ANL-88-44

OTTER 1.0 Users' Guide

by William W. McCune

Mathematics and Computer Science Division
Mathematics and Computer Science Division
Mathematics and Computer Science Division

Argonne National Laboratory, Argonne, Illinois 60439
operated by The University of Chicago
for the United States Department of Energy under Contract W-31-109-Eng-38

This report has been reproduced from the best available copy.

Available from the
National Technical Information Service
NTIS Energy Distribution Center
P.O. Box 1300
Oak Ridge, TN 37831

Price: Printed Copy A03
Microfiche A01

ANL--88-44

DE89 008656

ANL-88-44

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439-4801

OTTER 1.0 Users' Guide

by

William W. McCune

Mathematics and Computer Science Division

January 1989

This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

LEGIBILITY NOTICE

A major purpose of the Technical Information Center is to provide the broadest dissemination possible of information contained in DOE's Research and Development Reports to business, industry, the academic community, and federal, state and local governments.

Although a small portion of this report is not reproducible, it is being made available to expedite the availability of information on the research discussed herein.

Contents

OTTER 1.0 Users' Guide

by

William W. McCune

Abstract

OTTER (Other Techniques for Theorem-proving and Effective Research) is a resolution-style theorem-proving program for first-order logic with equality. OTTER includes the inference rules binary resolution, hyperresolution, UR-resolution, and binary paramodulation. Some of its other abilities are conversion from first-order formulas to clauses, forward and back subsumption, factoring, weighting, answer literals, term ordering, forward and back demodulation, and evaluable functions and predicates. OTTER is coded in C, and it is portable to a wide variety of computers.

1 Introduction

OTTER (Other Techniques for Theorem-proving and Effective Research) is a resolution-style theorem prover, similar in scope and purpose to the AURA [9] and LMA/ITP [6] theorem provers, which are also associated with Argonne. The primary design considerations have been performance, portability, and compactness and simplicity of the code. The programming language C is used.

OTTER features the inference rules binary resolution, hyperresolution, UR-resolution, and binary paramodulation. These inference rules take a small set of clauses and infer a clause; if the inferred clause is new, interesting, and useful, it is stored and may become available for subsequent inferences.

Other features of OTTER are the following:

- Statements of the problem may be input with either first-order formulas or with clauses (a clause is a disjunction with implicit universal quantifiers and existential quantifiers). If first-order formulas are input, OTTER translates them to clauses.
- Forward demodulation rewrites and simplifies newly inferred clauses with a set of equalities, and back demodulation uses a newly inferred equality (which has been added to the set of demodulators) to rewrite all existing clauses.

- Forward subsumption deletes an inferred clause if it is subsumed by any existing clause, and back subsumption deletes all clauses that are subsumed by an inferred clause.
- Factoring can help to overcome the complexity of non-Horn clauses.
- Weight functions and lexical ordering decide the "goodness" of clauses and terms.
- Answer literals give information about the proofs that are found.
- Evaluable functions and predicates build in integer arithmetic, Boolean operations, and lexical comparisons, and enable users to "program" aspects of deduction processes.

OTTER is not automatic. Even after the user has encoded a problem into first-order logic or into clauses, the user must choose inferences rules, set options to control the processing of inferred clauses, and decide which input formulas or clauses are to be in the initial set of support and which (if any) equalities are to be demodulators. If OTTER fails to find a proof, the user may wish to try again with different initial conditions.

Summaries of other theorem-proving systems can be found in the proceedings of recent CADE meetings [8,7].

It is assumed that the reader knows the terminology of first-order logic and automated theorem proving, including *term* (*variable*, *constant*, *complex term*), *atom*, *literal*, *clause*, *propositional variable*, *function symbol*, *predicate symbol*, *Skolem constant*, *Skolem function*, *formula*, and *conjunctive normal form* (*CNF*). See [10], [1], or [5] for an introduction to automated theorem proving, and see [11] for an overview of the field.

2 Outline of OTTER's Inference Process

Like AURA and LMA/ITP, OTTER uses the given-clause algorithm, which can be viewed as a simple implementation of the set of support strategy. OTTER maintains three lists of clauses: **axioms**, **sos** (set of support), and **demodulators**. (AURA and LMA/ITP have a list called have-been-given; OTTER appends clauses that have been given to **axioms** rather than keeping them in a separate list. The name **axioms** is a bit misleading, because inferred clauses become members of **axioms**—the name has been retained by evolution.)

The main loop for inferring and processing clauses and searching for a refutation is

```
While (sos is not empty and no refutation has been found)
    1. Let given_clause be the 'lightest' (or optionally the first)
       clause in sos;
    2. Move given_clause from sos to axioms;
    3. Infer and process new clauses using the inference rules in
           effect; each new clause must have the given_clause as
           one of its parents and members of axioms as its other
           parents;  new clauses that pass the retention tests
           are appended to sos;
End of while loop.
```

The procedure for processing a newly inferred clause **new_cl** is

```
1.  (optional) Output new_cl.
2.  Demodulate new_cl (including $ evaluation).
3.  (optional) Orient equalities.
4.  Merge identical literals (leftmost copy is kept).
5.  (optional) Sort literals.
6.  (optional) Discard new_cl and exit if new_cl has too many literals.
7.  Discard new_cl and exit if new_cl is a tautology.
8.  (optional) Discard new_cl and exit if new_cl is too 'heavy'.
9.  (optional) Discard new_cl and exit if new_cl is subsumed by any clause
        in axioms or sos (forward subsumption).
10. (optional) Apply unit deletion.
11. Integrate new_cl and append it to sos.
12. (optional) Output kept clause.
13. (optional) If new_cl is an equality unit, try to introduce a
        new function symbol.
14. (optional) Try to make new_cl into a demodulator.

15. If new_cl has 0 literals, a refutation has been found.
16. If new_cl has 1 literal, then search axioms and sos for
        unit conflict (refutation) with new_cl.
17. (optional) Print the proof if a refutation has been found.
18. (optional) Back demodulate if Step 14 made new_cl into a demodulator.
19. (optional) Discard each clause in axioms and each clause in sos that
        is subsumed by new_cl (back subsumption).
20. (optional) Factor new_cl and process factors.
```

Steps 15–20 are delayed until steps 1–14 have been applied to all clauses inferred from the current given clause.

3 Using OTTER

OTTER is not interactive. On UNIX and on UNIX-like systems it reads from the standard input and writes to the standard output:

```
otter < input_file > output_file
```

3.1 Syntax

Comments can be placed in the input file by using the symbol %. All characters from the first % on a line to the end of the line are ignored. Comments can occur within terms. Comments are not echoed to the output file.

