ANL-90/9

OTTER 2.0 Users Guide

by W. 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
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161

Price: Printed Copy A03
Microfiche A01

Distribution Category:
Mathematics and Computer
Science (UC-405)

ANL-90/9

ANL--90/9

DE90 007978

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

OTTER 2.0 Users Guide

by

William W. McCune

Mathematics and Computer Science Division

March 1990

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.

Contents

OTTER 2.0 Users Guide

by

William W. McCune

Abstract

OTTER (Organized 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, evaluable functions and predicates, and Knuth-Bendix completion. OTTER is coded in C, it is free, and it is portable to many different kinds of computer.

1 Introduction

OTTER (Organized Techniques for Theorem-proving and Effective Research) is a resolution-style theorem prover, similar in scope and purpose to the AURA [12] and LMA/ITP [9] 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 either with first-order formulas or with clauses (a clause is a disjunction with implicit universal quantifiers and no 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.
- A variant of the Knuth-Bendix method can search for a complete set of reductions.
- 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 inference 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.

There have been two previous releases of OTTER—version 0.9 was distributed at CADE-9 in May 1988, and version 1.0 was released in January 1989. Summaries of other theorem-proving systems can be found in the proceedings of recent CADE meetings [11, 8].

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 [15], [1], [7], or [14] for an introduction to automated theorem proving, and see [13] for an overview of the field.

1.1 Major Changes Since Version 1.0

1. OTTER can now use a termination ordering (recursive path ordering with status) to ensure that demodulation terminates. See Section 5.2.
2. Users now have the option of using Prolog-style (upper-case) variables. See Section 4.1.7.
3. New options can be used to restrict paramodulation from and/or into unit clauses and to prevent paramodulation into subterms of Skolem expressions. See Section 4.1.3.
4. Other paramodulation options and defaults have changed. See Section 4.1.3.
5. New options control selection of the given clause. See Section 4.1.1.
6. Atoms as well as terms can be rewritten by demodulation. See Section 9.2.

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' 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. If new_cl has 0 literals, a refutation has been found.
14. If new_cl has 1 literal, then search axioms and sos for
        unit conflict (refutation) with new_cl.
15. (optional) Print the proof if a refutation has been found.
16. (optional) Try to make new_cl into a demodulator.

17. (optional) Back demodulate if Step 16 made new_cl into a demodulator.
18. (optional) Discard each clause in axioms and each clause in sos that
        is subsumed by new_cl (back subsumption).
19. (optional) Factor new_cl and process factors.
```

Steps 17–19 are delayed until steps 1–16 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`. (The option `prolog_style_variables` says that variables start with upper-case letters. See Section 4.1.7.) 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)`. (See Section 4.1.7.)

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`, $[t_1 \,|\, t_2]$ an abbreviation for `$cons`(t_1, t_2), and $[t_1, t_2, t_3, t_4]$ an

abbreviation for $cons(t_1,$cons(t_2,$cons(t_3,$cons(t_4,$nil)))). The notation [t_1,t_2|t_3] is *not* allowed—such an expression must be written [t_1|[t_2|t_3]].

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 = t_2) and (t_1 != 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 <-> G) and (F -> G) are formulas.
3. If $F_1,\ldots,F_n$ are formulas, then (F_1 | $\cdots$ | F_n) and (F_1 & $\cdots$ & 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 "not", <-> 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
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
skolem(symbol_list).               % identify skolem functions
lex(symbol_list).                  % assign an ordering on symbols
lrpo_lr_status(symbol_list).       % specify RPO status
lrpo_rl_status(symbol_list).       % specify RPO status
```

Table 1: Commands

There are a few constraints on the order of commands.

- Options that affect input and output, such as `check_arity`, `prolog_style_variables`, `simply_fol`, and `bird_print`, should occur before any clause lists or weight lists.
- If the command `set(lex_rpo)` is present, it should occur before any demodulators.

