Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-263

OTTER 3.3 Reference Manual

by

William McCune

Mathematics and Computer Science Division

Technical Memorandum No. 263

August 2003

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

Contents

OTTER 3.3 Reference Manual

William McCune

Abstract

OTTER 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 and features 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, Knuth-Bendix completion, and the hints strategy. OTTER is coded in ANSI C, is free, and 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 [24] and LMA/ITP [15] theorem provers, which are also associated with Argonne. OTTER applies to statements written in first-order logic with equality. The primary design considerations have been performance, portability, and extensibility. The programming language ANSI 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 and help with proof searches.

- Weight functions and lexical ordering decide the value of clauses and terms.
- Answer literals can give information about the proofs that are found. See Sec. 11.
- Evaluable functions and predicates build in integer arithmetic, Boolean operations, and lexical comparisons and enable users to "program" aspects of deduction processes. See Sec. 9.
- Proofs can be presented in a very detailed form, called *proof objects*, which can be used by other programs, for example, to check or to translate the proofs. See Sec. 20.1.
- The *hints strategy* can be used to provide heuristic guidance to the search. To apply this feature, the user gives a set of *hint clauses*, and clauses similar to the hint clauses are emphasized during the search. See Sec. 14.
- OTTER's input is compatible, for the most part, with a complementary program MACE 2.0 [17], which looks for finite models of first-order statements. Given a conjecture, OTTER can search for a proof, and MACE can look for a counterexample, usually from the same input file. MACE 2.0 is included in the standard OTTER 3.3 distribution packages.

Although OTTER has an autonomous mode, most work with OTTER involves interaction with the user. After encoding a problem into first-order logic or into clauses, the user usually chooses inference rules, sets options to control the processing of inferred clauses, and decides 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. In the autonomous mode, the user inputs a set of clauses and/or formulas, and OTTER does a simple syntactic analysis and decides inference rules and strategies. The autonomous mode is frequently useful for the first attempt at a proof.

1.1 What OTTER Isn't

Some of the first applications that come to mind when one hears "automated theorem proving" are number theory, calculus, and plane geometry, because these are some of the first areas in which math students try to prove theorems. Unfortunately, OTTER cannot do much in these areas: interesting number theory problems usually require induction, interesting calculus and analysis problems usually require higher-order functions, and the first-order axiomatizations of geometry are not practical. (Nonetheless, Art Quaife has proved many interesting theorems in number theory and geometry using OTTER [22, 21].) For practical theorem proving in inductive theories, see the work of Boyer, Moore, and Kaufmann [2, 10].

OTTER is also not targeted toward synthesizing or verifying formal hardware or software systems. See [6, 5] for work in those areas.

Summaries of other theorem-proving systems can be found in proceedings of the recent Conferences on Automated Deduction (CADE) and in coverage of the CADE ATP System Competition (CASC).

1.2 History, New Features, and Changes

There have been several previous releases of OTTER, starting with version 0.9, which was distributed at the 9th International Conference on Automated Deduction (CADE-9) in May 1988. Many new features have been added since then, many bugs have been fixed, and (of course) many bugs have been introduced.

1.3 Useful Background

This manual does not contain an introduction to first-order logic or to automated deduction. We assume that the reader knows the basic terminology including *term* (*variable*, *constant*, *complex term*), *atom*, *literal*, *clause*, *propositional variable*, *function symbol*, *predicate symbol*, *Skolem constant*, *Skolem function*, *formula*, *conjunctive normal form* (*CNF*), *resolution*, *hyperresolution*, and *paramodulation*. See [3, 14, 32] for an introductions and overviews of automated theorem proving, see [23, 1] for collections of important papers, see [30] for a list of general problems in the field, and see [33, 8, 18] for introductions and applications that focus on the use of OTTER.

2 Outline of OTTER's Inference Process

Once OTTER gets going with its real work—making inferences and searching for proofs—it operates on clauses and on clauses only. If the user inputs nonclausal first-order formulas, OTTER immediately translates them to clauses, by a straightforward procedure involving negation normal form conversion, Skolemization, quantifier operations, and conjunctive normal form conversion.

As with its predecessors AURA and LMA/ITP, OTTER's basic inference mechanism is the *given-clause algorithm*, which can be viewed as a simple implementation of the set of support strategy [31]. OTTER maintains four lists of clauses:

`usable.` This list contains clauses that are available to make inferences.

`sos.` Clauses in list `sos` (set of support) are not available to make inferences; they are waiting to participate in the search.

`passive.` These clauses do not directly participate in the search; they are used only for forward subsumption and unit conflict. The passive list is fixed at input and does not change during the search. See Sec. 12.

`demodulators.` These are equalities that are used as rules to rewrite newly inferred clauses.

The *main loop* for inferring and processing clauses and searching for a refutation operates mainly on the lists `usable` and `sos`:

```
While (sos is not empty and no refutation has been found)
```

```
      1. Let given_clause be the ``best'' clause in sos;
      2. Move given_clause from sos to usable;
      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 usable as its other parents;  new clauses
            that pass the retention tests are appended to sos;
  End of while loop.
```

The set of support strategy requires the user to partition the input clauses into two sets: those with support and those without. For each inference, at least one of the parents must have support. Retained inferences receive support. In other words, no inferences are made in which all parents are nonsupported input clauses. At input time, OTTER's list `sos` is the set of supported clauses, and `usable` is the nonsupported clauses. (Once the main loop has started, `usable` no longer corresponds to nonsupported clauses, because `sos` clauses have moved there.) OTTER's main loop implements the set of support strategy, because no inferences are made in which all of the parents are from the initial `usable` list.

The following paragraph tries to answer the frequently asked question "At a certain point, OTTER *has all of the clauses available to make the inference I want, and one of the potential parents is selected as the given clause—why doesn't the program make the inference?"*

OTTER's main loop eliminates an important kind of redundancy. Suppose one can infer clause C from clauses A and B, and suppose both A and B are in list `sos`. If A is selected as the given clause, it will be moved to `usable` and inferences will be made; but A *will not* mate with B to infer C, because B is still in `sos`. We must wait until B has also been selected as given clause. Otherwise, we would infer C twice. (The redundancy would be much worse with inference rules such as hyperresolution and UR-resolution with which a clause can have many parents.) In general, all parents that participate in an inference must either have been in the initial `usable` list or have been selected as given clauses. (This is not true when demodulators are considered as parents.)

The procedure for processing a newly inferred clause `new_cl` follows; steps marked with `*` are optional.

```
    1.  Renumber variables.
  * 2.  Output new_cl.
    3.  Demodulate new_cl (including $ evaluation).
  * 4.  Orient equalities.
  * 5.  Apply unit deletion.
    6.  Merge identical literals (leftmost copy is kept).
  * 7.  Apply factor-simplification.
  * 8.  Discard new_cl and exit if too many literals or variables.
    9.  Discard new_cl and exit if new_cl is a tautology.
  * 10. Discard new_cl and exit if new_cl is too 'heavy'.
  * 11. Sort literals.
  * 12. Discard new_cl and exit if new_cl is subsumed by any clause
            in usable, sos, or passive (forward subsumption).
    13. Integrate new_cl and append it to sos.
  * 14. Output kept clause.
```

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

Steps 19–21 are delayed until steps 1–18 have been applied to all clauses inferred from the active given clause.

3 Starting OTTER

Although OTTER has a primitive interactive feature (Sec. 15), it is essentially a noninteractive program. On UNIX-like systems it reads from the standard input and writes to the standard output:

`otter <` *input-file* `>` *output-file*

No command-line options are accepted; all options are given in the input file.

4 Syntax

OTTER recognizes two basic types of statement: clauses and formulas. Clauses are simple disjunctions whose variables are implicitly universally quantified. OTTER's searches for proofs operate on clauses. Formulas are first-order statements without free variables—all variables are explicitly quantified. When formulas are input, OTTER immediately translates them to clauses.

4.1 Comments

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.

4.2 Names for Variables, Constants, Functions, and Predicates

Three kinds of character string, collectively referred to as *names*, can be used for variables, constants, function symbols, and predicate symbols:

- An *ordinary name* is a string of alphanumerics, `$`, and `_`.

- A *special name* is a string of characters in the set `` *+-/\^<>=`~:?@&!;# `` (and sometimes `|`).
- A *quoted name* is any string enclosed in two quotation marks of the same type, either `"` or `'`. We have no trick for including a quotation mark of the same type in a quoted name.

(The reason for separating ordinary and special names has to do with infix, prefix, and postfix operators; see Sec. 4.6.) For completeness, we list here the meanings of the remaining printable characters.

- `.` (period) — terminates input expressions.
- `%` — starts a comment (which ends with the end of the line).
- `,` `()` `[]` `{}` (and sometimes `|`) — are punctuation and grouping symbols.

Variables. Determining whether a simple term is a constant or a variable depends on the context of the term. If it occurs in a clause, the symbol determines the type: the default rule is that a simple term is a variable if it starts with `u`, `v`, `w`, `x`, `y`, or `z`. If the flag `prolog_style_variables` is set, a simple term is a variable if and only if it starts with an upper-case letter or with _. (Therefore, variables in clauses must be ordinary names.) In a formula, a simple term is a variable if and only if it is bound by a quantifier.

Reserved and Built-in Names. Names that start with `$` are reserved for special purposes, including evaluable functions and predicates (Sec. 9), answer literals and terms (Sec. 11), and some internal system names. The name = and any name that starts with `eq`, `EQ`, or `Eq`, when used as a binary predicate symbol, is recognized as an equality predicate by the demodulation and paramodulation processes. And some names, when they occur in clauses or formulas, are recognized as logic symbols.

Overloaded Symbols. The user can use a name for more than one purpose, for example as a constant and as a 5-ary predicate symbol. When the flag `check_arity` is set (the default), the user is warned about such uses. Some built-in names are also overloaded; for example, `|` is used both for disjunction and as Prolog-style list punctuation, and although the symbol `-` is built in as logical negation, it can be used for both unary and binary minus as well.

4.3 Terms and Atoms

Recall that, when interpreted, terms are evaluated as objects in some domain, and atoms are evaluated as truth values. Constants and variables are terms. An n-ary function symbol applied to n terms is also a term. An n-ary predicate symbol applied to n terms is an atom. A nullary predicate symbol (also referred to as a propositional variable) is also an atom.