3.1.1 Names

Names are alphanumeric strings that may contain some other characters such as $ and _. A name may contain up to 50 characters. Names are used as constant symbols, function symbols, predicate symbols, propositional variables, and regular variables. In general, the type (predicate symbol, function symbol, constant, variable) of a name is determined by its context. Since the variables in

clauses are not explicitly bound by universal quantifiers, a convention must be used to distinguish constants from variables. The rule is that in clauses, variables start with (lower case) `u`, `v`, `w`, `x`, `y`, or `z`. In formulas, any name can be used as a variable, because variables are explicitly quantified.

A name usually cannot be used for two different purposes. For example, an input error will be flagged if a symbol has different occurrences with different numbers of arguments. (This protective feature can be overridden by the command `clear(check_arity)`, Section 4.1.6.)

Some names are special. Any binary predicate symbol that starts with `EQ`, `Eq`, or `eq` is understood by demodulation and paramodulation as an equality predicate. The symbol = can be used to write infix equality atoms. All symbols that start with `$` are reserved for special purposes. Any predicate symbol that starts with `$ANS`, `$Ans`, or `$ans` is understood as an answer predicate (answer literal, Section 8). Other symbols that start with `$` are evaluable functions or predicates (Section 6).

3.1.2 Terms and Atoms

Determining whether a simple term is a constant or a variable depends on the context of the term. If it occurs in a clause, then the name determines the type (see above). If it occurs in a formula, it is a variable if it is bound by a quantifier. Most complex terms are written in prefix form, for example `f(a,b,c)`.

Prolog-style list notation can be used to write terms that represent lists: the symbol [] is an abbreviation for `$nil`, $[t_1|t_2]$ abbreviates for $\texttt{\$cons}(t_1,t_2)$, and $[t_1,t_2,t_3,t_4]$ abbreviates $\texttt{\$cons}(t_1,\texttt{\$cons}(t_2,\texttt{\$cons}(t_3,\texttt{\$cons}(t_4,\texttt{\$nil}))))$. White space (spaces, tabs, newlines) can occur in complex terms anywhere except within names and between a function or predicate symbol and the opening parenthesis.

Atoms are similar to complex terms, except that a name is also an atom (a propositional variable), and equalities and negated equalities can be written in infix form as $(t_1 \texttt{ = } t_2)$ and $(t_1 \texttt{ != } t_2)$. White space is required around = and !=, and parentheses are required.

3.1.3 Literals and Clauses

If a is an atom, then a and $-a$ are literals. There should be no white space between the negation sign and the atom. A clause is a sequence of literals separated with |. White space is optional before and after literals. A clause is always terminated with a period (but the period is not considered to be part of the clause).

3.1.4 Formulas

1. Atoms are formulas.
2. If F and G are formulas, then $(F \texttt{ <-> } G)$ and $(F \texttt{ -> } G)$ are formulas.
3. If $F_1, \ldots, F_n$ are formulas, then $(F_1 \texttt{ | } \cdots \texttt{ | } F_n)$ and $(F_1 \texttt{ \& } \cdots \texttt{ \& } F_n)$ are formulas.
4. The symbols `all` and `exists` are quantifiers. If $Q_1 \ldots Q_n$ are quantifiers, $x_1 \ldots x_n$ are names, and F is a formula, then $(Q_1\ x_1 \cdots Q_n\ x_n\ F)$ is a formula.
5. If F is a nonnegated formula, then $-F$ is a formula.

The symbols have their expected meanings: <-> means "if and only if", -> means "implies", | means "or", and & means "and".

All parentheses are required, and white space is required around <->, ->, |, and &, and after quantifiers and their associated variable occurrences.

Note that the following are *not* formulas: `--p(a)` (double negation), `(p & q -> r)` (not enough parentheses), `(all x p(x) & q(x))` (not enough parentheses), `(p&q)` (not enough white space).

Clauses are different from formulas, both in syntax and in treatment by OTTER. The string "`p | q | r`" is a clause, and "`(p | q | r)`" is a formula. Formulas are translated into clauses (negation normal form, Skolemization, then CNF) when input.

3.2 Commands and the Input File

Input to OTTER consists of a small set of commands, some of which indicate that a list of objects (clauses, formulas, or weight templates) follows the command. All lists of objects are terminated with **end_of_list**. The commands are given in Table 1.

set(*flag_name***).**	**% set a flag**
clear(*flag_name***).**	**% clear a flag**
assign(*parameter_name***,***integer***).**	**% assign an integer to a parameter**
lex(*lex_list***).**	**% assign an ordering on symbols**
list(axioms).	**% read axioms in clause form**
list(sos).	**% read set of support in clause form**
list(demodulators).	**% read demodulators in clause form**
formula_list(axioms).	**% read axioms in formula form**
formula_list(sos).	**% read set of support in formula form**
weight_list(*weight_list_name***).**	**% read weight templates**

Table 1: Commands

There are a few constraints on the order of commands. All options that control the conversion of formulas to clauses must occur before any **formula_list** command. The only constraints on combinations and replications of commands concerns the **lex** command (Section 3.2.5) and the **weight_list** commands (Section 7).

3.2.1 Input of Options

OTTER recognizes two kinds of options: flags and parameters. Flags are Boolean-valued options; they are changed with the **set** and the **clear** commands, which take the name of the flag as the argument. Parameters are integer-valued options; they are changed with the **assign** command, which takes the name of the parameter as the first argument and an integer as the second. Examples are

```
set(print_gen).                 % print all generated clauses
clear(back_sub).                % do not do back subsumption
assign(max_seconds, 300).       % stop after about 300 seconds
```

The options are described and their default values are given in Section 4.

3.2.2 Input of Lists of Clauses

A list of clauses is specified with one of the following, and is terminated with **end_of_list**. Each clause is terminated with a period.

```
list(axioms).
list(sos).
list(demodulators).
```

Example:

```
list(axioms).
  (x = x).                          % reflexivity
  (f(e,x) = x).                     % left identity
  (f(g(x),x) = e).                  % left inverse
  (f(f(x,y),z) = f(x,f(y,z))).      % associativity
  (f(z,x) != f(z,y)) | (x = y).     % left cancellation
  (f(x,z) != f(y,z)) | (x = y).     % right cancellation
end_of_list.
```

3.2.3 Input of Lists of Formulas

A list of formulas is specified with one of the following, and is terminated with `end_of_list`. Each formula is terminated with a period.

```
formula_list(axioms).
formula_list(sos).
```

Example (equivalent to above):

```
formula_list(axioms).
  (all a (a = a)).                                    % reflexivity
  (all a (f(e,a) = a)).                               % left identity
  (all a (f(g(a),a) = e)).                            % left inverse
  (all a all b all c (f(f(a,b),c) = f(a,f(b,c)))).    % associativity
  (all a all b all c ((f(c,a) = f(c,b)) -> (a = b))). % left cancellation
  (all a all b all c ((f(a,c) = f(b,c)) -> (a = b))). % right cancellation
end_of_list.
```