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(binary_res).                   % switch on binary resolution
clear(back_sub).                   % do not use back subsumption
assign(max_seconds, 300).          % stop after about 300 CPU 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 The Commands skolem, lex, lrpo_lr_status, and lrpo_rl_status

Each of the commands **skolem**, **lex**, **lrpo_lr_status**, and **lrpo_rl_status** takes a list of terms as an argument. The **lex** command specifies an ordering on symbols, and the others give properties to symbols. An example is

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

The arguments of **f** and **g** serve as place-holders only; they identify **f** and **g** as function or predicate symbols and specify the arity.

skolem([...]). The **skolem** command identifies constant and function symbols as Skolem symbols. (If the user inputs quantified formulas and OTTER Skolemizes, this command is not necessary.) The Skolem property is used by the options **para_skip_skolem** (Section 4.1.3) and **delete_identical_nested_skolem** (Section 4.1.4).

lex([...]). The **lex** command specifies an ordering on function and constant symbols. Lexical ordering on terms is used in three contexts: orienting equality literals (Sections 5.1.3 and 5.2.3), deciding whether to apply a lex-dependent demodulator (Sections 5.1.2 and 5.2.2), and evaluating functions/predicates that perform lexical comparisons (Section 6).

lrpo_lr_status([...]) and lrpo_rl_status([...]). These commands specify status with respect to the recursive path ordering. See Section 5.2.

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 is informed when such a change occurs.

4.1 Flags

4.1.1 Main Loop Flags

A given clause is taken from **sos** at the beginning of each iteration of the main loop. The default is to take the lightest clause with respect to either **weight_list(pick_given)** or **weight_list(pick_and_purge)**. If neither weight list is present, the weight of a clause is its number of symbols. Both lists cannot be present. See Section 7.

input_sos_first — default clear. If this flag is set, the input clauses in **sos** are chosen (in order) as the first given clauses; then the lightest clauses are chosen.

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

sos_stack — default clear. If this flag is set, the last clause in **sos** becomes the given clause (the set of support list operates as a stack).

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

4.1.2 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.

4.1.3 Paramodulation Flags

para_from_left — default set. 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 set. 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_into_left — default set. If this flag is set, then allow paramodulation *into* left arguments of positive and negative equalities. (Applies to both **para_into** and **para_from** inference rules.)

para_into_right — default set. If this flag is set, allow paramodulation *into* right arguments of positive and negative equalities. This flag is one of the options to be cleared when searching for a complete set of reductions. (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_from_units_only — default clear. If this flag is set, paramodulate only if the *from* clause is a unit (equality). (Applies to both **para_into** and **para_from** inference rules.)

para_into_units_only — default clear. If this flag is set, paramodulate only if the *into* clause is a unit. (Applies to both **para_into** and **para_from** inference rules.)

para_skip_skolem — default clear. If this flag is set, do not paramodulate *into* subterms of Skolem expressions [10]. (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.)

4.1.4 Flags for Handling Generated Clauses

(Section 4.1.5 gives additional, equality-related flags for handling generated clauses.)

very_verbose — default clear. If this flag is set, much information about the processing of generated clauses is output.

order_eq — default clear. If this flag is set, flip equalities if the right side is heavier than the left. See Sections 5.1.3 and 5.2.3 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.

delete_identical_nested_skolem — default clear. If this flag is set, delete a clause if a Skolem expression (properly) contains an occurrence of its leading Skolem symbol. For example, if **f** is a Skolem function, delete a clause if it has a term **f(f(x))** or a term **f(g(f(x)))**.

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 units only are being generated.)

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

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

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.