The pure way of writing complex terms and atoms is with *standard application*: the function or predicate symbol, opening parenthesis, arguments separated by commas, then closing parenthesis, for example, `f(a,b,c)` and `=(f(x,e),x)`. If all subterms of a term are written with standard application, the term is in *pure prefix form*. Whitespace (spaces, tabs, newlines, and comments) can appear in standard application terms anywhere *except* between a function or predicate symbol and its opening parenthesis. If the flag `display_terms` is set, OTTER will output terms in pure prefix form.

Infix Equality. Some binary symbols can be written in infix form; the most important is =. In addition, a negated equality, `-(a=b)` can be abbreviated `a!=b`.

List Notation. Prolog-style list notation can be used to write terms that usually represent lists. Table 1 gives some example terms in list notation and the corresponding pure prefix form. Of course, lists can contain complex terms, including other lists.

Table 1: List Notation

`[]`	`$nil`
`[x|y]`	`$cons(x,y)`
`[x,y]`	`$cons(x,$cons(y,$nil))`
`[a,b,c,d]`	`$cons(a,$cons(b,$cons(c,$cons(d,$nil))))`
`[a,b,c|x]`	`$cons(a,$cons(b,$cons(c,x)))`

4.4 Literals and Clauses

A literal is either an atom or the negation of an atom. A clause is a disjunction of literals. The built-in symbols for negation and disjunction are `-` and `|`, respectively. Although clauses can be written in pure prefix form, with `-` as a unary symbol and `|` as a binary symbol, they are rarely written that way. Instead, they are almost always written in infix form, without parentheses. For example, the following is a clause in both forms.

Pure prefix: `|(-(a),|(=(b1,b2),-(=(c1,c2))))`
Infix (abbreviated): `-a | b1=b2 | c1!=c2`

OTTER accepts both forms. (Clauses are parsed by the general term-parsing mechanism presented in Sec. 4.6).

4.5 Formulas

Table 2 lists the built-in logic symbols for constructing formulas.

Formulas in Pure Prefix Form. Although the practice is rarely done, formulas can be written in pure prefix form. Quantification is the only tricky part: there is a special variable-arity symbol, `$Quantified`, for quantified formulas. For example, $\forall xy \exists z (P(x,y,z)|Q(x,z))$ is represented by

Table 2: Logic Symbols

negation	`-`
disjunction	`|`
conjunction	`&`
implication	`->`
equivalence	`<->`
existential quantification	`exists`
universal quantification	`all`

```
$Quantified(all,x,y,exists,z,|(P(x,y,z),Q(x,z))).
```

Abbreviated Formulas. Formulas are usually abbreviated in a natural way. The associativity and precedence rules for abbreviating formulas and the mechanism for parsing formulas are presented in Sec. 4.6. Here are some examples.

Standard Usage	OTTER syntax (abbreviated)
$\forall x P(x)$	`all x P(x)`
$\forall xy \exists z(P(x,y,z) \lor Q(x,z))$	`all x y exists z (P(x,y,z) | Q(x,z))`
$\forall x(P(x) \land Q(x) \land R(x) \rightarrow S(x))$	`all x (P(x) & Q(x) & R(x) -> S(x))`

Note that if a formula has a string of identical quantifiers, all but the first can be dropped. For example, `all x all y all z p(x,y,z)` can be shortened to `all x y z p(x,y,z)`. In expressions involving the associative operations `&` and `|`, extra parentheses can be dropped. Moreover, a default precedence on the logic symbols allows us to drop more parentheses: `<->` has the same precedence as `->`, and the rest in decreasing order are `->`, `|`, `&`, `-`. Greater precedence means closer to the root of the term (i.e., larger scope). For example, the following three strings represent the same formula.

```
p | -q & r -> -s | t.
(p | (-(q) & r)) -> (-(s) | t).
->(|(p,&(-(q),r)),|(-(s),t)).
```

When in doubt about how a particular string will be parsed, one can simply add additional parentheses and/or test the string by having OTTER read it and then display it in pure prefix form. The following input file can be used to test the preceding example.

```
assign(stats_level, 0).
set(display_terms).
formula_list(usable).
p| -q&r-> -s|t.       % This formula has minimum whitespace.
end_of_list.
```

In general, whitespace is required around `all` and `exists` and to the left of `-`; otherwise, whitespace around the logic symbols can be removed. See Sec. 4.6 for the rules.

4.6 Infix, Prefix, and Postfix Expressions

Many Prolog systems have a feature that allows users to declare that particular function or predicate symbols are infix, prefix, or postfix and to specify a precedence and associativity so that parentheses can sometimes be dropped. OTTER has a similar feature. In fact, the clause and formula parsing routines use the feature. Users who use only the predeclared logic operators for clauses and formulas and the predeclared infix equality = can skip the rest of this section.

Prolog users who are familiar with the declaration mechanism should note the following differences between the ordinary Prolog mechanism and OTTER's.

- The predeclared operators are different. See Table 3.
- OTTER does not treat comma as an operator; in particular, `a,b,c` cannot be a term, as in `a,b,c -> d,e,f`.
- OTTER treats the quantifiers `all` and `exists` as special cases, because they don't seem to fit neatly into the standard Prolog mechanism.
- OTTER requires whitespace in some cases where the Prolog systems do not.

Symbols to be treated in this special way are given a type and a precedence. Either OTTER predeclares the symbol's properties, or the user gives OTTER a command of one of the following forms.

```
op(precedence, type, symbol).
op(precedence, type, list-of-symbols).
```

The `precedence` is an integer i, $0 < i < 1000$, and `type` is one of the following: `xfx`, `xfy`, `yfx` (infix), `fx`, `fy` (prefix), `xf`, `yf` (postfix). See Table 3 for the commands corresponding to the predeclared symbols.

Table 3: Predeclared Symbols

`op(800, xfy, # ).`	
`op(800, xfx, ->).`	`op(700, xfx, @<).`
`op(800, xfx, <->).`	`op(700, xfx, @>).`
`op(790, xfy, |).`	`op(700, xfx, @<=).`
`op(780, xfy, &).`	`op(700, xfx, @>=).`
`op(700, xfx, =).`	`op(500, xfy, +).`
`op(700, xfx, !=).`	`op(500, xfx, -).`
`op(700, xfx, <).`	`op(500, fx, +).`
`op(700, xfx, >).`	`op(500, fx, -).`
`op(700, xfx, <=).`	
`op(700, xfx, >=).`	`op(400, xfy, *).`
`op(700, xfx, ==).`	`op(400, xfx, /).`
`op(700, xfx, =/=).`	`op(300, xfx, mod).`

Given an expression that looks like it might be associated in a number of ways, the relative precedence of the operators determines, in part, how it is associated. A symbol with higher precedence is more dominant (closer to the root of the term), and one with lower precedence binds more tightly. For example, the symbols `->`, `|`, `&`, and `-` have decreasing precedence; therefore the expression `p & - q | r -> s` is understood as `((p & (-q)) | r) -> s`.

In each of the types, `f` represents the symbol, and `x` and `y`, which represent the expressions to which the symbol applies, specify how terms are associated. Given an expression involving symbols of the *same* precedence, the types of the symbol determines, in part, the association. See Table 4. The following are examples of associativity:

Table 4: Symbol Types

`xfx`	infix (binary)	don't associate
`xfy`	infix (binary)	associate right
`yfx`	infix (binary)	associate left
`fx`	prefix (unary)	don't associate
`fy`	prefix (unary)	associate
`xf`	postfix (unary)	don't associate
`yx`	postfix (unary)	associate

- If + has type `xfy`, then `a+b+c+d` is understood as `a+(b+(c+d))`.
- If -> has type `xfx`, then `a->b->c` is not well formed.
- If - has type `fy`, then `- - -p` is understood as `-(-(-(p)))`. (The spaces are necessary; otherwise, `---` will be parsed as single name.)
- If - has type `fx`, then `- - -p` is not well formed.

Caution: The associativity specifications in the infix symbol declarations say nothing about the logical associativity of the operation, for example, whether `(a+b)+c` is the same object as as `a+(b+c)`. The specifications are only about parsing ambiguous expressions. In most cases, when an operator is `xfy` or `yfx`, it is also logically associative, but the logical associativity is handled separately; it is built-in in the case of the logic symbols `|` and `&` in OTTER clauses and formulas, and it must be axiomatized in other cases.

Details of the Symbol Declarations. (This paragraph can be skipped by most users.) The precedence of symbols extends to the precedence of expressions in the following way. The precedence of an atomic, parenthesized, or standard application expression is 0. Respective examples are `p`, `(x+y)`, and `p(a+b,c,d)`. The precedence of a (well-formed) nonparenthesized nonatomic expression is the same as the precedence of the root symbol. For example, `a&b` has the precedence of `&`, and `a&b|c` has the precedence of the greater symbol. In the type specifications, `x` represents an expression of lower precedence than the symbol, and `y` represents an expression with precedence less than or equal to the symbol. Consider `a+b+c`, where + has type `xfy`; if association is to the left, then the second occurrence of + does *not* fit the type, because `a+b`, which corresponds to `x`, does not have a lower precedence than +; if association is to the right, then all is well. If we extend the example, under the declarations `op(700, xfx, =)` and `op(500, xfy, +)`, the expression `a+b+c=d+e` must be understood as `(a+ (b+c))= (d+e)`.

4.7 Whitespace in Expressions

The reason for separating ordinary names from special names (Sec. 4.2) is so that some whitespace (spaces, tabs, newline, and comments) can be removed. We can write `a+b+c` (instead of having to write `a + b + c`), because "`a+b+c`" cannot be a name, that is, it must be parsed into five names.

Caution. There is a deficiency in OTTER's parser having to do with whitespace between a name and opening parenthesis. The rule to use is: *Insert some white space if and only if it is* not *a standard application.* For example, the two pieces of white space in `(a+ (b+c))= (d+e)` are required, and no white space is allowed after `f` or `g` in `f(x,g(x))`.

4.8 Bugs and Other Anomalies in the Input and Output of Expressions

- The symbol | is either Prolog-style list punctuation or part of a special name. With the built-in declaration of | as infix, the term `[a|b]` is ambiguous, with possible interpretations t_1 =`$cons(a,b)` and t_2 =`$cons(|(a,b),$nil)`. OTTER recognizes it as the first. The term t_2 can be written `[(a|b)]`. The bug is that t_2 will be output without the parentheses. This is the only case we know in which OTTER cannot correctly read a term it has written.

