f(x, z), f(y, w))$$

with $=\!\!(gL)\!\!\Rightarrow$ by unifying $u$ and $f(x, z)$ and introducing the new variable $w$.

When used as a rewrite rule, $=\!\!(gL)\!\!\Rightarrow$ is applied automatically to all pairs of identical terms in appropriate positions, in analogy to ordinary rewriting. The equality $F[a, x] = F[a, y]$ is rewritten by $=\!\!(gL)\!\!\Rightarrow$ to $F[z, x] = F[z, y]$, where $z$ is a new variable. For example, from

$$f(e, e, e, e) = f(e, e, x, f(e, e, x, e)),$$

we can derive

$$f(y, z, e, e) = f(y, z, x, f(e, e, x, e))$$

by two applications of $=\!\!(gL)\!\!\Rightarrow$ as a rewrite rule: to the first occurrences of $e$ on each side, then to the second occurrences of $e$.

The user has the options of applying $=\!\!(gL)\!\!\Rightarrow$ as an inference rule, as a rewrite rule before ordinary rewriting, as a rewrite rule after ordinary rewriting, or any combination of these.

### Strategies for use with $=\!\!(gL)\!\!\Rightarrow$

OTTER's implementation of $=\!\!(gL)\!\!\Rightarrow$ can be difficult to work with, because it does not fit well with our ordinary strategies for equational deduction (activated with the OTTER flag `knuth_bendix`). The main reason is that in many cases, the positions at which $=\!\!(gL)\!\!\Rightarrow$ is effectively and usefully applied occur in *simplifiable* terms. With our ordinary strategies, the simplifiable terms are eliminated by demodulation before $=\!\!(gL)\!\!\Rightarrow$ can be applied. Consider, for example, the theorem

$$\{ex = xe = x\} \;=\!\!(gL)\!\!\Rightarrow\; \{xy = yx\}.$$

and the following proof (found by Otter 3.0.3b on merlin.mcs.anl.gov at 17.07 seconds).

| | | |
|---|---|---|
| 2 | $e \cdot x = x$ | |
| 3 | $x \cdot e = x$ | |
| 4 | $B \cdot A \neq A \cdot B$ | |
| 5 | $(e \cdot e) \cdot x = x$ | $[2 \rightarrow 2]$ |
| 7 | $(x \cdot e) \cdot e = x$ | $[3 \rightarrow 3]$ |
| 47 | $(x \cdot e) \cdot e = (e \cdot e) \cdot x$ | $[5 \rightarrow 7]$ |
| 812 | $(x \cdot y) \cdot e = (e \cdot y) \cdot x$ | [(gL) 47] |

| | | |
|---|---|---|
| 827 | $(e \cdot x) \cdot y = y \cdot x$ | [3 $\rightarrow$ 812, flip] |
| 899 | $x \cdot y = y \cdot x$ | [2 $\rightarrow$ 827] |
| 900 | $\square$ | [899,4] |

Note the following points about the preceding proof.

- The notation for the justification of each step is explained in Sec. 2.2.6
- The positions in clause 47 at which $=(gL)\Rightarrow$ applies (to derive clause 812) occur in terms that can be simplified by clauses 2 and 3; in fact, if clauses 2 and 3 become demodulators (as in our ordinary strategies), everything that OTTER derives is demodulated to $x = x$, and the search terminates without finding a proof.
- The proof is quite simple, but OTTER required 17 seconds to find it on a fast computer. (A simple proof like this but without $=(gL)\Rightarrow$ would be found immediately with our ordinary equational strategies.)
- The derivation of clauses 5, 7, and 47 involve paramodulation from a variable, a process that is never allowed with our ordinary equational strategies but is frequently required with $=(gL)\Rightarrow$ .

Our basic strategy for use with $=(gL)\Rightarrow$ allows unrestricted paramodulation from both sides of equalities and from and into variables, and it does not use any demodulation. Such a strategy (which produced the preceding proof) usually results in a very redundant search, due mostly to the lack of demodulation, and it takes a long time to find proofs.

In many cases, however, we can use demodulation with our basic $=(gL)\Rightarrow$ strategy. (Such cases usually have a richer set of axioms than the preceding example.) So when faced with a new $=(gL)\Rightarrow$ conjecture, we typically start with full demodulation as in our ordinary equational strategy, then iteratively restrict demodulation (in successive OTTER runs) until a well-behaved search is achieved.

Our search strategies for $=(gL)\Rightarrow$ are still quite primitive. There is a lot of room for research in this area.