4.1.5 Demodulation and Equality Flags

demod_history — default set. If this flag is set, then when a clause is demodulated, include the numbers of the demodulators 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 flag 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.

demod_out_in — default clear. If this flag is set, demodulate terms outside-in, left-to-right. In other words, the program attempts 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).

dynamic_demod — default clear. If this flag is set, attempt to make *some* newly kept equalities into demodulators (Sections 5.1.4 and 5.2.4). 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 5.1.4). 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 **demodulators**, **axioms**, and **sos** 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**. (*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 orient new demodulators, 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. Setting the knuth_bendix flag automatically causes the following flags to be altered, if necessary, as follows: set(para_from), set(para_into), set(para_from_left), clear(para_from_right), set(para_into_left), clear(para_into_right), set(dynamic_demod_all), set(back_demod). See Section 5.3 for an example. The user may wish to also set(lex_rpo) (see next flag).

lex_rpo — default clear. If this flag is set, then use the lexicographic recursive path ordering (also called RPO with status) to compare terms. If this flag is clear, weight templates and lexicographic order are used. See Section 5.2.

dynamic_demod_lex_dep — default clear. If this flag is set, dynamic demodulators may be lex-dependent or LRPO-dependent. See Sections 5.1.4 and 5.2.4.

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. *If* lex_rpo *is set,* lex_order_vars *has no effect on demodulation.* See Section 5.1.1 for more detail.

4.1.6 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.

4.1.7 Input and Output Flags

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.

prolog_style_variables — default clear. If this flag is set, a name with no arguments in a clause is a variable if and only if it starts with A through Z (upper case) or _.

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, and 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.)

bird_print — default clear. If this flag is set, output terms constructed with the binary function a 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.

4.1.8 Miscellaneous Flags

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 numbers in the "in use" column of the memory statistics should all be close to 0. This flag is used primarily for system debugging.

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.

really_delete_clauses — default clear. If this flag is clear, clauses that are deleted by back subsumption or back demodulation are not really removed from memory; they are retained in a special place so that they can be printed if they occur in a proof. If the job involves much back subsumption or back demodulation and if memory conservation is important, these deleted clauses can be removed from memory by setting this flag (and any

proof containing such a clause will not be printed in full).

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 CPU seconds. The time 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 CPU seconds. The time 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. The number 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. The number 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.

max_mem — default 0, range [0..MAX_INT]. If n is not 0, then OTTER will terminate the search before more than n Kbytes have been dynamically allocated (**malloc**).

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; see 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, but it can be faster. 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, but it can be faster. 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. The count includes **$** symbol evaluation. 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**, **dynamic_demod_all**, **lex_rpo**, and **dynamic_demod_lex_dep**, and it gives all the rules—built in and optional—for orienting equality literals and deter-

mining dynamic demodulators. In this section, α and β always refer to the left and right arguments, respectively, of the equality literal under consideration.

Through the years, we have accumulated a collection of ad hoc ordering techniques, which are presented in Section 5.1. However, simpler and more predictable behavior can occur if the Lexicographic Recursive Path Ordering (LRPO, flag `lex_rpo`) is used. Section 5.1 applies if `lex_rpo` is clear, and Section 5.2 applies if `lex_rpo` is set.

The `lex` command, which applies to both sections, can be used to assign an ordering on symbols. 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). Behavior is undefined if relevant symbols are omitted from the lex command.

5.1 lex_rpo Is Clear

5.1.1 Lexical Order (lex_rpo is clear)

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 is 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. If $a \prec b$, then `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 5.1.2), evaluating functions/predicates that perform lexical comparisons (Section 6), and orienting equality literals (Section 5.1.3). When orienting equality literals, partial lexical ordering is used, even if the flag `lex_order_vars` is set.

5.1.2 Lex-dependent Demodulation (lex_rpo is clear)

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 5.1.4 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).
  (or(x,y) = or(y,x)).
  (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).

5.1.3 Orienting Equalities (`lex_rpo` is clear)

Orienting equality literals (positive and negative) except positive unit equalities. The arguments α and β are weighed (Section 7) by 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 5.1.1); 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 4.1.5), 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 $vars(\alpha) \subseteq vars(\beta)$, the literal is flipped.