3.2.4 Input of Lists of Weight Templates

A list of weight templates is specified with one of the following, and is terminated with `end_of_list`. Each weight template is terminated with a period.

```
weight_list(pick_given).          % for picking given clauses
weight_list(purge_gen).           % for discarding generated clauses
weight_list(pick_and_purge).      % for both picking and purging
weight_list(terms).               % for ordering terms
```

Example:

```
weight_list(pick_and_purge).
   weight(a, 0).                  % weight of constant a is 0
   weight(g(2), -50).             % twice weight of argument - 50
   weight(P(1,1), 100).           % sum of weights of arguments + 100
   weight(x, 5).                  % all variables have weight 5
```

```
    weight(f(g(-3), 4), -300).        % see Section 7
end_of_list.
```

See Section 7 for the syntax and use of weight templates.

3.2.5 Input of the Lex Term

The lex term is used to specify an ordering on function and constant symbols, and thereby a lexical ordering on terms. Lexical ordering on terms is used in three contexts: orienting equality literals (Sections 4.1.4 and 5.3), deciding whether to apply a lex-dependent demodulator (Section 5.2), and evaluating functions/predicates that perform lexical comparisons (Section 6).

There cannot be more than one **lex** command in the input file, and the lex term should contain all appropriate symbols. The order for terms that do not occur in the lex term is the order in which they occur in the input file (the order in which they are entered into the symbol table).

For example, if **or** is a binary function symbol, and **a**, **b**, **c**, **d** are constants, the lex command

```
lex( [a, b, c, d, or(x,x)] ).
```

specifies **a** $\prec$ **b** $\prec$ **c** $\prec$ **d** $\prec$ **or**. The arguments of **or** serve as place-holders only; they identify **or** as a 2-place function symbol.

There are two ways of lexically ordering terms with variables—see Sections 4.1.4 and 5.1.

4 Options

Flags are Boolean-valued options, and parameters are integer-valued options. When the user changes an option, OTTER sometimes automatically changes other options—the user will be informed when such a change occurs.

4.1 Flags

4.1.1 Inference Rules

binary_res — default clear. If this flag is set, use the inference rule binary resolution (along with any other inference rules that are set) to generate new clauses.

hyper_res — default clear. If this flag is set, use the inference rule (positive) hyperresolution (along with any other inference rules that are set) to generate new clauses.

ur_res — default clear. If this flag is set, use the inference rule UR-resolution (unit-resulting resolution) (along with any other inference rules that are set) to generate new clauses.

para_into — default clear. If this flag is set, use the inference rule "paramodulation into the given clause" (along with any other inference rules that are set) to generate new clauses.

para_from — default clear. If this flag is set, use the inference rule "paramodulation from the given clause" (along with any other inference rules that are set) to generate new clauses.

demod_inf — default clear. If this flag is set, apply demodulation, as if it were an inference rule, to the given clause. This is useful for debugging sets of demodulators. When this flag is set, the given clause is copied, then processed just like any newly generated clause.

4.1.2 Paramodulation Flags

para_from_left — default clear. If this flag is set, allow paramodulation from the left sides of equality literals. (Applies to both "para_into" and "para_from" inference rules.)

para_from_right — default clear. If this flag is set, allow paramodulation from the right sides of equality literals. (Applies to both "para_into" and "para_from" inference rules.)

para_from_vars — default clear. If this flag is set, allow paramodulation from variables. (Applies to both "para_into" and "para_from" inference rules.)

para_into_vars — default clear. If this flag is set, allow paramodulation into variables. (Applies to both "para_into" and "para_from" inference rules.)

para_ones_rule — default clear. If this flag is set, paramodulation obeys the 1's rule. (The 1's rule is a special-purpose strategy for problems in combinatory logic—its usefulness has not been demonstrated elsewhere.) (Applies to both "para_into" and "para_from" inference rules.)

para_all — default clear. If this flag is set, then replace all occurrences of the *into term* with the replacement term. (Applies to both "para_into" and "para_from" inference rules.)

no_para_into_left — default clear. If this flag is set, then prohibit paramodulation into left arguments of positive and negative equalities. (Applies to both "para_into" and "para_from" inference rules.)

no_para_into_right — default clear. If this flag is set, then prohibit paramodulation into right arguments of positive and negative equalities. This flag is one of the options to be used when searching for a complete set of reductions. (Applies to both "para_into" and "para_from" inference rules.)

4.1.3 Flags for Handling Generated Clauses

(Section 4.1.4 gives additional, equality-related flags for handling generated clauses.)

print_gen — default clear. If this flag is set, output new clauses at the beginning of processing.

order_eq — default clear. If this flag is set, flip equalities if the right side is heavier than the left. See Section 5 for the meaning of "heavier".

sort_literals — default clear. If this flag is set, literals of newly generated clauses are sorted: negative literals, then positive literals, then answer literals. The main purpose of this flag is to make clauses more readable. In some cases, this flag can speed up subsumption on non-unit clauses.

for_sub — default set. If this flag is set, apply forward subsumption during the processing of newly generated clauses. (Delete the new clause if it is subsumed by any clause in **axioms** or **sos**.)

unit_deletion — default clear. If this flag is set, apply unit deletion to newly generated clauses. Unit deletion removes a literal from a newly generated clause if the literal is the negation of an instance of a unit clause that occurs in **axioms** or **sos**. For example, the second literal of `p(a,x) | q(a,x)` is removed by the unit `-q(u,v)`; but it is not removed by the unit `-q(u,b)`, because that unification causes the instantiation of `x`. All such literals are removed from the newly generated clause, even if the result is the empty clause. (Unit deletion is not useful if only units are being generated.)

print_kept — default set. If this flag is set, output new clauses if they pass all retention tests.

back_sub — default set. If this flag is set, apply back subsumption during the processing of newly kept clauses. (Delete all clauses in **axioms** or **sos** that are subsumed by the newly kept clause.)

print_back_sub — default set. If this flag is set, output clauses when they are back subsumed.

factor — default clear. If this flag is set, factor newly kept clauses. Note that unlike other inference rules, factoring is not applied to the given clause—it is applied to a new clause as soon as it is kept. All factors are generated in an iterative manner. Factoring *is* attempted on answer literals. If factoring is enabled, a clause with n literals will never subsume a clause with fewer than n literals.