### 2.2.5 Theorems with Deduction Rules

Some of the theories we deal with are specified by axiom sets that contain equational deduction rules (implications) as well as equalities. An example is left cancellation (in its more useful form):

$$xy = u \ \& \ xz = u \ \rightarrow \ y = z.$$

We use the following OTTER flags whenever equational deduction rules are present.

```
set(hyper_res). set(unit_deletion).
set(para_from_units_only). set(para_into_units_only).
set(output_sequent).
```

The flag **hyper_res** enables the inference rule hyperresolution, **unit_deletion** says to delete an antecedent when justified by a derived equation, the paramodulation flags disallow paramodulation from or into nonunit clauses (the deduction rules), and **output_sequent** causes deduction rules to be printed as implications instead of as disjunctions of literals.

#### Multiple Goals

If the equational theorem or conjecture we are trying to prove, say $E \Rightarrow C$, has multiple goals (i.e., if $C$ is not a singleton), then the denial of C is a nonunit negative clause. For example, if we are trying to prove that $E$ is commutative and associative, the denial would be

$$AB \neq BA \mid (AB)C \neq A(BC),$$

where $A$, $B$, and $C$ are Skolem constants. This can be viewed as the deduction rule

$$AB = BA,\ (AB)C = A(BC) \ \rightarrow\ \Box,$$

where $\Box$ represents falsehood. If commutativity and associativity are derived, then hyperresolution with that deduction rule gives a contradiction. Also, the goals can be rewritten with derived demodulators (and individual goals can be removed from the denial with the *unit deletion* process) before hyperresolution produces a contradiction.

### 2.2.6 Example Input Files and Proofs

This section contains two examples of OTTER input files and proofs, with some explanation of the input language and the proof format.

#### Ordinary Equational Logic

Consider the basic theorem that groups satisfying $xx = e$ are commutative. The following annotated input file causes OTTER to find a proof.

```
op(400,xfx,*).           % Declare * to be an infix operation.

set(knuth_bendix).       % Specify the Knuth-Bendix search strategy.

clear(print_kept).       % Disable printing of retained clauses.
clear(print_new_demod).  % Disable printing of new demodulators.
clear(print_back_demod). % Disable printing of back demodulated clauses.

assign(pick_given_ratio, 4).  % 4 best-first : 1 breadth-first.
assign(max_weight, 17).  % Do not retain clauses with weight > 17.
```

```
assign(max_mem, 16000).  % Stop if more 16 megabytes are required.

list(usable).            % The usable list.
x = x.
end_of_list.

list(sos).               % The sos list.
e * x = x.               % Left identity.
i(x) * x = e.            % Left inverse.
(x * y) * z = x * (y * z).  % Associativity.
x * x = e.               % The special hypothesis.
end_of_list.

list(passive).           % The passive list.
A * B != B * A.          % Denial of commutativity.
end_of_list.
```

Note the following points about the input file.

- The first part of the input file, through the end of `list(usable)`, is our standard header for ordinary equational problems. For many theorems, the only part of the header that is adjusted is the parameter `max_weight` (as explained in Sec. 2.2.2).
- OTTER requires the clause "`x = x`" to be present when the inference rule paramodulation is used. (The operational reason is so that a contradiction can be found when $t \neq t$, for some $t$, is derived. The main reason is historical, due to the original definition of paramodulation.)
- OTTER's rule for distinguishing variables from constants is that variables start with (lower case) `u`–`z`. Our convention for naming Skolem constants is to use `A`, `B`, `C`, ... .
- With OTTER, we use "`!=`" for "$\neq$".
- The denial of the conclusion is placed in the passive list, because we wish a forward proof; in this case it is safe in the passive list, because no derived clause can simplify it (Sec. 2.2.2).

If we run OTTER with the preceding input file, it quickly finds a proof. We use a simple program to extract the proof from the output file and translate it into the following form.

*Proof* (found by Otter 3.0.3b on gyro at 0.10 seconds).

| | | |
|---|---|---|
| 1 | $A \cdot B \neq B \cdot A$ | |
| 4,3 | $e \cdot x = x$ | |
| 7 | $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ | |
| 9 | $x \cdot x = e$ | |
| 11 | $x \cdot (x \cdot y) = y$ | [9 $\rightarrow$ 7 :4, flip] |

| | | |
|---|---|---|
| 15 | $x \cdot (y \cdot (x \cdot y)) = e$ | [$9 \rightarrow 7$, flip] |
| 18,17 | $x \cdot e = x$ | [$9 \rightarrow 11$] |
| 25 | $x \cdot (y \cdot x) = y$ | [$15 \rightarrow 11$ :18, flip] |
| 29 | $x \cdot y = y \cdot x$ | [$25 \rightarrow 11$] |
| 30 | $\square$ | [29,1] |

The clause numbers in the proof are not sequential, because they reflect the sequence of *retained* clauses. Some clauses are identified with two numbers; the first is cited when the clause is used as a demodulator, and the second when it is used with an inference rule (paramodulation or hyperresolution).