5.1.4 Determining Dynamic Demodulators (`lex_rpo` is clear)

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. (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 $\alpha \succ \beta$),

(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 `dynamic_demod_lex_dep` is set, if α and β are incomparable, if they are identical-except-variables (Section 5.1.2), and if $vars(\alpha) \supseteq vars(\beta)$, then the equality becomes a lex-dependent demodulator.

5.2 `lex_rpo` Is Set

5.2.1 Lexical Order (`lex_rpo` is set)

To use lexicographic recursive path ordering (LRPO, or RPO with status) [2, 3, 5] the user must assign an ordering on constant and function symbols. This is accomplished with the `lex` command. (OTTER uses a total ordering. Other implementations of LRPO use partial orderings or dynamically changing orderings.) If a `lex` command is not present, OTTER assigns an ordering, which is usually ineffective.

The total ordering on symbols extends to a total ordering on ground terms. The lexical aspect of LRPO is that certain specified function symbols can have their arguments compared left-to-right or right-to-left. This is accomplished with the commands `lrpo_lr_status` and `lrpo_rl_status`. For example, if associativity is a demodulator, `f(f(x,y),z) = f(x,f(y,z))`, and if expressions in `f` are to be right associated, one can use the command

```
lrpo_lr_status( [f(x,x)] ).
```

LRPO comparison is used when orienting equality literals, deciding whether an equality should be a demodulator or an LRPO-dependent demodulator, and deciding whether to apply an LRPO-dependent demodulator. LRPO comparison is never used when evaluating the functions/predicates that perform lexical comparison (`$LLT`, `$LGT`, etc.).

5.2.2 Lex-dependent Demodulation (`lex_rpo` is set)

The notion of lex-dependent demodulator is replaced with that of LRPO-dependent demodulator. An input demodulator becomes LRPO-dependent if neither argument is LRPO-less-than the other. An LRPO-dependent demodulator is allowed to rewrite a term iff its application results in an LRPO-less-than term.

5.2.3 Orienting Equalities (`lex_rpo` is set)

All equality literals (positive and negative) are oriented in the same way. If alpha is LRPO-less-than beta, the literal is flipped.

5.2.4 Determining Dynamic Demodulators (lex_rpo is set)

If the flag dynamic_demod is set, OTTER attempts to make all equalities into demodulators (dynamic_demod_all is ignored). If beta is LRPO-less-than alpha, the derived equality becomes a demodulator. (Alpha is not LRPO-less-than alpha, because orienting has already occurred.) If dynamic_demod_lex_dep is set, neither argument is LRPO-less-than the other, and every variable that occurs in beta also occurs in alpha, the derived equality becomes an LRPO-dependent demodulator.

5.3 Completion and Termination

LRPO enables an implementation of the Knuth-Bendix completion procedure [6]. Here is an input file that causes OTTER to search for and quickly find a complete set of reductions for free groups. (As with any application of the Knuth-Bendix completion procedure, the critical issue is the choice of ordering scheme and/or the specific ordering on symbols.)

```
set(knuth_bendix).
set(lex_rpo).
set(process_input).
set(print_lists_at_end).

lex([e,f(x,x),g(x)]).
lrpo_lr_status([f(x,x)]).

list(sos).
(x = x).
(f(e,x) = x).                  % left identity
(f(g(x),x) = e).               % left inverse
(f(f(x,y),z) = f(x,f(y,z))).   % associativity
end_of_list.
```

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`, `$OR`
bool → *bool*	`$TRUE`, `$NOT`
→ *bool*	`$T`, `$F`
term → *bool*	`$ATOMIC`, `$NUMBER`, `$VAR`
→ *int*	`$NEXT_CL_NUM`
bool × *term* × *term* → *term*	`$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` (Sections 4.1.5 and 5.1.1) has effect.