4.1.4 Demodulation and Equality Flags

demod_history — default set. If this flag is set, then when a clause is demodulated, the numbers of the demodulators are included in the derivation history of the clause.

demod_linear — default clear. If this flag is set, disable demodulation indexing and use a linear search of **demodulators** when rewriting a term. With indexing disabled, if more than one demodulator can be applied to rewrite a term, then the one that occurs first in the input file is applied; this is useful when demodulation is used to do "procedural" things. With indexing enabled (the default), demodulation is much faster, but the order in which **demodulators** is applied is not under the control of the user, and no two demodulators may have the same left side.

demod_out_in — default clear. If this flag is set, terms are demodulated outside-in, left-to-right. In other words, attempt to rewrite a term before rewriting (left-to-right) its subterms. The algorithm is "repeat {rewrite the left-most outer-most rewritable term} until no more rewriting can be done or the limit is reached". (The effect is like a standard reduction in lambda-calculus or in combinatory logic.) If this flag is clear, terms are demodulated inside-out (all subterms are fully demodulated before attempting to rewrite a term). The one exception when inside-out demodulation is in effect is the evaluable conditional term **$IF**(*condition*,*then-value*,*else-value*) (Section 6).

dollar_eval — The setting and clearing of this flag are handled by OTTER. If evaluable functions or predicates are found in the input, this flag is set automatically.

dynamic_demod — default clear. If this flag is set, attempt to make *some* newly kept equalities into demodulators (Section 5.4). Setting this flag automatically sets the flag **order_eq**.

dynamic_demod_all — default clear. If this flag is set, attempt to make *all* newly kept equalities into demodulators (Section 5.4). Setting this flag automatically sets the flag **dynamic_demod**.

print_new_demod — default set. If this flag is set, print demodulators that are adjoined during the search (**dynamic_demod**).

back_demod — default clear. If this flag is set, back demodulate **axioms**, **sos**, and **demodulators** whenever a new demodulator is added. Back demodulation is delayed until the inference rules are finished generating clauses from the current given clause (delayed until **post_process**). Setting the **back_demod** flag automatically sets the flags **order_eq** and **dynamic_demod** (Section 2). (*Warning:* the order in which clauses are back demodulated is in effect nondeterministic—it may change from run to run.)

print_back_demod — default set. If this flag is set, print clauses before they are back demodulated.

symbol_elim — default set. If this flag is set, then new demodulators will be oriented, if possible, so that function symbols (excluding constants) are eliminated. A demodulator can eliminate all occurrences of a function symbol if the arguments on the left side are all different variables, and the function symbol of the left side does not occur in the right side. For example, the demodulators `g(x) = f(x,x)` and `h(x,y) = f(x,f(y,f(g(x),g(y))))` eliminate all occurrences of `g` and `h`, respectively.

knuth_bendix — default clear. If this flag is set, then OTTER will approximate a version of the Knuth-Bendix completion procedure. It is not a correct implementation of the completion pro-

cedure, because term ordering is not correct—in particular, termination of demodulation is not guaranteed. Setting the **knuth_bendix** flag automatically causes the following flags to be set: **para_from**, **para_into**, **para_from_left**, **no_para_into_right**, **dynamic_demod_all**, **back_demod**, and **print_lists_at_end**.

new_functions — default clear. If this flag is set, then positive equality units of a particular type cause the introduction of new function symbols and equalities. If an equality has the property that each side has at least one variable that does not occur in the other side, then a new function symbol and two new equalities are introduced. For example, `(d(x,d(y,d(d(x,x),x))) = d(d(z,z),y))` causes the introduction of `(d(x,d(y,d(d(x,x),x))) = k1(y))` and `(d(d(z,z),y) = k1(y))`. The new function symbol is **k1**; its argument list is the intersection of the variable sets of the two sides. See [4] for more detail.

lex_order_vars — default clear. This flag affects lex-dependent demodulation and the evaluable functions and predicates that perform lexical comparisons. If this flag is set, then lexical ordering is a total order on terms; variables are lowest in the term order, with **x** $\prec$ **y** $\prec$ **z** $\prec$ **u** $\prec$ **v** $\prec$ **w** $\prec$ **v6** $\prec$ **v7** $\prec$ **v8** $\prec \cdots$. If this flag is clear, then a variable is comparable only to another occurrence of the same variable; it is not comparable to other variables or to nonvariables. For example, `$LLT(f(x),f(y))` evaluates to **$T** if and only if **lex_order_vars** is set. See Section 5.1 for more detail.

4.1.5 Indexing Flags

for_sub_fpa — default clear. If this flag is set, use FPA indexing for forward subsumption. If this flag is clear, use discrimination tree indexing for forward subsumption. This flag can be set to decrease the amount of memory required by OTTER. Discrimination tree indexing can require a lot of memory, but it is much faster than FPA indexing.

no_fapl — default clear. If this flag is set, do not index positive literals for unit conflict or back subsumption. This should be used only when no negative units will be generated (as with hyperresolution), back subsumption is disabled, and discrimination tree indexing is being used for forward subsumption. This option can save a little time and memory.

no_fanl — default clear. If this flag is set, do not index negative literals for unit conflict or back subsumption. This should be used only when no positive units will be generated, back subsumption is disabled, and discrimination tree indexing is being used for forward subsumption. This option can save a little time and memory.

4.1.6 Miscellaneous Flags

process_input — default clear. If this flag is set, input **axioms** and **sos** clauses (including clauses from formula input) are processed as if they had been generated by an inference rule. The processing includes subsumption, demodulation, back demodulation. (see Section 2, "procedure for processing newly inferred clause").

simplify_fol — default clear. If this flag is set, then attempt some simplification when converting input first-order formulas into clauses. The simplification occurs after Skolemization, during the CNF translation. (Future releases may attempt simplification of quantified formulas.)

print_given — default set. If this flag is set, output clauses when they become given clauses.

print_weight — default clear. If this flag is set, print the weight of each given clause. This is useful for debugging sets of weight templates.

sos_queue — default clear. If this flag is set, the first clause in **sos** becomes the given clause (the set of support is a queue). If this flag is clear, the lightest clause (Section 7) in **sos** becomes the

given clause.

free_all_mem — default clear. If this flag is set, then at the end of the run, return all memory to the memory managers. (This is used to ensure that no memory is being lost.) When this flag is set, the "in use" column of the memory statistics should be all 0's. This flag is used primarily for system debugging.

check_arity — default set. If this flag is set, symbols must not have variable arities (different numbers of arguments in different places in the input). For example, the term `p(a,a(b))` would not be allowed. (Constants have arity 0.) If this flag is clear, then variable arities are permitted; in the preceding term, the two occurrences of `a` would be treated as different symbols.

bird_print — default clear. If this flag is set, terms constructed with the binary function `a` are output in combinatory logic notation (without the function symbol `a` and left associated unless otherwise indicated). For example, the clause `(a(a(a(S,x),y),z) = a(a(x,z),a(y,z)))` is output as `(S x y z = x z (y z))`. At present, terms cannot be input in combinatory logic notation.

atom_wt_max_args — default clear. If this flag is set, the default weight of an atom (the weight if no template matches the atom) is 1 + the maximum of the weights of the arguments. If this flag is clear, the default weight of an atom is 1 + the sum of the weights of the arguments.

term_wt_max_args — default clear. If this flag is set, the default weight of a term (the weight if no template matches the atom) is 1 + the maximum of the weights of the arguments. If this flag is clear, the default weight of a term is 1 + the sum of the weights of the arguments.

print_lists_at_end — default clear. If this flag is set, then **axioms**, **sos**, and **demodulators** are printed at the end of the search.

print_proofs — default set. If this flag is set, all proofs found are printed to the output file. If this flag is clear, no proofs are printed to the output file.