- A term consisting of a unary + or − applied to a nonnegative integer is always translated to a constant.

- Parsing large terms without parentheses, say `a1+a2+a3+...+a1000`, can be very slow if the operator is left associative (`yfx`). If one intends to parse such terms, one should make the operator right associative (`xyf`).

- Quoted strings cannot contain a quotation mark of the same type.

- The flag `check_arity` sometimes issues warnings when it should not.

- Braces (`{}`) can be used to group input expressions, but OTTER always uses ordinary parentheses on output.

4.9 Examples of Operator Declarations

Group Theory. Suppose we like to see group theory expressions in the form $(ab^{-1}c^{-1-1})^{-1}$, in which right association is assumed. We can approximate this for OTTER with `(a*b^ *c^ ^)^`. (We have to make the group operator explicit; `-1` is not a legal OTTER name; the whitespace shown is required.) The declarations `op(400, xfy, *)` and `op(350, yf, ^)` suffice. Other examples of expressions (with minimum whitespace) using these declarations are `(x*y)*z=x*y*z` and `(y*x)^ =x^ *y^`.

OTTER **Options.** Options are normally input (Sec. 5.1) as in the following examples.

```
set(prolog_style_variables).
clear(print_kept).
assign(max_given, 300).
```

If, however, we make the declarations (the precedences are irrelevant in this case)

```
op(100, fx, set).
op(100, fx, clear).
op(100, xfx, assign).
```

then we may write

```
set prolog_style_variables.
clear print_kept.
max_given assign 300.
```

5 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 5. There are a few other commands for fringe features (Sec. 19).

Table 5: Commands

`include(`*file_name*`).`	`% read input from another file`
`op(`*precedence*`, `*type*`, `*name(s)*`).`	`% declare operator(s)`
`make_evaluable(`*sym*`, `*eval-sym*`).`	`% make a symbol evaluable`
`set(`*flag_name*`).`	`% set a flag`
`clear(`*flag_name*`).`	`% clear a flag`
`assign(`*parameter_name*`,`*integer*`).`	`% assign to a parameter`
`list(`*list_name*`).`	`% read a list of clauses`
`formula_list(`*list_name*`).`	`% read a list formulas`
`weight_list(`*weight_list_name*`).`	`% read weight templates`
`lex(`*symbol_list*`).`	`% assign an ordering on symbols`
`skolem(`*symbol_list*`).`	`% identify skolem functions`
`lrpo_multiset_status(`*symbol_list*`).`	`% status for LRPO`

5.1 Input of Options

OTTER recognizes two kinds of option: 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).            % enable 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 Sec. 6.

5.2 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(usable).
list(sos).
list(demodulators).
list(passive).
```

Example:

```
list(usable).
  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.
```

If the input contains more than one clause list of the same type, the lists will simply be concatenated.

5.3 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. (Note that demodulators cannot be input as formulas.)

```
formula_list(usable).
formula_list(sos).
formula_list(passive).
```

Example (analogous to above):

```
formula_list(usable).
  all a (a = a).                              % reflexivity
  all a (f(e,a) = a).                         % left identity
  all a (f(g(a),a) = e).                      % left inverse
```

```
  all a b c (f(f(a,b),c) = f(a,f(b,c))).  % associativity
  all a b c (f(c,a) = f(c,b) -> a = b).   % left cancellation
  all a b c (f(a,c) = f(b,c) -> a = b).   % right cancellation
end_of_list.
```

If the input contains more than one formula list of the same type, the lists will simply be concatenated.

5.4 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).      % to select given clauses
weight_list(purge_gen).       % to discard generated clauses
weight_list(pick_and_purge).  % to both pick and purge
weight_list(terms).           % to order terms
```

Example:

```
weight_list(pick_and_purge).
  weight(a, 0).                     % weight of constant a is 0
  weight(g($(2)), -50).             % twice weight of arg -50
  weight(P($(1),$(1)), 100).        % sum of weights of args +100
  weight(x, 5).                     % all variables have weight 5
  weight(f(g($(3)),$(4)), -300).    % see Sec. ``Weighting''
end_of_list.
```

See Sec. 10 for the syntax and use of weight templates.

5.5 The Commands `lex`, `skolem`, and `lrpo_multiset_status`

Each of the commands `lex`, `skolem`, and `lrpo_multiset_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(_,_), d, g(_), 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.

`lex([...]).` The `lex` command specifies an ordering (smallest-first) on function and constant symbols. Lexical ordering on terms is used in four contexts: orienting equality literals (Secs. 8.1.2 and 8.2.2), deciding whether an equality will be used as a demodulator (Secs. 8.1.3 and 8.2.3), deciding whether to apply a lex-dependent demodulator (Secs. 8.1.4 and 8.2.4), and evaluating functions/predicates that perform lexical comparisons (Sec. 9). If a `lex` command is not present, then OTTER uses a default ordering (Sec. 8).

`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` (Sec. 6.1.4) and `delete_identical_nested_skolem` (Sec. 6.1.5).

`lrpo_multiset_status([...]).` This command specifies multiset status for the lexicographic recursive path ordering (flag `lrpo`). See Sec. 8.2.

5.6 Other Commands

The command `op(`*precedence*`,` *type*`,` *name(s)*`)`, example `op(400,xfy,+)`, declares one or more symbols to have special properties with respect to input and output. See Sec. 4.6.

The command `make_evaluable(`*symbol*`,` *evaluable-symbol*`)`, for example `make_evaluable(_+_, $SUM(_,_)`, copies evaluation properties from an evaluable symbol to another symbol, so that one can write `x+3` instead of `$SUM(x,3)`. See Sec. 9.1.

The command `include(`*file_name*`)` causes input to be read from another input file. When the included file has been read, OTTER resumes reading commands after the `include` command. The file name must be recognized as an OTTER name, so if it contains characters such as period, slash, or hyphen, it must be enclosed in (single or double) quotes. Included files can include still other files. *A list of objects (clauses, formulas, or weight templates) cannot be split among different input files.* One can, however, read clauses into a list from more than one file, as in the following example.

standard input	file f1.in	file f2.in
`include("f1.in").`	`list(usable).`	`list(usable).`
`include("f2.in").`	`p(a).`	`p(b).`
	`end_of_list.`	`end_of_list.`

6 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 in the output file when such a change occurs.

Several additional flags and parameters are described in Sec. 19.

6.1 Flags

Flags are changed with the `set` and `clear` commands, for example,

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

6.1.1 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.

`sos_queue`. Default clear. If this flag is set, the first clause in `sos` is selected as the given clause (the set of support list operates as a queue). This causes a breadth-first search, also called level saturation. Some information about search levels is printed (see Sec. 16) if this flag is set.

`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). This causes a depth-first search (which rarely is useful with OTTER).

`input_sos_first`. Default clear. If this flag is set, the input clauses in `sos` are given a very low `pick_given` weight so that they are the first clauses selected as given clauses.

`interactive_given`. Default clear. If this flag is set, then when it's time to select a new given clause, the user is prompted for a choice. This flag has priority over all other flags that govern selection of the given clause.

`print_given`. Default set. If this flag is set, clauses are output when they become given clauses.

`print_lists_at_end`. Default clear. If this flag is set, then `usable`, `sos`, and `demodulators` are printed at the end of the search.

6.1.2 Inference Rules

`binary_res`. Default clear. If this flag is set, the inference rule binary resolution (along with any other inference rules that are set) is used to generate new clauses. Setting this flag causes the flags `factor` and `unit_deletion` to be automatically set.

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

`neg_hyper_res`. Default clear. If this flag is set, the inference rule negative hyperresolution (along with any other inference rules that are set) is used to generate new clauses.

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

`para_into`. Default clear. If this flag is set, the inference rule "paramodulation *into* the given clause" (along with any other inference rules that are set) is used to generate new clauses. When using paramodulation, one should include the appropriate clause for reflexivity of equality, for example, `x=x`.

`para_from`. Default clear. If this flag is set, the inference rule "paramodulation *from* the given clause" (along with any other inference rules that are set) is used to generate new clauses. When using paramodulation, one should include the appropriate clause for reflexivity of equality, for example, `x=x`.

`demod_inf`. Default clear. If this flag is set, demodulation is applied, as if it were an inference rule, to the given clause. This is useful when term rewriting is the main objective. When this flag is set, the given clause is copied, then processed just like any newly generated clause.

6.1.3 Resolution Restriction Flags

`order_hyper`. Default set. If this flag is set, then the inference rules `hyper_res` and `neg_hyper_res` are constrained by an ordering strategy. A literal in a satellite is allowed to resolve only if it is maximal in the satellite. (A literal is maximal in a clause if and only if there is no larger literal.) The ordering uses only the lexical value (as in the `lex` command or the default, Sec. 5.5) of the predicate symbol. (This flag is irrelevant for positive hyperresolution with a Horn set.)

`unit_res`. Default clear. This flag is a restriction on binary resolution. If it is set, then all binary resolution inferences must be unit resolutions; that is, one of the parents must be a unit clause. Setting this flag causes to the flag `binary_res` to be set as well.

`ur_last`. Default clear. This flag is a restriction on unit-resulting resolution. If it is set, then the UR-resolvent must come from the last literal of the nonunit parent (the nucleus). This is related to the *target strategy* in linked UR-resolution.

6.1.4 Paramodulation Restriction Flags

`para_from_left`. Default set. If this flag is set, paramodulation is allowed *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, paramodulation is allowed *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, paramodulation is allowed *into* left sides 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, paramodulation is allowed *into* right sides of positive and negative equalities. (Applies to both `para_into` and `para_from` inference rules.)

`para_from_vars`. Default clear. If this flag is set, paramodulation *from* variables is allowed. *Warning: Setting this option may produce too many paramodulants.* (Applies to both `para_into` and `para_from` inference rules.)

`para_into_vars`. Default clear. If this flag is set, paramodulation *into* variables is allowed. *Warning: Setting this option may produce too many paramodulants.* (Applies to both `para_into` and `para_from` inference rules.)

`para_from_units_only`. Default clear. If this flag is set, paramodulation is allowed 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, paramodulation is allowed 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, paramodulation is never allowed *into* subterms of Skolem expressions [16]. (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, all occurrences of the *into* term are replaced with the replacement term. (Applies to both `para_into` and `para_from` inference rules.)