Each derived clause has a justification. The notation "$m \rightarrow n$" indicates paramodulation from $m$ into $n$; ": $i, j, k, \ldots$" indicates rewriting with the demodulators $i, j, k, \ldots$; and ",flip" indicates that equality was reversed (usually so that the complex sides occurs on the left).

### An Example with $=\!(gL)\!\Rightarrow$

Consider the theorem (new to this work) that a cancellative semigroup satisfying $=\!(gL)\!\Rightarrow$ must be commutative. OTTER finds a proof with the following input file.

```
op(400,xfx,*).           % Declare * to be an infix operation.

set(para_from).          % Paramodulate into the given clause.
set(para_into).          % Paramodulate from the given clause.
set(para_from_vars).     % Allow paramodulation into variables.
set(para_into_vars).     % Allow paramodulation from variables.
set(order_eq).           % Orient equalities.

set(geometric_rule).     % Apply =(gL)=> as an inference rule.
set(geometric_rewrite_before).  % Apply =(gL)=> as an rewrite rule
                                % before ordinary demodulation.

assign(pick_given_ratio, 4).    % 4 best-first : 1 breadth-first.
assign(max_weight, 30).  % Do not retain clauses with weight > 30.
assign(max_mem, 16000).  % Stop if more 16 megabytes are required.

clear(print_kept).       % Disable printing of retained clauses.
clear(print_new_demod).  % Disable printing of new demodulators.
clear(print_back_demod). % Disable printing of back demodulated clauses.

% set(back_demod).       % Apply back demodulation.
% set(process_input).    % Process input clauses as if derived.

set(output_sequent).     % Output nonunit clauses as implications.
set(hyper_res).          % Apply the inference rule hyperresolution.
set(para_from_units_only).  % Disable paramodulation from nonunits.
set(para_into_units_only).  % Disable paramodulation into nonunits.
```

```
list(usable).
x = x.
x * y != u | x * z != u | y = z.  % left cancellation
y * x != u | z * x != u | y = z.  % right cancellation
end_of_list.

list(sos).
(x * y) * z = x * (y * z).
end_of_list.

list(passive).
B * A != A * B.
end_of_list.
```

The first part of the input file, through `process_input` is fairly standard for $=\!(gL)\!\Rightarrow$ searches. The parameter `max_weight` is usually adjusted, and the flags `back_demod` and `process_input` (disabled here), are used for the first attempts, then disabled if they constrain the search too much.

The second part of the input file, from `output_sequent` through the end of the usable list, is used when deduction rules are present.

The preceding input file leads to the following proof.

*Proof* (found by Otter 3.0.3b on gyro at 5.65 seconds).

| | | |
|---|---|---|
| 2 | $x \cdot y = u,\ x \cdot z = u \ \rightarrow\ y = z$ | |
| 3 | $y \cdot x = u,\ z \cdot x = u \ \rightarrow\ y = z$ | |
| 4 | $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ | |
| 5 | $B \cdot A = A \cdot B \ \rightarrow\ \square$ | |
| 6 | $(((x \cdot y) \cdot z) \cdot u) \cdot v = (x \cdot (y \cdot z)) \cdot (u \cdot v)$ | [4 $\rightarrow$ 4] |
| 11 | $(x \cdot (y \cdot z)) \cdot u = (x \cdot y) \cdot (z \cdot u)$ | [4 $\rightarrow$ 4] |
| 432 | $(((x \cdot y) \cdot z) \cdot u) \cdot v = (x \cdot u) \cdot ((y \cdot z) \cdot v)$ | [(gL) 6] |
| 433 | $((x \cdot y) \cdot z) \cdot u = x \cdot (u \cdot (y \cdot z))$ | [432,3,11, flip] |
| 1205 | $x \cdot y = y \cdot x$ | [433,2,432] |
| 1206 | $\square$ | [1205,5] |

The justification "[(gL)" indicates the use of $=\!(gL)\!\Rightarrow$ as an inference rule, and ":(gL)" (not present in the preceding proof) indicates its use as an rewrite rule. Hyperresolution with a deduction rule is indicated by a sequence of clause identifiers at the beginning of the justification, for example, clauses 433 and 1205. Note also the the flag `output_sequent` causes the denial, clause 5, to be written as an implication.

## 2.3 MACE

The program MACE (Models and Counter-Examples) is used to search for finite models of formulas or sets of clauses. When the input is interpreted as the denial of a conjecture, any models found are counter-examples of the conjecture.

MACE is easy to use. Its input language is the same as OTTER's, and few decisions need to be made when using it. The search algorithm, unlike OTTER's, is not adjustable by the user. To run MACE, the user takes the following steps.

1. Prepare a set of clauses. These are usually the same as the clauses prepared for OTTER. If an equivalent set of simpler clauses is known, that should be used instead. (The key parameter is the length of the longest clause.) If very simple lemmas are known, they should be included as well.