4.2 Parameters

Parameters are integer-valued options. In the descriptions that follow, n is the value of the parameter, and MAX_INT is a large integer, usually the size of the largest normal integer on the user's computer.

4.2.1 Monitoring Progress

report — default 0, range [0..MAX_INT]. If n is not 0, then output statistics approximately every n seconds. It is not exact, because statistics will be output only after the current given clause is finished. n should not be too small; $n = 30$ is a good start. This feature can be used in conjunction with UNIX programs such as **grep** and **awk** to conveniently monitor OTTER jobs.

4.2.2 Placing Limits on the Search

max_seconds — default 0, range [0..MAX_INT]. If n is not 0, then terminate the search after about n seconds. It is not exact, because OTTER will wait until the current given clause is finished before stopping.

max_gen — default 0, range [0..MAX_INT]. If n is not 0, then terminate the search after about n clauses have been generated. It is not exact, because OTTER will wait until it is finished with the current given clause before stopping.

max_kept — default 0, range [0..MAX_INT]. If n is not 0, then terminate the search after about n

clauses have been kept. It is not exact, because OTTER will wait until it is finished with the current given clause before stopping.

max_given — default 0, range [0..MAX_INT]. If n is not 0, then terminate the search after n given clauses have been used. This one is exact.

max_mem — default 0, range [0..MAX_INT]. If n is not 0, then OTTER will terminate the search before more than n K bytes have been dynamically allocated (**malloc**). Processing of the current given clause is not completed with this kind of termination.

4.2.3 Limits on the Size of Generated Clauses

max_literals — default 0, range [0..MAX_INT]. If n is not 0, then new clauses are discarded if they contain more than n literals.

max_weight — default 0, range [0..MAX_INT]. If n is not 0, then new clauses are discarded if their weight is more than n. The weight list **purge_gen** or the weight list **pick_and_purge** is used to weigh clauses (both lists cannot be present, Section 7).

4.2.4 Indexing Parameters

fpa_literals — default 3, range [0..8]. n is the FPA indexing depth for literals. (FPA literal indexing is used for resolution inference rules, back subsumption, and unit conflict. It is also used for forward subsumption if the flag **for_sub_fpa** is set.) If $n = 0$, indexing is by predicate symbol only; if $n = 1$, indexing looks at the predicate symbol and the symbols that are arguments of the literal, and so on. Greater indexing depth requires more memory. Changing this parameter should never change the clauses that are generated or kept.

fpa_terms — default 3, range [0..8]. n is the FPA indexing depth for terms. (FPA term indexing is used for paramodulation inference rules and back demodulation.) If $n = 0$, indexing is by function symbol only; if $n = 1$, indexing looks at the function symbol and the symbols that are arguments of the literal, and so on. Greater indexing depth requires more memory. Changing this parameter should never change the clauses that are generated or kept.

4.2.5 Miscellaneous Parameters

demod_limit — default 100, range [0..MAX_INT]. If n is not 0, then n is the maximum number of rewrites that will be applied when demodulating a clause. If n is 0, there is no limit. A warning message is printed if OTTER attempts to exceed the limit.

max_proofs — default 1, range [0..MAX_INT]. If $n = 1$, OTTER will stop if it finds a proof. If $n > 1$, then OTTER will not stop when it has found the first proof; instead, it will try to keep searching until it has found n proofs. (Some of the "different" proofs may in fact be identical.) (Because forward subsumption occurs before unit conflict, a clause representing a truly different proof may be discarded by forward subsumption before unit conflict detects the proof.) If $n = 0$, OTTER will find as many proofs as it can.

neg_weight — default 0, range [-MAX_INT..MAX_INT]. n is the additional weight (positive or negative) that is given to negated literals. Weight templates cannot be used to do this, because the negation sign cannot occur in weight templates (Atoms, not literals, are weighed with weight templates, Section 7.)

stats_level — default 2, range [0..3]. This is the level of detail of statistics printed in reports and at the end of the search. If $n = 0$, no statistics are output; if $n = 1$, a few important statistics

are output; if $n = 2$, most relevant statistics are output; and if $n = 3$, most relevant statistics and subsumption counts are output. This parameter does not affect the speed of OTTER, because all statistics are always kept.

5 Ordering and Dynamic Demodulation

This section contains a more complete explanation of the options **lex_order_vars**, **order_eq**, **symbol_elim**, **dynamic_demod**, and **dynamic_demod_all**, and it gives all the rules—built in and optional—for orienting equality literals and determining dynamic demodulators. In this section, α and β always refer to the left and right arguments, respectively, of the equality literal under consideration.

5.1 Lexical Order

One can assign an ordering on symbols by using the **lex** command (Section 5.1). For example, the command

```
lex( [a, b, c, d, or(x,x)] ).
```

specifies **a** $\prec$ **b** $\prec$ **c** $\prec$ **d** $\prec$ **or** (**or** is a binary function symbol). If relevant symbols are omitted from the **lex** command, OTTER chooses an order. An ordering on symbols gives a lexical ordering on terms. Continuing the example, **a** $\prec$ **or(a,a)** $\prec$ **or(a,c)** $\prec$ **or(b,a)**. The flag **lex_order_vars** controls lexical ordering of terms containing variables.

lex_order_vars is set: Variables are the lowest in the symbol ordering, with **x** $\prec$ **y** $\prec$ **z** $\prec$ **u** $\prec$ **v** $\prec$ **w** $\prec$ **v6** $\prec$ **v7** $\prec$ **v8** $\prec \cdots$. Since the order on symbols it total (any two symbols are comparable), the lexical order on terms is total (any two terms are comparable). Note that applying a substitution to a pair of terms may change their relative order.

lex_order_vars is clear (the default): A variable is comparable only to itself; it is not comparable to different variables or to nonvariable terms. Continuing the example, **f(a,x,y)** $\prec$ **f(b,y,x)**, but **f(x,a,y)** and **f(y,b,x)** are *not* comparable. The order on terms is partial. Note that if $t_1 \prec t_2$, and if σ is any substitution, then $t_1\sigma \prec t_2\sigma$.