6.1.5 Flags for Handling Generated Clauses

(Section 6.1.6 describes equality-related flags for handling generated clauses.)

`detailed_history`. Default set. This flag affects the parent lists in clauses that are derived by `binary_res`, `para_from`, or `para_into`. If the flag is set, the positions of the unified literals or terms are given along with the IDs of the parents. See Sec. 16 for examples.

`order_history`. Default clear. This flag affects the order of parent lists in clauses that are derived by hyperresolution, negative hyperresolution, or UR-resolution. If the flag is set, then the nucleus is listed first, and the satellites are listed in the order in which the corresponding literals appear in the nucleus. If the flag is clear (or if the clause was derived by some other inference rule), the given clause is listed first.

`unit_deletion`. Default clear. If this flag is set, unit deletion is applied 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 `usable` 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. One can view unit deletion with unit clause `P` as demodulation applied to literals with the demodulator `P = $T`. (Unit deletion is not useful if all generated clauses are units.)

back_unit_deletion. Default clear. If this flag is set, then whenever a unit clause is derived and kept, it is used to apply unit deletion to all existing clauses in usable or sos.

delete_identical_nested_skolem. Default clear. If this flag is set, clauses with the nested Skolem property are deleted. A clause has the nested Skolem property if it contains a a Skolem expression that (properly) contains an occurrence of its leading Skolem symbol. For example, if f is a Skolem function, a clause containing a term f(f(x)) or a term f(g(f(x))) is deleted.

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, forward subsumption is applied during the processing of newly generated clauses. (New clauses are deleted if subsumed by any clause in usable or sos.)

back_sub. Default set. If this flag is set, back subsumption is applied during the processing of newly kept clauses. (Clauses in usable or sos are deleted if subsumed by the newly kept clause.)

factor. Default clear. If this flag is set, factoring is applied in two ways. First, factoring is applied as a simplification rule to newly generated clauses. If a generated clause C has factors that subsume C, it is replaced with its smallest subsuming factor. Second, it is applied as an inference rule to 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 factor is set, a clause with n literals will *not* cause a clause with fewer than n literals to be deleted by subsumption.

6.1.6 Demodulation and Ordering Flags

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

order_eq. Default clear. If this flag is set, equalities are flipped if the right side is heavier than the left. See Secs. 8.1.2 and 8.2.2 for the meaning of "heavier".

eq_units_both_ways. Default clear. If this flag is set, unit equality clauses (both positive and negative) are sometimes stored in both orientations; the action taken depends on the flag order_eq. If order_eq is clear, then whenever a unit, say $\alpha = \beta$, is processed, $\beta = \alpha$ is automatically generated and processed. If order_eq is set, then the reversed equality is generated only if the equality cannot be oriented (see Secs. 8.1.2 and 8.2.2).

demod_linear. Default clear. If this flag is set, demodulation indexing is disabled, and a linear search of demodulators are used when rewriting terms. With indexing disabled, if more than one demodulator can be applied to rewrite a term, then the one whose clause number is lowest 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, terms are demodulated 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 leftmost outermost 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 evaluable conditional term `$IF(`*condition*`,`*then-value*`,`*else-value*`)` is an exception when inside-out demodulation is in effect. See Sec. 9.)

`dynamic_demod`. Default clear. If this flag is set, *some* newly kept equalities are made into demodulators (Secs. 8.1.3 and 8.2.3). Setting this flag automatically sets the flag `order_eq`.

`dynamic_demod_all`. Default clear. If this flag is set, OTTER attempts to make *all* newly kept equalities into demodulators (Sec. 8.1.3). Setting this flag automatically sets the flags `dynamic_demod` and `order_eq`.

`dynamic_demod_lex_dep`. Default clear. If this flag is set, dynamic demodulators may be lex-dependent or LRPO-dependent. See Secs. 8.1.3 and 8.2.3.

`back_demod`. Default clear. If this flag is set, back demodulation is applied to `demodulators`, `usable`, 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`.

`anl_eq`. Default clear. If this flag is set, a standard equational strategy will be applied to the search. This flag is really a metaflag; its only effect is to alter other flags 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(para_from_vars)`, `set(eq_units_both_ways)`, `set(dynamic_demod_all)`, `set(back_demod)`, `set(process_input)`, and `set(lrpo)`. This strategy is derived mostly from equational strategies developed at Argonne by Larry Wos and Ross Overbeek. It can also be used for Knuth-Bendix completion. See Sec. 8.3 for more details.

`knuth_bendix`. Default clear. Setting this flag simply causes the preceding flag, `anl_eq`, to be set.

`lrpo`. Default clear. If this flag is set, then the lexicographic recursive path ordering (also called RPO with status) is used to compare terms. If this flag is clear, weight templates and lexicographic order are used (Secs. 8.2 and 8.3).

`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* `lrpo` *is set,* `lex_order_vars` *has no effect on demodulation* (Sec. 8.1.1).

`symbol_elim`. Default clear. If this flag is set, then new demodulators are 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 if 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.

`rewriter`. Default clear. If this flag is set, then the clauses in the `sos` list will simply be demodulated by the demodulators, the run will terminate. This is really just a metaflag, which automatically causes the several other options parameters to be changed as follows: `set(demod_inf)`, `clear(for_sub)`, `clear(back_sub)`, and `assign(max_levels, 1)`.

6.1.7 Input Flags

`check_arity`. Default set. If this flag is set, a warning is given if symbols have variable arities (different numbers of arguments in different places in the input). For example, the term `f(a,a(b))` would be flagged. (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 with _.

`echo_included_files`. Default set. If this flag is set, input files included with the `include(`*filename*`)` command are echoed in the same way as ordinary input.

`simplify_fol`. Default set. If this flag is set, then some propositional simplification is attempted when converting input first-order formulas into clauses. The simplification occurs after Skolemization, during the CNF translation. If simplification detects a refutation, it will always produce the empty clause `$F`, but OTTER will not recognize the proof (i.e., give the proof message and stop) unless the flag `process_input` is set.

`process_input`. Default clear. If this flag is set, input `usable` and `sos` clauses (including clauses from formula input) are processed as if they had been generated by an inference rule. (See the procedure for processing newly inferred clauses in Sec. 2.) The exceptions are (1) the following clause-processing options are not applied to input clauses: `max_literals`, `max_weight`, `delete_identical_nested_skolem`, and `max_distinct_vars`, (2) clauses input on list `usable` remain there if retained, and (3) some output appears even if the output flags (Sec. 6.1.8) are clear.

`tptp_eq`. Default clear. If this flag is set, then "`EQUAL`" is the one and only symbol recognized as the equality relation for the operations that build in equality (demodulation and paramodulation).

6.1.8 Output Flags

`very_verbose`. Default clear. If this flag is set, a tremendous amount of information about the processing of generated clauses is output.

`print_kept`. Default set. If this flag is set, new clauses are output if they are retained (if they pass all retention tests).

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

`build_proof_object_1`. Default clear. If this flag is set, then whenever a proof is found, a *type 1 proof object* is printed to the output file. Proof objects are very detailed proof and were introduced for two purposes: so that proofs can be checked by an independent program, and so that proofs can be translated into other forms by other programs. Proof objects are written in a Lisp-like notation. (Type 2 proof objects are usually preferred.) *Warning: Construction of proof objects is fragile—sometimes it simply fails.*

`build_proof_object_2`. Default clear. If this flag is set, then whenever a proof is found, a *type 2 proof object* is printed to the output file. Type 2 proof objects are used in the IVY verification project [19], and a detailed description (definition in ACL2) can be found there. *Warning: construction of proof objects is fragile—sometimes it simply fails.*

`print_new_demod`. Default set. If this flag is set, demodulators that are adjoined during the search (`dynamic_demod`) are printed. New demodulators are always printed during input processing.

`print_back_demod`. Default set. If this flag is set, clauses are printed as they are back demodulated. Back-demodulated clauses are always printed during input processing.

`print_back_sub`. Default set. If this flag is set, clauses are printed if they are back subsumed. Back-subsumed clauses are always printed during input processing.

`display_terms`. Default clear. If this flag is set, all clauses and terms are printed in pure prefix form (Sec. 4.3). This feature can be useful for debugging the input.

`pretty_print`. Default clear. If this flag is set, clauses are output in an indented form that is sometimes easier to read. The parameter `pretty_print_indent` (default 4) specifies the number of spaces for each indent level.

`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)`. Terms cannot be input in combinatory logic notation.

`formula_history`. Default clear. If this flag is set, and if quantified formulas are given as input, then the formulas will occur in proofs, and the clauses derived from the formulas will refer to the formulas with the justification `clausify`.

6.1.9 Indexing Flags

index_for_back_demod. Default set. If this flag is set, all nonvariable terms in all clauses are indexed so that the appropriate ones can be quickly retrieved when applying a dynamic demodulator to the clause space (back demodulation). This type of indexing can use a lot of memory. If the flag is clear, back demodulation still works, but it is much slower.

for_sub_fpa. Default clear. If this flag is set, FPA indexing is used for forward subsumption. If this flag is clear, discrimination tree indexing is used. Setting this flag can decrease the amount of memory required by OTTER. Discrimination tree indexing can require a lot of memory, but it is usually *much* faster than FPA indexing.

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

6.1.10 Miscellaneous Flags

control_memory. Default clear. If this flag is set, then the automatic memory-control feature is enabled (Sec. 17).

propositional. Default clear. If this flag is set, OTTER assumes that all clauses are propositional, and it makes some optimizations. *The user should set this flag only when all clauses are propositional; otherwise* OTTER *may make unsound inferences and/or crash.*

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).

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 plus the maximum of the weights of the arguments. If this flag is clear, the default weight of an atom is 1 plus 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 plus the maximum of the weights of the arguments. If this flag is clear, the default weight of a term is 1 plus the sum of the weights of the arguments.

`free_all_mem`. Default clear. If this flag is set, then at the end of the search, most dynamically allocated memory is returned to the memory managers. This flag is used mainly for debugging, in particular, to help find memory leaks. Setting this flag will *not* cause OTTER to use less memory.

`sigint_interact`. Default set. If this flag is set, then when OTTER receives an interrupt signal from the operating system (usually caused by the user pressing control-C), OTTER will enter a primitive interactive mode, which is described in Sec. 15.