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*. The 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. *term* → *bool*. A term is `$ATOMIC` iff it is a constant (including integer), a term is a `$NUMBER` iff it is an integer, and a term is a `$VAR` iff it is a (unbound) variable.

8. → *int*. The term `$NEXT_CL_NUM` (no arguments) evaluates to the next integer that will be assigned as a clause identifier (this is useful for placing the ID of a clause within the clause).

9. *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 (prolog_style_variables)
(member(X,[H|T]) = $IF($ID(X,H),
                       $T,
                       member(X,T))).

(reverse(L) = rev2(L,[])).
(rev2([], L) = L).
(rev2([H|T],L) = rev2(T,[H|L])).
```

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:

$-L_1$ | $\cdots$ | -$TRUE(member(*element*, *list*)) | $\cdots$ | $-L_n$ | M.

Evaluable functions and predicates are 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 (lex_rpo clear).
```

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`, Section 4.1.8 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 command `clear(check_arity)` has been issued.)

The default weight of an answer literal is 0, but templates can be used to assign weights to answer literals. The parameter `neg_weight` never applies to answer literals.

Because of an inadequacy of the input parser, weight templates cannot contain infix equality atoms—the template `weight((a = b), 0)` will *not* be accepted. Instead, one must use the internal system name and write the atom in prefix form—the template `weight($eq_infix(a,b), 0)` matches the atom `(a = b)`.

If one wishes to have a weight template containing a Skolem function or constant that is generated by OTTER, one must first make a short trial run to find out how the formulas are Skolemized, then return to the input file and insert the weight list *after* the formula lists.

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 count literals and decide whether 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 Meta-experimental Features

This section describes some features that are new, not well tested, and/or not well documented.

9.1 Linked UR-Resolution

OTTER 2.0 has an inference rule, `linked_ur_res`, which is an application of the linked inference principle [16] to UR-resolution. As this manual is written, there is not yet any documentation. The inference rule is still evolving and is highly experimental. For current information on the status of linked UR-resolution, send e-mail to wos@mcs.anl.gov, veroff@unmvax.cs.unm.edu, and karonis@mcs.anl.gov.

9.2 Atom (Literal) Demodulation

Atoms, as well as terms, are demodulated.

- The negation symbol cannot occur in a demodulator.

- Because of an inadequacy of the input parser, demodulators to rewrite infix equalities must use the internal (prefix) system name `$eq_infix`. For example, the demodulator `((f(x,y) = x) = (y = 0))` must be written `($eq_infix(f(x,y),x) = $eq_infix(y,0))`
- Things work as expected if the right side of a literal demodulator is one of the Boolean constants `$T` or `$F`.

9.3 Conditional Demodulation

A conditional demodulator has the form

`$CONDITIONAL(`*condition*`,` *alpha*`,` *beta*`).`

It is read "if *condition* then *alpha* = *beta*". A conditional demodulator is applied only if the instantiated *condition* evaluates to `$T`. In other respects, it behaves as a regular demodulator. Examples are (`member` and `gcd` are defined in Section 6)

```
$CONDITIONAL($ATOMIC(x), conjunctive_normal_form(x), x).
$CONDITIONAL(member(gcd(4,x),y), f(x,y), g(y)).
$CONDITIONAL($GT($NEXT_CL_NUM,1000), e(x,x), junk).
```

9.4 Another Demodulation Trick

A new feature, activated by the `set_special_unary` command, allows OTTER to avoid one of the problems caused by the lack of associative-commutative matching during demodulation. The feature is useful when an associative-commutative function and an inverse are present, as in rings. Without this feature, the following `lex` command and demodulators

```
lex([0,a,b,c,d,e,g(x),f(x,x)]).

list(demodulators).
(f(x,y) = f(y,x)).
(f(x,f(y,z)) = f(y,f(x,z))).
(f(x,g(x)) = 0).
(f(x,f(g(x),y)) = f(0,y)).
(f(0,x) = x).
end_of_list.
```