Lexical ordering on terms is used in three contexts: deciding whether to apply a lex-dependent demodulator (Section 5.2), evaluating functions/predicates that perform lexical comparisons (Section 6), and orienting equality literals (Sections 4.1.4 and 5.3). When orienting equality literals, partial lexical ordering is used, even if **lex_order_vars** is set.

5.2 Lex-dependent Demodulation

Two terms are *identical-except-variables* if they are identical after replacing all occurrences of variables with **x**. An input demodulator is lex-dependent if and only if α and β are identical-except-variables. A dynamic demodulator is lex-dependent *only if* α and β are identical-except-variables. (See Section 5.4 for determining lex-dependent dynamic demodulators.) A lex-dependent demodulator applies to a term only if its application produces a lexically smaller term. When checking "lexically smaller", the flag **lex_order_vars** is consulted.

In the presence of the **lex** command and the (lex-dependent) demodulators

```
lex( [a, b, c, d, or(x,x)] ).

list(demodulators).
  eq(or(x,y), or(y,x)).
  eq(or(x,or(y,z)), or(y,or(x,z))).
end_of_list.
```

the term `or(or(d,b),or(a,c))` will be demodulated to `or(a,or(b,or(c,d)))` (in several steps).

5.3 Orienting Equalities

Orienting equality literals (positive and negative) except positive unit equalities. The arguments α and β are weighed (Section 7) using **weight_list(terms)**. If $wt(\alpha) < wt(\beta)$, the literal is flipped; if $wt(\alpha) = wt(\beta)$, then α and β are compared in the partial lexical order (Section 5.1); if $\alpha \prec \beta$, the literal is flipped.

Orienting positive unit equalities. More care is taken in orienting positive unit equalities, because they may become dynamic demodulators. The procedure is the following:

1. If the **symbol_elim** flag is set and if the equality is a symbol-eliminating type (Section 4.1.4), then orient the equality in the appropriate direction and exit.
2. If one argument is a proper subterm of the other argument, then orient the equality so that the subterm is the right argument and exit.
3. Proceed as in the preceding paragraph "Orienting equality literals ...". If the lexical comparison shows that the two arguments are incomparable, then if $vars(\alpha) \not\supseteq vars(\beta)$ and $vars(\alpha) \subseteq vars(\beta)$, then the literal is flipped.

5.4 Determining Dynamic Demodulators

A dynamic demodulator is a demodulator that is inferred rather than input. If either of the flags **dynamic_demod** or **dynamic_demod_all** is set, OTTER will attempt to make some or all inferred positive equality units into demodulators.

If either of the flags **dynamic_demod** or **dynamic_demod_all** is set, then the flag **order_eq** is automatically set (Section 4.1.4). (Dynamic demodulators are decided when equalities are oriented, before forward subsumption. An equality actually becomes a dynamic demodulator after forward subsumption.) The procedure assumes that equalities have already been oriented.

1. If the flag **symbol_elim** is set and if it applies, the equality becomes a demodulator.
2. If β is a proper subterm of α, the equality becomes a demodulator.
3. If α and β are comparable, in particular, if $wt(\alpha) < wt(\beta)$ or $(wt(\alpha) = wt(\beta)$ and $\alpha \prec \beta)$,
   (a) if **dynamic_demod_all** is set, the equality becomes a demodulator;
   (b) if **dynamic_demod_all** is clear and if $wt(\beta) \leq 1$, the equality becomes a demodulator.
4. If α and β are incomparable, if they are identical-except-variables (Section 5.2), and if $vars(\alpha) \supseteq vars(\beta)$, then the equality becomes a lex-dependent demodulator.

5.5 Completion and Termination

The weighting and lexical ordering schemes of OTTER do not guarantee termination of demodulation. It is up to the user to try to make sure that demodulation terminates. (Parameter **demod_limit** can be used to limit the number of rewrites—see Section 4.1.4.)

The flag **knuth_bendix** (Section 4.1.4) can be used to simulate the Knuth-Bendix completion procedure and some of its variants, but it is at best an approximation, because weighting and lexical ordering schemes do not guarantee termination. Future releases may include orderings with the appropriate properties. See [3] for a state-of-the-art theorem prover based on rewriting and completion, and see [2] for recent work in that field.

6 Evaluable Functions and Predicates ($SUM, $LT, ...)

OTTER, like AURA and ITP, recognizes some special function and predicate symbols as evaluable symbols. Integer arithmetic, lexical comparison, Boolean evaluation, and conditional expressions can be employed when a user wishes to "program" some aspect of a theorem-proving task. (The speed of $ evaluation is not outstanding—it may be improved in future releases.)

Evaluation occurs during demodulation and during hyperresolution. If, for example, demodulation encounters a term **$SUM**($i_1$, i_2), where i_1 and i_2 are integers, the term is rewritten to i_3, the sum of i_1 and i_2, as if the demodulator (**$SUM**($i_1$, i_2) = i_3) were present. If, for example, hyperresolution encounters the negative literal -**$LT**($t_1$, t_2), then t_1 and t_2 are demodulated; if the results are (respectively) integers i_1 and i_2, with $i_1 < i_2$, then the literal is removed as if the unit clause **$LT**($t_1$, t_2) were present.

The symbols that evaluate to type Boolean can occur as either function symbols (demodulation) or predicate symbols (demodulation and hyperresolution). If they are used as function symbols, the Boolean constants are **$T** (true) and **$F** (false).

int × *int* → *int*	**$SUM**, **$PROD**, **$DIFF**, **$DIV**, **$MOD**
int × *int* → *bool*	**$EQ**, **$NE**, **$LT**, **$LE**, **$GT**, **$GE**
term × *term* → *bool*	**$ID**, **$LNE**, **$LLT**, **$LLE**, **$LGT**, **$LGE**
bool × *bool* → *bool*	**$AND**, **$OR**
bool → *bool*	**$NOT**
→ *bool*	**$T**, **$F**
bool × *term* × *term* → *term*	**$IF**

Table 2: Evaluable Functions and Predicates

Table 2 contains all of the evaluable functions and predicates. Their behavior is the following:

1. *int* × *int* → *int*. The term evaluates if both arguments demodulate to integers. **$DIV** is integer division, and **$MOD** is remainder.

2. *int* × *int* → *bool*. The term evaluates if both arguments demodulate to integers.