6.2 Parameters

Parameters are integer-valued options. In the descriptions that follow, ∞ is a large integer, usually the size of the largest ordinary integer on the user's computer (i.e., INT_MAX in ANSI C).

6.2.1 Monitoring Progress

`assign(report,`n`)`. Default -1, range $[-1..\infty]$. If $n > 0$, then statistics are output approximately every n CPU seconds. The time is not exact because statistics will be output only after the current given clause is finished. This feature can be used in conjunction with UNIX programs such as `grep` and `awk` to conveniently monitor OTTER jobs.

6.2.2 Placing Limits on the Search

`assign(max_seconds,`n`)`. Default -1, range $[-1..\infty]$. If $n \neq -1$, the search is terminated after about n CPU seconds. The time is not exact because OTTER will wait until the current given clause is finished before stopping.

`assign(max_gen,`n`)`. Default -1, range $[-1..\infty]$. If $n \neq -1$, the search is terminated 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.

`assign(max_kept,`n`)`. Default -1, range $[-1..\infty]$. If $n \neq -1$, the search is terminated 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.

`assign(max_given,`n`)`. Default -1, range $[-1..\infty]$. If $n \neq -1$, the search is terminated after n given clauses have been used.

`assign(max_levels,`n`)`. Default -1, range $[-1..\infty]$. If $n \neq -1$, the flag `sos_queue` will be automatically set, causing a level saturation (breadth-first) search. In this case the search is terminated after n levels have been processed.

`assign(max_mem,`n`)`. Default -1, range $[-1..\infty]$. If $n \neq -1$, OTTER will terminate the search before more than n kilobytes have been dynamically allocated (`malloc`).

6.2.3 Limits on Properties of Generated Clauses

`assign(max_literals,`n`)`. Default -1, range $[-1..\infty]$. If $n \neq -1$, new clauses are discarded if they contain more than n literals.

`assign(max_weight,`n`)`. Default ∞, range $[-\infty..\infty]$. 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 may not be present; see Sec. 10).

`assign(max_distinct_vars,`n`)`. Default -1, range $[-1..\infty]$. If $n \neq -1$, new clauses are discarded if they contain more than n distinct variables.

`assign(max_answers,`n`)`. Default -1, range $[-1..\infty]$. If $n \neq -1$, new clauses are discarded if they contain more than n answer literals.

6.2.4 Indexing Parameters

`assign(fpa_literals,`n`)`. Default 8, range [0..100]. 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 leading symbols of the arguments of the literal, and so on. Greater indexing depth requires more memory, but it can be faster. Changing this parameter will not change the clauses that are generated or kept.

`assign(fpa_terms,`n`)`. Default 8, range [0..100]. 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 symbol only; if $n = 1$, indexing looks at the symbol and the leading symbols of the arguments of the term; and so on. Greater indexing depth requires more memory, but it can be faster. Changing this parameter will not change the clauses that are generated or kept.

6.2.5 Miscellaneous Parameters

`assign(pick_given_ratio,`n`)`. Default -1, range $[-1..\infty]$. This parameter causes some given clauses to be selected by weight and others in a breadth-first manner (by age). If $n \neq -1$, n given clauses are are selected by (smallest `pick_given`) weight, then the first clause in `sos` is selected as given clause, then n given clauses are selected by weight, and so forth. This method allows heavy clauses to enter into the search while focusing mainly on light clauses. It combines breadth-first search and best-first search (default selection by weight). If n is -1, then the clause with smallest `pick_given` weight is always selected.

`assign(age_factor,`n`)`. Default 0, range $[-\infty..\infty]$. If $n \neq 0$, then the pick-given weight of clauses is adjusted as follows. If g is the number of clauses that have been given *at the time the clause is kept*, and n is the age factor, then g/n (with integer division) is added to the pick-given weight of the clause.

`assign(distinct_vars_factor,`n`)`. Default 0, range $[-\infty..\infty]$. If $n \neq 0$, then the pick-given weight of clauses is adjusted as follows. If v is the number of variable in the clause, and n is the age factor, then v/n (with integer division) is added to the pick-given weight of the clause.

`assign(interrupt_given,`n`).` Default -1, range $[-1..\infty]$. If $n > 0$, then after n given clauses have been used, OTTER goes into its interactive mode (Sec. 15).

`assign(demod_limit,`n`).` Default 1000, range $[-1..\infty]$. If $n \neq -1$, n is the maximum number of rewrites that will be applied when demodulating a clause. The count includes `$` symbol evaluation. If n is -1, there is no limit. A warning message is printed if OTTER attempts to exceed the limit.

`assign(max_proofs,`n`).` Default 1, range $[-1..\infty]$. 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 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 $n = -1$, OTTER will find as many proofs as it can (within other constraints).

`assign(min_bit_width,`n`).` Default *bits-per-long*, range [0..*bits-per-long*]. When the evaluable bit operations (Sec. 9) produce a new bit string, leading zeros are suppressed under the constraint that n is the minimum string length. (The value *bits-per-long* is the number of bits in the C data type long integer.)

`assign(neg_weight,`n`).` Default 0, range $[-\infty..\infty]$. The value n is the additional weight (positive or negative) that is given to negated literals. Weight templates cannot be used for this purpose because the negation sign on a literal cannot occur in weight templates. (Atoms, not literals, are weighed with weight templates; see Sec. 10.)

`assign(pretty_print_indent,`n`).` Default 4, range [0..16]. See flag `pretty_print`, Sec. 6.1.8.

`assign(stats_level,`n`).` Default 2, range [0..4]. This indicates 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 search and time statistics are output; if $n = 2$, all search and time statistics are output; if $n = 3$, search, time, and memory statistics are output; and if $n = 4$, search, time, and memory statistics and option values are output. This parameter does not affect the speed of OTTER, because all statistics are always kept.

`assign(dynamic_demod_depth,`n`).` Default -1, range $[-1..\infty]$.
`assign(dynamic_demod_rhs,`n`).` Default 1, range $[-\infty..\infty]$.
These two parameters work together, allowing an extension of the ad hoc ordering when deciding whether a new equality should be a demodulator. (It is not used if flag `lrpo` is set.) The equality, say $\alpha = \beta$, is first oriented as described in Sec. 8.1. If $wt(\beta) \leq$ `dynamic_demod_rhs` and if $wt(\alpha) - wt(\beta) \geq$ `dynamic_demod_depth`, then the equality can be a demodulator. With the default values for these parameters, the behavior is as described in Sec. 8.1

`assign(new_symbol_lex_position,`n`).` Default ∞, range $[1..\infty]$. New symbols can be created during the search, usually by $-evaluation. With this parameter, the user can specify where they will occur in the symbol ordering. If there is a `lex` command, all new symbols will have a lexical values between the nth and $(n+1)$th symbol in the `lex` command. The ordering among the new symbols is the default ordering. This also applies to input symbols not occurring in the `lex` command.

7 Demodulation

Basic demodulation is straightforward, but there are many variations and enhancements whose descriptions are scattered throughout this manual. This section (which is mostly redundant) lists some overall comments on demodulation and points the reader to the appropriate sections on variations and enhancements.

The Equality Symbol. The binary symbol = (which can be used as an infix symbol) and any name that starts with `eq`, `EQ`, or `Eq`, when used as a binary predicate symbol, is recognized as an equality predicate by demodulation. *An exception:* if the flag `tptp_eq` is set, then `EQUAL` is the one and only equality symbol; this flag was introduced for compatibility with the TPTP problem library [25].

When and How It Is Applied. Demodulation is applied, using equalities in the list `demodulators`, to every clause that is generated by an inference rule. Also, when the flag `demod_inf` (Sec. 6.1.2) is set, demodulation is, in effect, treated as an inference rule.

Demodulation of Atomic Formulas. Atomic formulas (literals with any negation sign removed) can be demodulated. Useful examples are

```
(x*y = x*z) = (y = z).  % one form of cancellation
D(x,y) = D(y,x).        % lex-dependent atom demodulator
P(junk) = $T.           % trick to get rid of a literal
```

The appropriate clause simplification occurs if the right side of an atom demodulator is one of the Boolean constants `$T` or `$F`. Negated literals cannot be demodulated, but the atom of a negative literal can be demodulated.

Inside-out or Outside-in. The user has the option of having terms rewritten inside-out or outside-in. (See the description of the flag `demod_out_in` in Sec. 6.1.6.) Although the choice makes little difference for many applications, we nearly always recommend inside-out. Outside-in can be much faster in cases where the left side of the demodulator has a variable not in the right side.

Order of Demodulators. By default, demodulation uses an indexing mechanism to find demodulators that can rewrite a given term; if more than one demodulator can apply, the user has no control over which one is used. To order the set of demodulators for application, the user can set the flag `demod_linear` (Sec. 6.1.6).

Dynamic Demodulation and Back Demodulation. Positive equality units derived during the search can be made into demodulators (Secs. 6.1.6, 8.1.3, and 8.2.3). Demodulators adjoined during the search can be used to rewrite previously derived clauses (Sec. 6.1.6).

Termination. With the default ad hoc ordering, demodulation is not guaranteed to terminate by itself. Therefore, a parameter (`demod_limit`) specifies the maximum number of rewrite steps that will be applied to a clause. With the lexicographic recursive path ordering (flag `lrpo`), demodulation will always terminate by itself. (Even with `lrpo`, the parameter `demod_limit` has effect because demodulation sequences can have an unreasonable number of steps.)

Introduction of New Variables. A demodulator introduces new variables if it has variables on the right side that do not occur on the left. The LRPO flag does not allow demodulators to introduce new variables. The default ordering allows variable introductions only for input demodulators.

Lex- and LRPO-dependent Demodulation. Ordinary demodulators are used unconditionally; they usually simplify or canonicalize regardless of the context in which they are applied. But some equalities that are not normally thought of as rewrite rules can be used as such and are applied only if the application produces a "better" term. These are called lex- or LRPO-dependent demodulators (depending on whether the flag `lrpo` is set). For example, commutativity of an operation, say $x + y = y + x$, can be used to rewrite $b + a$ to $a + b$ if $a + b \prec b + a$. See Secs. 6.1.6, 8.1.4, and 8.2.4. Do not confuse this type of demodulation with conditional demodulation.

Demodulation of Evaluable Terms. OTTER has many built-in function and predicate symbols for doing arithmetic, logic operations, bit operations, and other operations. The evaluation of terms containing these built-in symbols is done as a part of demodulation (Sec. 9).