will cause the expression

```
f(f(f(g(b),a),c),f(b,g(c)))
```

to be sorted into

```
f(a,f(b,f(c,f(g(b),g(c))))).
```

One would like b and g(b) to be next to each other so that they could be canceled by one of the inverse demodulators. The new feature accomplishes just that. The new command

```
set_special_unary([g(x)])
```

causes g to be ignored during term comparisons, and the expression would be demodulated to a. The **set_special_unary** command has no effect if the flag **lex_rpo** is set. *This is an experimental feature. Its consequences have not been well analyzed.*

9.5 Introducing New Functions

When searching for a complete set of reductions, one sometimes encounters an equality that cannot be oriented, but that can be handled by introducing a new function symbol. See [4] for more detail.

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. The lexical value of the new symbol is low.

9.6 Ancestor Subsumption

OTTER does not necessarily prefer short or simple proofs—it simply reports the proofs that it finds. A new option **ancestor_subsume** extends the concept of subsumption to include the derivation history, so that if two clause occurrences are logically identical, the one with fewer ancestors is preferred. The motivation is to find short proofs.

ancestor_subsume — default clear. If this flag is set, the notion of subsumption (forward and backward) is replaced with *ancestor-subsumption.* Clause C ancestor-subsumes clause D iff C properly subsumes D or if C and D are variants and $size(ancestorset(C)) \leq size(ancestorset(D))$.

9.7 Reducing max_weight on the Fly

In many searches, the number of kept clauses grows much faster than the number of given clauses. In other words, the list **sos** is very large, and most of those clauses never participate in the search. To save memory, one can use the **max_weight** parameter to discard many of the clauses that will (probably) never become given clauses.

A few searches and proofs show a phenomenon we call the *complexity hump.* To get a search started, one must use complex clauses; then one can continue the search using simpler clauses. Analogously, the first few steps in a proof are complex, and the remaining steps are simpler. If one needs to carefully conserve memory when a complexity hump is present,

one can use the parameter **reduce_weight_limit** to change the value of **max_weight** after a specified number of given clauses.

reduce_weight_limit — default 0, range [0..MAX_INT]. If n (the value) is not 0, this parameter has effect. Two integers are packed in the parameter. The last two digits specify the new limit, and the others specify the given clause at which to make the change. For example, $n = 3975$ says to reduce **max_weight** to 75 after given clause number 39.

10 Limits, Abnormal Ends (ABENDS), and Fixes

OTTER has a several 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. (One must have his or her 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.

MAX_FS_TERM_DEPTH — Maximum depth of terms in forward subsumption discrimination tree.

MAX_AL_TERM_DEPTH — Maximum depth of left arguments of equalities in demodulation discrimination tree.

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.

11 Summary of the Options and Their Defaults

```
clear(input_sos_first).
clear(sos_queue).
clear(sos_stack).
set(print_given).

clear(binary_res).
clear(hyper_res).
clear(ur_res).
clear(para_into).
clear(para_from).
clear(demod_inf).

clear(knuth_bendix).
clear(lex_rpo).
clear(dynamic_demod_lex_dep).
clear(lex_order_vars).

clear(for_sub_fpa).
clear(no_fapl).
clear(no_fanl).

set(check_arity).
clear(prolog_style_variables).
```

```
set(para_from_left).
set(para_from_right).
set(para_into_left).
set(para_into_right).
clear(para_from_vars).
clear(para_into_vars).
clear(para_from_units_only).
clear(para_into_units_only).
clear(para_skip_skolem).
clear(para_ones_rule).
clear(para_all).

clear(very_verbose).
clear(order_eq).
clear(sort_literals).
clear(delete_identical_nested_skolem).
set(for_sub).
clear(unit_deletion).
set(print_kept).
set(print_proofs).
set(back_sub).
set(print_back_sub).
clear(factor).

set(demod_history).
clear(demod_linear).
clear(demod_out_in).
clear(dollar_eval).
clear(dynamic_demod).
clear(dynamic_demod_all).
set(print_new_demod).
clear(back_demod).
set(print_back_demod).
set(symbol_elim).
```

```
clear(process_input).
clear(simplify_fol).
clear(bird_print).

clear(free_all_mem).
clear(atom_wt_max_args).
clear(term_wt_max_args).
clear(print_lists_at_end).
clear(really_delete_clauses).

clear(prog_synthesis).
clear(ancestor_subsume).
clear(new_functions).
clear(linked_ur_res).
clear(linked_ur_trace).


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).
assign(reduce_weight_limit,0).
assign(max_ur_depth,5).
assign(max_ur_deduction_size,20).
```