3. *term* × *term* → *bool*. The term always evaluates. These operations are similar to the five operations in *int* × *int* → *bool*, except that the comparisons are lexical instead of arithmetic. The lexical comparison is the same as in lex-dependent demodulation; in particular, the flag **lex_order_vars** (Section 4.1.4) has effect. Note that **$LLT**($t_1$,$t_2$) is not the same as **$LGT**(t_2,t_1), because t_1 and t_2 are not necessarily comparable (Section 5.1).

4. *bool* × *bool* → *bool*. The term evaluates if both arguments demodulate to Booleans. (This is more restrictive than need be; for example, **$AND($F,bird)** does not evaluate.)

5. *bool* → *bool*. Tne term evaluates if its argument demodulates to Boolean.

6. → *bool*. If hyperresolution encounters a literal -**$T** or a literal **$F**, the literal is removed. If hyperresolution encounters a literal -**$F** or a literal **$T**, the entire hyperresolvent is discarded (because it is a tautology).

7. *bool* × *term* × *term* → *term*. The **$IF** function is the *if-then-else* operator. It is described in the following paragraph.

When inside-out (the default) demodulation encounters a term **$IF**(*condition*, t_1, t_2), demodulation deviates from its inside-out behavior. The term *condition* is demodulated (evaluated); if the result is **$T**, the value of the **$IF** term is the result of demodulating t_1; if the result is **$F**, the value of the **$IF** term is the result of demodulating t_2; if the result is neither **$T** nor **$F**, demodulation returns to its normal behavior. Note that if *condition* evaluates to a Boolean value, demodulation strays from its inside-out behavior, because just one of t_1 and t_2 is demodulated. If the outside-in demodulation option has been set, there is no need to treat **$IF** terms differently from the norm, because outside-in demodulation causes the **$IF** term to be evaluated before either t_1 or t_2.

The evaluable functions and predicates enable the use of equalities with demodulation as a general-purpose equational programming language. Here are some example functions.

```
(gcd(x,y) =                                  % greatest common divisor
   $IF($EQ(x,0),
       y,
       $IF($EQ(y,0),
           x,
           $IF($LT(x,y),
               gcd(x,$DIFF(y,x)),
               gcd(y,$DIFF(x,y)))))).

(member(x,□) = $F).                          % some list functions
(member(x,[y|z]) = $IF($ID(x,y),
                       $T,
                       member(x,z))).
(append(□,x) = x).
(append([x|y],z) = [x|append(y,z)]).

(reverse([]) = []).
(reverse([x|y]) = append(reverse(y),[x])).
```

A Boolean function defined with demodulators, such as **member** in the preceding set of definitions, can be used as an antecedent (negated literal) in a hyperresolution nucleus in the following way:

$-L_1 \mid \cdots \mid$ **$NOT(member(***element*, *list***))** $\mid \cdots \mid -L_n \mid M$.

Evaluable functions and predicates are very useful when using hyperresolution to perform state-space searches. An example is the Missionaries and Cannibals puzzle:

> There are 3 missionaries, 3 cannibals, and a boat on the west bank of a river. All wish to cross, but the boat holds at most 2 people. If the cannibals ever outnumber the missionaries on either bank of the river or in the boat, the outnumbered missionaries will be eaten. Can they all safely cross the river? If so, how? (The boat cannot cross empty.)

[start of input file]

```
%
%  State(X,Y,Z) means that X missionaries, Y cannibals,
%  and the boat are on the Z side of the river.
%

set(hyper_res).

list(axioms).

-State(xmbs,xcbs,xbp)          % If we have a provable state,

    | -pick(xm)                      % missionaries to cross
    | -pick(xc)                      % cannibals to cross
    | -$LE(xm, xmbs)
    | -$LE(xc, xcbs)
    | -$GT($SUM(xm, xc), 0)          % if number in boat > 0,
    | -$LE($SUM(xm, xc), 2)          % if number in boat <= 2,
    | -$OR($GE(xm, xc),$EQ(xm,0))    % if no feast in the boat,

       % if no feast after the boat leaves current side,

    | -$OR($GE($DIFF(xmbs, xm), $DIFF(xcbs, xc)),$EQ($DIFF(xmbs, xm),0))

       % if no feast when the boat arrives at the other side,

    | -$OR($GE($SUM($DIFF(3, xmbs), xm), $SUM($DIFF(3, xcbs), xc)),
          $EQ($SUM($DIFF(3, xmbs), xm),0))

       % then a crossing can occur

    | State($SUM($DIFF(3, xmbs), xm),$SUM($DIFF(3, xcbs), xc),Otherside(xbp)).

pick(0).
pick(1).
pick(2).

-State(3,3,East).   % goal state

end_of_list.

list(sos).
State(3,3,West).    % initial state
end_of_list.

list(demodulators).
(Otherside(West) = East).
(Otherside(East) = West).
end_of_list.
```

[end of input file]

7 Weighting

OTTER maintains four lists of weight templates.

```
weight_list(pick_given).      % Choose given clauses from the set of support.
weight_list(purge_gen).       % Is used in conjunction with the max_weight
                              % parameter to discard undesirable generated clauses.
weight_list(pick_and_purge).  % Plays the roles of both pick_given and purge_gen
                              % (if present, neither pick_given nor purge_gen
                              % can be present).
weight_list(terms).           % Used to orient equality literals and to decide
                              % dynamic demodulators.
```

See Section 3.2.4 for input of lists of weight templates.

7.1 Weighing Clauses and Literals

The weight of a clause is always the sum of the weights of its literals (excluding any answer literals). The weight of a positive literal is the weight of its atom. The weight of a negative literal is the weight of its atom plus the value of the **neg_weight** parameter (Section 4.2.5).

7.2 Weighing Atoms and Terms

Atoms and terms are weighed top-down. To weigh a given term, the appropriate weight list is searched (in the order input) for the first matching template. If a match is found, then the subterms of the given term that match the integers in the template are weighed. The weight of the given term is the sum of the products of each integer and the weight of its corresponding subterm, plus the second argument of the weight template. For example, the template

```
weight(f(g(2),-3), -50)
```

matches the given term

```
f(g(h(a)),f(b,x))
```

The weight of the given term is (2 * (the weight of `h(a)`)) + (-3 * (the weight of `f(b,x)`)) + (-50). If a matching weight template is not found, then the weight of the given term is 1 + sum of the weights of the subterms. (See the flags **atom_wt_max_args** and **term_wt_max_args** for overrides.) Note that this weighting scheme implies that if no weight templates are present, the default weight of a term or atom is the number of variable, constant, function, and predicate symbols (symbol count).

Variables in weight templates are generic. A variable in a weight template will match any variable (and only variables) in the given term. As a consequence, it is never necessary to use different variable names in a weight template. For example, `weight(f(x,x),-7)` matches the term `f(u,v)`, and `weight(x,32)` matches all variables.