2. Declare properties of operations. The two properties currently accepted by MACE are "quasigroup" for binary operations, and "bijection" for unary operations. These declarations speed the search; the clauses that specify the declared properties can be (should be) omitted.

3. Assign constants. To reduce the generation of isomorphic models, the user can assign constants in the clauses to elements of the domain. (If the domain has size $n$, its elements are named $0, 1, \cdots, n-1$.)

4. Specify a domain size. Each MACE search is for models of a fixed size. The user typically makes a sequence of searches, starting with a small size, then increasing it, until a model is found or until resources are exhausted. MACE is complete: if the search for a given domain size terminates without models, then there are none of that size.

MACE is quite limited, both in the size of clauses it can handle and in the size of domain it can search. (The space it uses is proportional to $n^{c+v}$, where $n$ is the domain size, $c$ is the length of the longest clause, and $v$ is the number of variables in the longest clause.) If the clauses are very simple, MACE may be able to complete searches of sizes 10–12 in a reasonable time; examples of the longest clauses that MACE can handle are the following.

$$x \cdot y = u \cdot v,\ x \cdot w = z \cdot v \ \rightarrow\ u \cdot w = z \cdot y$$
$$x \cdot (((x \cdot y) \cdot z')' \cdot y)' = z$$

With such clauses, MACE can search for models of size at most 4 or 5.

### An Example of a MACE Search

The following input file asks MACE to search for a non-commutative group.

```
op(400, xfx, *).
set(dp_transform).

list(usable).
e * x = x.                     % left identity
g(x) * x = e.                  % left inverse
(x * y) * z = x * (y * z).     % associativity
A * B != B * A.                % denial of commutativity
end_of_list.
```

```
list(passive).
assign(e, 0).    % The group identity is element 0.
assign(A, 1).    % The non-commuting elements are 1 and 2.
assign(B, 2).
end_of_list.
```

If we run a sequence of MACE searches specifying sizes $3, 4, \ldots$, the first model found has size six, as expected, because the smallest non-commutative group is order six. The following output is produced.

*Model.* The clauses have the following model (found by MACE 1.0.0 on merlin.mcs.anl.gov at 2.77 seconds).

```
*:  | 0 1 2 3 4 5       g:     0 1 2 3 4 5
  --+------------           ---------------
  0 | 0 1 2 3 4 5              0 1 2 4 3 5
  1 | 1 0 3 2 5 4
  2 | 2 4 0 5 1 3
  3 | 3 5 1 4 0 2
  4 | 4 2 5 0 3 1
  5 | 5 3 4 1 2 0
```

If we include the (dependent) clauses $\{x*e = x, x*g(x) = e\}$, or if we declare "*" to be a quasigroup and "g" to be a bijection, a model is found in a bit less time. If we omit the assignments of the constants $e$, $A$, and $B$, a model is found in about a minute.

MACE is a primitive program; in this work, it has made useful discoveries in only a few cases. In some of the cases where we cite its use, a mathematician familiar with the area could have found a model with little work. However, the program is useful to non-mathematicians, and model-searching programs are likely to become more powerful in the near future.

# Bibliography


[1] G. Grätzer and R. Padmanabhan. [[find title]]. *Pac. J.*, 0(0):??, 1978.

[2] D. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebras*, pages 263–297. Pergamon Press, Oxford, U.K., 1970.

[3] K. Kunen. Single axioms for groups. *Journal of Automated Reasoning*, 9(3):291–308, 1992.

[4] W. McCune. Single axioms for groups and Abelian groups with various operations. *Journal of Automated Reasoning*, 10(1):1–13, 1993.

[5] W. McCune. A Davis-Putnam program and its application to finite first-order model search: Quasigroup existence problems. Tech. Memo. ANL/MCS-TM-194, Argonne National Laboratory, Argonne, IL, 1994.

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

[7] R. N. McKenzie. Equational bases for lattice theories. *Math. Scand.*, 27:24–38, 1970.

[8] R. Padmanabhan. Equational theory of algebras with a majority polynomial. *Algebra Universalis*, 7(2):273–275, 1977.

[9] R. Padmanabhan. Logic of equality in geometry. *Discrete Mathematics*, 15:319–331, 1982.

[10] R. Padmanabhan and W. McCune. Self-dual bases for group theory. In preparation.

[11] R. Padmanabhan and W. McCune. Automated reasoning about cubic curves. *Computers and Math. Applic.*, 29(2):17–26, 1995.

[12] I. R. Shafarevich. *Basic Algebraic Geometry*. Springer-Verlag, New York, 1977.

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

[14] L. Wos, G. Robinson, and D. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *J. ACM*, 12(4):536–541, 1965.