Acknowledgements

Ross Overbeek, being the chief designer of OTTER's predecessors AURA and LMA/ITP, was an important influence during the design of OTTER. I am also grateful to the users of previous versions of OTTER who reported bugs and made many other valuable comments.

References

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

[2] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.

[3] J.-P. Jouannaud, editor. *Rewriting Techniques and Applications, Springer-Verlag Lecture Notes in Computer Science, Vol. 202*, New York, 1985. Springer-Verlag.

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

[5] D. Kapur and H. Zhang. RRL: Rewrite rule laboratory user's manual. Technical Report 89-03, Department of Computer Science, University of Iowa, 1989.

[6] D. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebras.* Pergamon Press, 1970.

[7] Donald Loveland. *Automated Theorem Proving: A Logical Basis.* North-Holland, Amsterdam, 1978.

[8] E. Lusk and R. Overbeek, editors. *Proceedings of the 9th International Conference on Automated Deduction, Springer-Verlag Lecture Notes in Computer Science, Vol. 310*, New York, 1988. Springer-Verlag.

[9] Ewing Lusk and Ross Overbeek. The automated reasoning system ITP. Tech. Report ANL-84/27, Argonne National Laboratory, Argonne, Ill., April 1984.

[10] William McCune. Skolem functions and equality in automated deduction. Preprint MCS-P136-0290, Argonne National Laboratory, Argonne, Ill., February 1990.

[11] J. Siekmann, editor. *Proceedings of the 8th International Conference on Automated Deduction, Springer-Verlag Lecture Notes in Computer Science, Vol. 230*, New York, 1986. Springer-Verlag.

[12] Brian Smith. Reference manual for the environmental theorem prover: An incarnation of AURA. Tech. Report ANL-88-2, Argonne National Laboratory, Argonne, Ill., March 1988.

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

[14] Larry Wos. *Automated Reasoning: 33 Basic Research Problems.* Prentice-Hall, Englewood Cliffs, N.J., 1988.

[15] Larry Wos, Ross Overbeek, Ewing Lusk, and James Boyle. *Automated Reasoning: Introduction and Applications.* Prentice-Hall, Englewood Cliffs, N.J., 1984.

[16] Larry Wos, Robert Veroff, Brian Smith, and William McCune. The linked inference principle II: The user's view. In R. Shostak, editor, *Proceedings of the 7th Conference on Automated Deduction, Springer-Verlag Lecture Notes in Computer Science, Vol. 170*, pages 316–332, New York, 1984. Springer-Verlag.

Distribution for ANL-90/9

Internal:

J. M. Beumer (150)
F. Y. Fradin
H. G. Kaper
W. W. McCune (50)
G. W. Pieper
D. Weber
C. L. Wilkinson

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

External:

DOE-OSTI, for distribution per UC-405 (61)
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:

W. W. Bledsoe, The University of Texas, Austin
J. L. Bona, Pennsylvania State University
P. Concus, Lawrence Berkeley Laboratory
E. F. Infante, University of Minnesota
H. B. Keller, California Institute of Technology
M. J. O'Donnell, The University of Chicago
D. O'Leary, University of Maryland