Conditional Demodulation. Demodulators can be written with conditions as

condition -> $\alpha = \beta$.

The demodulator is applied only if the condition, instantiated with the matching substitution, demodulates to `$T` (meaning *true*). This is a "fringe feature", and it has not been heavily used (Sec. 19.4).

Demodulation as Equational Programming. OTTER's demodulation, especially with the evaluable symbols, can be used as a general-purpose (although not particularly efficient or convenient) equational programming system (Sec. 9). We have not seen many cases where this is useful in the context of a traditional refutation search, but it has proved to be very useful for various symbolic programming tasks, particularly with hyperresolution.

Demodulation to Delete Clauses. Demodulation can be used as a trick to overcome one of the deficiencies of the weighting mechanism (Sec. 10) to discard undesired clauses. Weighting does not implement a true match (one-way unification) operation. If the user wishes to discard every clause that contains an instance of a particular term, say `f(x,x)`, a demodulator, say `f(x,x) = junk`, can be input along with a weight template that gives `junk` a `purge_gen` weight higher than `max_weight`. (When using this and similar tricks, the user must make sure that the clauses containing `junk` are really discarded by weighting or another means; on occasion we have found proofs that are incorrect because they depend on `junk`.)

8 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`, `lrpo`, and `dynamic_demod_lex_dep`. It gives all the rules—built in and optional—for orienting equality literals and deciding which equalities will be dynamic demodulators. OTTER uses two kinds of term ordering.

ad hoc ordering. This is a collection of ordering methods that we have accumulated through many years of experimentation. The methods do not have a substantial theoretical foundation, but they are useful in many cases. This is the default ordering; it is presented in Sec. 8.1.

LRPO**.** This is the *lexicographic recursive path ordering* (also called RPO with status). It has nice theoretical properties and is easier to use than the ad hoc ordering, but it is more computationally expensive. The LRPO ordering is enabled with the flag `lrpo`; it is described in Sec. 8.2.

Both kinds of term ordering use an ordering on constant and function symbols. The `lex` command (Sec. 5.5) is used to assign an ordering on symbols. For example, the command

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

specifies $\mathtt{a} \prec \mathtt{b} \prec \mathtt{c} \prec \mathtt{d} \prec \mathtt{or}$ (`or` is a binary symbol). If a `lex` command is given, all constant and function symbols in terms that will be compared must be included. If a `lex` command is not given, OTTER uses the following default ordering.

[*constants*, *high-arity*, $\cdots$, *binary*, *unary*]

Within arity, the lexicographic ASCII ordering (i.e., the C library routine `strcomp()`) is used.

The methods for orienting equalities and for determining dynamic and lex-dependent demodulators apply to all inferred clauses; if the flag `process_input` is set, they also apply to input `usable` and `sos` clauses.

In this section, α and β always refer to the left and right arguments, respectively, of the equality literal under consideration; $wt(\gamma)$ refers to the weight of γ using `weight_list_terms`; $vars(\gamma)$ is the set of variables in γ. The symbols $\succ$ and $\prec$ are used for several orderings; the one referred to should be clear from the context.

Table 6 is a quick reference guide to the ordering mechanisms presented in Secs. 8.1 and 8.2.

Table 6: Quick Reference to Ordering

Situation		Ad Hoc	LRPO
Input demods	flip?	no	if $\alpha \prec \beta$
	lex-dependent?	if ident-x-vars	if neither is greater
Orienting eqs (`order_eq` set)		flip if sym-elim, occurs-in, or wt-lex-ord	flip if $\alpha \prec \beta$
Dynamic demod?	`d_d_all` clear	if oriented, var-subset, and $wt(\beta) \leq 1$	if $\alpha \succ \beta$
	`d_d_all` set	if oriented and var-subset	if $\alpha \succ \beta$
	lex-dependent?	if ident-x-vars and `dynamic_demod_all` set	if neither is greater, and var-subset
Apply lex-dependent demod?		lex-order($\alpha\sigma, \beta\sigma$)	$\alpha\sigma \succ \beta\sigma$
Lex `$` evaluation		lex-order	lex-order

8.1 Ad Hoc Ordering

8.1.1 Term Ordering (Ad Hoc)

Two types of ad hoc term ordering are used: lex-order and weight-lex-order. The user does not have a choice between these two; the one that is applied depends on the context, as described in the following subsections.

lex-order. This is a basic lexicographic extension of the symbol order. To compare two terms, one reads them left to right, stopping at the first symbols where they differ; the relationship of those symbols determines the term order. The treatment of variables depends on the flag `lex_order_vars`:

`lex_order_vars` **is set.** Variables are the lowest in the symbol ordering, with $\mathtt{x} \prec \mathtt{y} \prec \mathtt{z} \prec \mathtt{u} \prec \mathtt{v} \prec \mathtt{w} \prec \mathtt{v6} \prec \mathtt{v7} \prec \mathtt{v8} \prec \cdots$. 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 and to a term that contains the variable. The order on terms is partial. Note that if $t_1 \prec t_2$, and if σ is any substitution, then $t_1\sigma \prec t_2\sigma$.

weight-lex-order. In comparing two terms, they are first weighed with `weight_list_terms`. If one term is heavier, it is greater in the order. If the terms have equal weight, they are compared with respect to the lex-order as if `lex_order_vars` is clear.

8.1.2 Orienting Equalities (Ad Hoc)

If the flag `order_eq` is set and `lrpo` is clear, then equality literals (both positive and negative) in inferred clauses are processed as follows.

1. If the `symbol_elim` flag is set and if the equality is a symbol-eliminating type (Sec. 6.1.6), the equality is oriented in the appropriate direction.
2. If one argument is a proper subterm of the other argument, the equality is oriented so that the subterm is the right-hand argument.
3. If one argument is greater in the weight-lex-order, say $\gamma \succ \delta$, the equality is oriented with γ as the left side.

The preceding steps do not apply to equalities input on the list `demodulators`.

8.1.3 Determining Dynamic Demodulators (Ad Hoc)

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, the flag `order_eq` will also be set, and OTTER will attempt to make some or all inferred positive equality units into demodulators. If the flag `process_input` is set, the procedure applies to input `usable` and `sos` equalities. The procedure assumes that equalities have already been oriented.

1. If the flag `symbol_elim` is set and if $\alpha = \beta$ is symbol-eliminating, the equality becomes a demodulator.
2. If β is a proper subterm of α, the equality becomes a demodulator.
3. If $\alpha \succ \beta$ in the weight-lex-order, and if $vars(\alpha) \supseteq vars(\beta)$,
   (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` and `dynamic_demod_all` are both set, if α and β are identical-except-variables (Sec. 8.1.4), and if $vars(\alpha) \supseteq vars(\beta)$, the equality becomes a lex-dependent demodulator.

8.1.4 Lex-dependent Demodulation (Ad Hoc)

Two terms are *identical-except-variables* if they are identical after replacing all occurrences of variables with `x`. An input or dynamic demodulator is lex-dependent only if α and β are identical-except-variables. (See Sec. 8.1.3 for determining lex-dependent dynamic demodulators.) A lex-dependent demodulator applies to a term only if the replacement term is smaller in the lex-order. In particular, OTTER will apply a lex-dependent demodulator $\alpha = \beta$ if and only if $\alpha\sigma \succ \beta\sigma$ in the lex-order, where σ is the matching substitution.

For example, in the presence of the `lex` command and the (lex-dependent) demodulators

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

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).

8.2 LRPO

8.2.1 Term Ordering (LRPO)

The *lexicographic recursive path ordering* (LRPO, or RPO with status) [4, 7, 9] is a method for comparing terms. The important theoretical property of LRPO is that it is a *termination ordering*. That is, let R be a set of demodulators in which in each demodulator, the left side is LRPO-greater than the right side; then demodulation (applying the demodulators left to right) is guaranteed to terminate.

To use LRPO one typically uses the `lex` command (Sec. 5.5) to assign an ordering on constant and function symbols. If the `lex` command is not present, OTTER assigns an ordering (which is frequently ineffective). (OTTER uses a total ordering on symbols that is fixed at input time. Other implementations of LRPO use partial orderings or dynamically changing orderings.)

With respect to LRPO, function symbols can have either *left-to-right status* (the default) or *multiset status*. The command `lrpo_multiset_status(`*symbol_list*`)` gives symbols multiset status.

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.).

8.2.2 Orienting Equalities (LRPO)

If the flag `order_eq` is set and if one argument of the equality literal (positive or negative) is greater in the LRPO order, the greater argument is placed on the left side. This rule applies to input demodulators, to inferred clauses, and, if the flag `process_input` is set, to input `usable` and `sos` clauses.

8.2.3 Determining Dynamic Demodulators (LRPO)

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

8.2.4 LRPO-dependent Demodulation (LRPO)

An LRPO-dependent demodulator is allowed to rewrite a term if and only if its application produces an LRPO-less-than term.

8.3 Knuth-Bendix Completion

The Knuth-Bendix completion procedure [12] attempts to transform a set E of equalities into a terminating, canonical set of rewrite rules (demodulators). If it is successful, the resulting set of rewrite rules, a *complete set of reductions*, is a decision procedure for equality of terms in the theory E. There are many variations and refinements of the Knuth-Bendix procedure.

Setting either of the flags `anl_eq` or `knuth_bendix` causes OTTER to automatically alter a set of options so that its search will behave like a Knuth-Bendix completion procedure. If OTTER's search stops because its `sos` list is empty, and if certain other conditions are met, then the resulting set of equalities should be a complete set of reductions. (OTTER was not designed to implement a completion procedure, and it has not been optimized for completion.)

Conjecture. If (1) the set E of equalities, along with `x=x`, is input in list `sos`, (2) flag `anl_eq` is set, (3) other options that are changed from the defaults do not affect the search, (4) OTTER stops with "sos empty", and (5) other than `x=x`, the final `usable` list is the same as the final `demodulators` list, then the `demodulators` list is a complete set of reductions for E.