Warning: The two occurrences of symbol `f` in the term `f(f,x)` are treated by OTTER as different symbols because they have different arities. The weight template `weight(f, 0)` applies to the second occurrence but not to the first. (This warning applies only if the `clear(check_arity)` command has been issued.)

8 Answer Literals

The main use of answer literals is to record, during a search for a refutation, instantiations of variables in input clauses. For example, if the theorem under consideration states that an object exists, then

the denial of the theorem contains a variable, and an answer literal containing the variable can be appended to the denial. If a refutation is found, then the empty clause has an answer literal that contains the object whose existence has just been proved.

Any literal whose predicate symbol starts with **$ans**, **$Ans**, or **$ANS** is an answer literal. Most routines—including the ones that weigh clauses, count literals, and decide if a clause is positive or negative—ignore any answer literals. The inference rules insert, into the children, the appropriate instances of any answer literals in the parents. If factoring is enabled, OTTER *does* attempt to factor answer literals.

9 Limits, Abnormal Ends (ABENDS), and Fixes

OTTER has a number of compile-time limits. If a limit is exceeded, a message containing the name of the limit will appear in the output file and/or at the terminal. To raise the limit, find the appropriate definition (**#define**) in a **.h** or **.c** file, increase the limit, and recompile OTTER. (Of course, you must have your own copy of the source code to do this.) Some of the limits are

MAX_NAME — Maximum number of characters in a variable, constant, function, or predicate symbol.

MAX_BUF — Maximum number of characters in an input string (clause, formula, command, weight template, etc.).

MAX_VARS — Maximum number of distinct variables in a clause.

If OTTER is using too much memory, one can decrease (down to 0) the value of the **fpa_literals** parameter, set the **for_sub_fpa** flag to switch forward subsumption indexing from discrimination tree to FPA indexing, and use weighting to discard (more) generated clauses.

At present, demodulation with discrimination tree indexing (the default) does not allow more than one demodulator with the same left side. If demodulation is being used and OTTER exits with a message something like "**ABEND, two demodulators with the same left side**", one can get around the problem by disabling discrimination tree indexing with the command **set(demod_linear)**. (It makes sense to have two demodulators with the same left side only if lex-dependent demodulation is being used.)

10 Summary of the Options and their Defaults

```
set(for_sub).                    clear(dynamic_demod).
set(print_kept).                 clear(dynamic_demod_all).
set(back_sub).                   clear(back_demod).
set(print_back_sub).             clear(knuth_bendix).
set(demod_history).              clear(new_functions).
set(print_new_demod).            clear(lex_order_vars).
set(print_back_demod).           clear(for_sub_fpa).
set(symbol_elim).                clear(no_fapl).
set(print_given).                clear(no_fanl).
set(check_arity).                clear(process_input).
set(print_proofs).               clear(simplify_fol).
                                 clear(print_weight).
clear(binary_res).               clear(sos_queue).
clear(hyper_res).                clear(free_all_mem).
clear(ur_res).                   clear(bird_print).
clear(para_into).                clear(atom_wt_max_args).
```

```
clear(para_from).
clear(demod_inf).
clear(para_from_left).
clear(para_from_right).
clear(para_from_vars).
clear(para_into_vars).
clear(para_ones_rule).
clear(para_all).
clear(no_para_into_left).
clear(no_para_into_right).
clear(print_gen).
clear(order_eq).
clear(sort_literals).
clear(unit_deletion).
clear(factor).
clear(demod_linear).
clear(demod_out_in).
clear(dollar_eval).
clear(term_wt_max_args).
clear(print_lists_at_end).

assign(report, 0).
assign(max_seconds, 0).
assign(max_gen, 0).
assign(max_kept, 0).
assign(max_given, 0).
assign(max_mem, 0).
assign(max_literals, 0).
assign(max_weight, 0).
assign(fpa_literals, 3).
assign(fpa_terms, 3).
assign(demod_limit, 100).
assign(max_proofs, 1).
assign(neg_weight, 0).
assign(stats_level, 2).
```

References

[1] Chang, C., and Lee, R. C., *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

[2] Jouannaud, J.-P. (ed.), *Rewriting Techniques and Applications*, Springer-Verlag Lecture Notes in Computer Science #202 (1985).

[3] Kapur, D. and Zhang, H., *RRL: A Rewrite Rule Laboratory—A User's Manual*, General Electric R & D Center (June 1987).

[4] Kapur, D., and Zhang, H., "Proving equivalence of different axiomatizations of free groups", *J. Automated Reasoning* 4(3), 331–352 (1988).

[5] Loveland, D., *Automated Theorem Proving*, North Holland, Amsterdam (1978).

[6] Lusk, E., and Overbeek, R., *The Automated Reasoning System ITP*, Report ANL-84-27, Argonne National Laboratory, Argonne, Ill. (April 1984).

[7] Lusk, E., and Overbeek, R. (eds.), *Proceedings of the 9th International Conference on Automated Deduction*, Springer-Verlag Lecture Notes in Computer Science #310 (1988).

[8] Siekmann, J. (ed.), *Proceedings of the 8th International Conference on Automated Deduction*, Springer-Verlag Lecture Notes in Computer Science #230 (1986).

[9] Smith, B., *Reference Manual for the Environmental Theorem Prover: An Incarnation of AURA*, Report ANL-88-2, Argonne National Laboratory, Argonne, Ill. (March 1988).

[10] Wos, L., Overbeek, R., Lusk, E., and Boyle, J., *Automated Reasoning: Introduction and Applications*, Prentice-Hall, Englewood Cliffs, N.J. (1984).

[11] Wos, L., Pereira, F., Boyer, R., Moore, J, Bledsoe, W., Henschen, L., Buchanan, B., Wrightson, G., and Green, C., "An overview of automated reasoning and related fields", *J. Automated Reasoning* 1(1), 5–48 (1985).

[12] Wos, L., *Automated Reasoning: 33 Basic Research Problems*, Prentice-Hall, Englewood Cliffs, N.J. (1988).

Internal:

J. M. Beumer (175)
F. Y. Fradin
H. G. Kaper
A. B. Krisciunas
W. W. McCune (50)
G. W. Pieper
D. P. Weber

ANL Patent Department
ANL Contract File
ANL Libraries
TIS Files (3)

External:

DOE-OSTI, for distribution per UC-405 (66)
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
J. L. Bona, Pennsylvania State University
T. L. Brown, University of Illinois, Urbana
P. Concus, Lawrence Berkeley Laboratory
S. Gerhart, Microelectronics and Computer Technology Corp., Austin, TX
H. B. Keller, California Institute of Technology
J. A. Nohel, University of Wisconsin, Madison
M. J. O'Donnell, University of Chicago