Here is an input file that causes OTTER to search for and quickly find a complete set of reductions for free groups. Note that the predeclared (right associative) infix operator `*` is used.

```
set(anl_eq).
set(print_lists_at_end).
lex([e, _*_, g(_)]).

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

```
end_of_list.
```

The critical issue in most applications of the Knuth-Bendix completion procedure is the choice of ordering scheme and/or the specific ordering on symbols. Note, in this case, that if the `lex` command is absent, the default symbol ordering suffices because it is essentially the same as the one specified.

The `anl_eq` flag is also very useful when trying to prove equational theorems. When using `anl_eq` to search for proofs, we are not bound by the conditions listed in the above claim; in fact, we usually apply additional strategies such as limiting the size of retained equalities, being more selective about making equalities into demodulators, and disabling LRPO ordering.

With the following input file, OTTER uses the `anl_eq` option to prove the difficult half of a group theory theorem of Levi: *The commutator operation is associative if and only if the commutator of any two elements lies in the center of the group.* (A textbook proof can be found in [13].) Note that, contrary to common practice, the symbol order does not cause the definition of the commutator operation `h(_,_)` to be used as a rewrite rule to eliminate commutator expressions in `h`. Note also that weight templates are used to eliminate clauses containing terms with particular structures; this decision is purely heuristic, derived from experimentation and intuition. OTTER finds a proof in about a minute and uses about 6 megabytes of memory.

```
set(anl_eq).
lex([a,b,c,e,h(_,_),f(_,_),g(_)]).
assign(max_weight, 20). assign(pick_given_ratio, 5).
clear(print_kept).
clear(print_new_demod). clear(print_back_demod).

list(usable).
x = x.
f(e,x) = x.                      % group theory
f(g(x),x) = e.
f(f(x,y),z) = f(x,f(y,z)).
end_of_list.

list(sos).
f(g(x),f(g(y),f(x,y))) = h(x,y). % definition of commutator
h(h(x,y),z) = h(x,h(y,z)).       % commutator is associative
% Denial: there are two elements whose commutator
% is not in the center.
f(h(a,b),c) != f(c,h(a,b)).
end_of_list.

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

```
end_of_list.
```

9 Evaluable Functions and Predicates (`$SUM`, `$LT`, ...)

OTTER can be used in a "programmed" mode that is quite different from normal refutational theorem proving. When using the programmed mode, one generally has in mind a particular method for solving a problem; and when writing clauses for the programmed mode, one generally knows exactly how they will be used by OTTER.

The programmed mode frequently involves a set of evaluable function and predicate symbols known as the `$`-symbols (because each starts with `$`). Examples are `$SUM` and `$LT` for integer arithmetic and `$AND` for Boolean operations.

The evaluable symbols operate on five types of OTTER term: integer constants, floating-point constants, bit-string constants, the Boolean constants `$T` and `$F`, and arbitrary terms. The symbols that evaluate to type Boolean can occur either as function symbols or as predicate symbols. The integer, bit, and floating-point operations behave the same as the underlying C operations applied to the data types "long int", "unsigned long int", and "double", respectively. Table 7 lists the evaluable functions and predicates by type.

Table 7: Evaluable Functions and Predicates

$int \times int \to int$	`$SUM`, `$PROD`, `$DIFF`, `$DIV`, `$MOD`
$int \times int \to bool$	`$EQ`, `$NE`, `$LT`, `$LE`, `$GT`, `$GE`
$float \times float \to float$	`$FSUM`, `$FPROD`, `$FDIFF`, `$FDIV`
$float \times float \to bool$	`$FEQ`, `$FNE`, `$FLT`, `$FLE`, `$FGT`, `$FGE`
$bits \times bits \to bits$	`$BIT_AND`, `$BIT_OR`, `$BIT_XOR`
$bits \times int \to bits$	`$SHIFT_LEFT`, `$SHIFT_RIGHT`
$bits \to bits$	`$BIT_NOT`
$int \to bits$	`$INT_TO_BITS`
$bits \to int$	`$BITS_TO_INT`
$\to bool$	`$T`, `$F`
$bool \times bool \to bool$	`$AND`, `$OR`
$bool \to bool$	`$TRUE`, `$NOT`
$bool \times term \times term \to term$	`$IF`
$term \times term \to bool$ (lexical)	`$ID`, `$LNE`, `$LLT`, `$LLE`, `$LGT`, `$LGE`
$term \times term \to bool$ (other)	`$OCCURS`, `$VOCCURS`, `$VFREE`, `$RENAME`
$term \to bool$	`$ATOMIC`, `$INT`, `$BITS`, `$VAR`, `$GROUND`
$\to int$	`$NEXT_CL_NUM`, `$UNIQUE_NUM`

Additional notes on the operations (unless otherwise stated, the term in question evaluates if all arguments demodulate/evaluate to the appropriate type):

- $int \times int \to int$. The symbol `$SUM` is addition, `$PROD` is multiplication, `$DIFF` is subtraction, `$DIV` is integer division, and `$MOD` is remainder.
- $float \times float \to float$. These operations are analogous to the integer operations

except that there is no floating-point remainder operation. The syntax of floating-point numbers is described in Sec. 19.8

- $int \times int \rightarrow bool$. These are the ordinary relational operations on integers. The symbol `$EQ` is $=$, `$NE` is $\neq$, `$LT` is $<$, `$LE` is $\leq$, `$GT` is $>$, and `$GE` is $\geq$.

- $bits \times int \rightarrow bits$. The shift operations `$SHIFT_LEFT` and `$SHIFT_RIGHT` shift the first argument by the number of places given by the second argument.

- $bits \times bits \rightarrow bits$. The symbols `$BIT_AND`, `$BIT_OR`, and `$BIT_XOR` are the bitwise conjunction, disjunction, and exclusive-or operations.

- $bits \rightarrow bits$. The symbol `$BIT_NOT` is the one's complement operation on bit strings.

- $int \rightarrow bits$. The symbol `$INTS_TO_BITS` translates a decimal integer to a bit string.

- $bits \rightarrow int$. The symbol `$BITS_TO_INT` translates a bit string to the corresponding decimal integer.

- $\rightarrow bool$. The symbols `$T` and `$F` represent *true* and *false*. When they appear as literals or atomic formulas in clauses, the clauses are simplified as appropriate.

- $bool \rightarrow bool$. The symbol `$TRUE` is essentially a "no operation" on Boolean constants. It is used to trick hyperresolution into evaluating literals (see below).

- $bool \times term \times term \rightarrow term$. The `$IF` function is the *if-then-else* operator. When inside-out (the default) demodulation encounters a term `$IF(`*condition*`,` t_1`,` t_2`)`, demodulation takes a path different from its normal 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 the condition evaluates to a Boolean value, demodulation deviates from its inside-out behavior, because just one of t_1 and t_2 is demodulated. (If demodulation were always outside-in, `$IF` would not need to be built in because it could be efficiently defined with the two demodulators `if($T,x,y)=x` and `if($F,x,y)=y`.)

- $term \times term \rightarrow bool$ (lexical). These operations are analogous to the six operations in $int \times int \rightarrow bool$ except that the comparisons are lexical instead of arithmetic. The symbol `$ID` tests identity of terms. The lexical comparison is the same as in lex-dependent demodulation; in particular, the flag `lex_order_vars` (Secs. 6.1.6 and 8.1.1) is consulted during these operations.

- $term \times term \rightarrow bool$ (other). The term `$OCCURS(`t_1`,`t_2`)` is true if t_1 is a subterm of t_2, including the case when they are the same. The term `$VOCCURS(`t_1`,`t_2`)` is true if t_1 is a variable that occurs in t_2. The term `$VFREE(`t_1`,`t_2`)` is true if t_1 is a variable that does not occur in t_2. The term `$RENAME(`t_1`,`t_2`)` is true if t_1 and t_2 have the same structure; that is, if we rename all variables to x, the terms are identical.

- $term \rightarrow bool$. A term is `$ATOMIC` iff it is a constant (including integer and bit string), a term is a `$INT` iff it is an integer, a term is a `$BITS` iff it is a string of {0,1}, a term is a `$VAR` iff it is a (unbound) variable, and a term is a `$GROUND` iff it does not contain any variables.

- $\rightarrow int$. The term `$NEXT_CL_NUM` (no arguments) evaluates to the next integer that will be assigned as a clause identifier (this is useful for placing the ID of a clause within the clause). A sequence of calls to `$UNIQUE_NUM` (no arguments) returns $[1, 2, 3, \cdots]$.

Evaluation occurs as part of the demodulation process. In particular, if demodulation comes across an evaluable term, say `$SUM(2,3)`, it tries to convert the arguments into the appropriate type (integers for `$SUM`); then if the arguments have the correct type, it rewrites the term to the result of the operation, in this case, just as if the demodulator `$SUM(2,3)=5` had been present. The evaluation mechanisms, along with ordinary demodulation, form a reasonably complete (although not particularly speedy or convenient) equational programming subsystem.

Evaluation/demodulation can also occur, in a very particular way, during hyperresolution. (Recall that hyperresolution takes a clause, the *nucleus*, with some negative literals, the conditions, and resolves each negative literal with a positive clause, producing a clause with no negative literals.) Just as evaluation during demodulation can be thought of as rewriting with an implicit demodulator, evaluation during hyperresolution can be thought of resolving with the implicit positive unit clause `$T` (meaning "true"). The mechanism is this: if hyperresolution encounters a negative literal that has an evaluable predicate symbol, then it demodulates the atom (the literal without the sign); if the result of the demodulation is `$T`, then the literal is considered to have been resolved.

During hyperresolution, demodulation/evaluation is triggered by the presence of an evaluable literal. In many cases, however, the user defines a Boolean function to trigger the mechanism. Consider the following definition of list membership, written as demodulators:

```
member(x,[]) = $F.
member(x,[y|z]) = $IF($ID(x,y),
                      $T,
                      member(x,y)).
```

Because the symbol `member` is not evaluable, the demodulation/evaluation mechanism will not be activated; however, the unary evaluable predicate `$TRUE` can be used in the following way to trigger demodulation/evaluation.

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

Evaluable functions and predicates are useful to implement forward-chaining rule-based systems, for example, state-space search problems (Sec. 9.2).

Hyperresolution operates on the conditions (negative literals) in order, left to right. (The preceding sentence is not quite true because the first step is typically resolution of a positive given clause with any one of the conditions, but for this paragraph, we may assume that it is true.) If a literal resolves or evaluates, the next literal is considered. If nothing more can be done with a literal, then hyperresolution backtracks to the preceding literal in search of an alternative. When a nucleus contains evaluable conditions, the order of the conditions is important both for efficiency and for actually deriving hyperresolvents. Evaluable conditions typically have variables that must be instantiated when nonevaluable literals are resolved. If an evaluable literal is too far to the left, its variables will not be sufficiently instantiated when hyperresolution encounters it, evaluation will fail, and possible paths to hyperresolvents will be blocked. If an evaluable literal is too far to the right, then hyperresolution can explore many paths that are sure to fail.

Technical Note and Advice. The evaluable symbols are an add-on feature rather than an integral part of OTTER. In particular, the objects that are manipulated (integers, bit strings, etc.) in most cases are stored by OTTER as character strings rather than as the appropriate data type. To evaluate a term, say `$SUM(2,3)`, OTTER must find the strings `"2"` and `"3"` in a hash table, translate them to integers, add them, translate the result to the string `"5"`, then look up `"5"`, and possibly insert it into the hash table. This procedure is obviously much slower than it needs to be. If a problem requires a hundred million evaluations, the user should consider using something else, including writing a special-purpose C program.

Warning 1. The evaluable symbols should not be thought of as theories "built in" to OTTER. As theories, they are very incomplete, and OTTER uses them only in very constrained ways.

Warning 2. Ordinary resolution inference rules (e.g., `binary_res`, `hyper_res`, `ur_res`) never apply to evaluable literals.

9.1 Using More Natural Expressions for Evaluation

Writing complex evaluable expressions with `$`-symbols can be quite tedious. Therefore, a feature was added that allows more natural expressions. The command `make_evaluable` copies the evaluation properties from a `$`-symbol to any other symbol of the same arity. The form of the command is

    make_evaluable(*any-symbol*,*evaluable-symbol*).

The symbols in the command are given dummy arguments to specify the arity. The following list contains typical examples for integer arithmetic (assuming the symbols on the left are already known to be infix).

```
make_evaluable(_+_, $SUM(_,_)).
make_evaluable(_-_, $DIFF(_,_)).
make_evaluable(_>_, $GT(_,_)).
make_evaluable(_>=_, $GE(_,_)).
```

Warning 1. If a binary symbol that is recognized by paramodulation or demodulation as an equality symbol is given evaluation properties, it will no longer be recognized by paramodulation or demodulation. For example, if the command `make_evaluable(_=_, $EQ(_,_))` is issued, paramodulation and demodulation will not recognize `a=b` as an equality. The convention is to use == for evaluation.

Warning 2. This is not an "alias" mechanism; the symbols remain distinct for unification, matching, and identity testing.

9.2 Evaluation Examples

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

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

factorial(x) =    % factorial for nonnegative integers
   $IF($EQ(x,0),
       1,
       $PROD(x,factorial($DIFF(x,1)))).

quick_sort([]) = [].     % naive quicksort
quick_sort([x|y]) = append(quick_sort(le_list(x,y)),
                           [x|quick_sort(gt_list(x,y))]).
le_list(z,[]) = [].
le_list(z,[x|y]) = $IF($LLE(x,z),
                        [x|le_list(z,y)],
                        le_list(z,y)).
gt_list(z,[]) = [].
gt_list(z,[x|y]) = $IF($LGT(x,z),
                        [x|gt_list(z,y)],
                        gt_list(z,y)).
```

A State-Space Search. Here is a complete OTTER input file for a simple state-space search.

```
% We have a 3-gallon jug and a 4-gallon jug, both empty,
% and a well. Our goal is to have exactly 2 gallons in the
% 4-gallon jug.  We can fill a jug from the well, empty a
% jug onto the ground, and carefully pour water from one
% jug into the other.
%
% j(m, n) is the state in which the 3-gallon jug contains
% m gallons, and the 4-gallon jug contains n gallons.
```

```
set(hyper_res).

make_evaluable(_+_, $SUM(_,_)).
make_evaluable(_-_, $DIFF(_,_)).
make_evaluable(_<=_, $LE(_,_)).
make_evaluable(_>_, $GT(_,_)).

list(usable).
-j(x, y) | j(3, y).  % fill the 3-gallon jug
-j(x, y) | j(0, y).  % empty the 3-gallon jug
-j(x, y) | j(x, 4).  % fill the 4-gallon jug
-j(x, y) | j(x, 0).  % empty the 4-gallon jug
-j(x, y) | -(x+y <= 4) | j(0, y+x).     % small -> big; it fits
-j(x, y) | -(x+y >  4) | j(x- (4-y),4). % small -> big, until full
-j(x, y) | -(x+y <= 3) | j(x+y, 0).     % big -> small; it fits
-j(x, y) | -(x+y >  3) | j(3,y- (3-x)). % big -> small, until full

-j(x, 2).  % goal state --- 4-gallon jug containing 2 gallons
end_of_list.

list(sos).
j(0, 0).   % initial state --- both jugs empty
end_of_list.
```

10 Weighting

OTTER recognizes four lists of weight templates. (See Sec. 5.4 for input of weight template lists.)

`weight_list(pick_given).` This list is used for selection of given clauses from list `sos`. When the weight of a clause is printed, it is the `pick_given` weight.

`weight_list(purge_gen).` This list is used in conjunction with the `max_weight` parameter to discard generated clauses.

`weight_list(pick_and_purge).` In many cases, one can use the same weighting strategy for both selecting given clauses and purging generated clauses. The `pick_and_purge` list serves the purposes of both the `pick_given` and the `purge_gen` lists. If the `pick_and_purge` list is present, then neither the `pick_given` nor the `purge_gen` list may be present.

`weight_list(terms).` This list is for calculating the weight of terms when using the weight-lex-order (Sec. 8.1.1) to compare terms. This occurs when the flag `lrpo` is clear when orienting equality literals (Secs. 8.1.2 and 8.1.3).

10.1 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 (Sec. 6.2.5).

10.2 Weighing Atoms and Terms

Atoms and terms are weighed top-down. To weigh a given term, OTTER searches the appropriate weight list (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)).
```

Let $wt(t)$ be the weight of term or atom t. Then

$$wt(\texttt{f(g(h(a)),f(b,x))}) = 2{*}wt(\texttt{h(a)}) + (-3){*}wt(\texttt{f(b,x)}) + (-50).$$

If a matching weight template is not found, then the weight of the given term is 1 plus the sum of the weights of the subterms. (See the flags `atom_wt_max_args` and `term_wt_max_args`, Sec. 6.1.10, 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 (the symbol count).

Variables in weight templates are generic. A variable in a weight template will match any variable, and only a variable, 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.

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.

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 containing the Skolem symbol *after* the formula lists.

10.3 Containment Weight Templates

Term weighting has an additional feature that allows the user to specify terms that *contain* particular terms. This is done with a unary function symbol `$dots(`t`)`. If `$dots(`t`)`

occurs in a weight template, it will match any term that contains a term that matches t. This is very useful for discarding "bad" clauses. Here is part of an output file that illustrates this feature.

```
list(sos).
1 [] p(f(g(g(g(g(g(h(h(h(h(j(b))))))))))))).
2 [] p(F(g(g(h(g(H(g(h(g(g(g(B)))))))))))))).
3 [] p(f3(g(h(a)),g(g(b)),h(h(c)))).
end_of_list.

weight_list(pick_given).
weight(f($dots(j($(5)))),100).
weight(F($dots(H($dots(B)))),1000).
weight(f3($dots(a),$dots(b),$dots(c)),2000).
end_of_list.

======= end of input processing =======
=========== start of search ===========

given #1: (wt=106) 1 [] p(f(g(g(g(g(g(h(h(h(h(j(b))))))))))))).
given #2: (wt=1001) 2 [] p(F(g(g(h(g(H(g(h(g(g(g(B)))))))))))))).
given #3: (wt=2001) 3 [] p(f3(g(h(a)),g(g(b)),h(h(c)))).
```

11 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.

12 The Passive List

Either clauses or formulas can be input to list `passive`. After input, the passive list is fixed for the rest of the run. Clauses in the passive list are used for exactly two purposes: forward subsumption and unit conflict. If forward subsumption is enabled, a newly generated clause will be deleted if it is subsumed by any clause in `usable`, `sos`, or `passive`, and newly kept unit clauses are checked for unit conflict against unit clauses in `usable`, `sos`, or `passive`.

The passive list has been most useful for monitoring the progress of a search. Suppose we are trying to prove a difficult theorem, we have some lemmas in mind, and we would like to know whether OTTER has proved the lemmas. Then denials of the lemmas can be placed in the passive list, and OTTER will report proofs if it proves any lemmas, but the denials of the lemmas will not interfere with the search for the main theorem. (Recall that an appropriate value must be assigned to `max_proofs`; otherwise OTTER will stop at the first proof.)

13 Clause Attributes

Attributes can be attached to clauses. This feature was introduced at the same time as the hints strategy (Sec. 14), and all of the current attributes are specifically for the hints strategy. In case some future enhancements of OTTER will use attributes, the general attribute mechanism is given here.

Each attribute is identified by a name, and each attribute has a type. (Users cannot introduce new attributes—they are built into the code of OTTER.) The attribute types are integer, string, and term. Attributes are attached to clauses with the operator "`#`", and must appear after all literals.

For example, if attribute `a1` has type integer, attribute `a2` has type string, and attribute `a3` has type term, then a user can write a clause with attributes as follows.

```
f(x,y)!=f(x,z) | y=z # a1(23) # a2("left cancel") # a3(g(b)).
```

14 The Hints Strategy

The *hints strategy* can be used if the user has a set of clauses that might be relevant to finding a proof. The clauses, called *hints*, do not necessarily hold in the theory being explored, and they are not used for making inferences. Hints are used only as a heuristic for guiding the search, in particular, in selecting the given clauses and in deciding whether to keep derived clauses.

The main function of the hints strategy is to adjust the pick-given weight of clauses. The user can specify, for example, that any derived clause that matches a hint should have its pick-given weight reduced by 1000. In addition, the user can specify, with an attribute on a hint, how that hint should be used to adjust the pick-given weight of clauses that match the hint. Clauses can match hints in several ways as specified by (ordinary) parameters and by attributes on hints.

Because of the distinction between the pick-given and purge-gen weights of clauses, and because the hints mechanism affects only the pick-given weight, several additional flags exist. If a clause matching a hint is derived, one typically wants it to be kept so that it can be selected as a given clause. However, the clause may be discarded by the `max_weight` parameter. To address this problem, the flags `keep_hint_subsumers` and `keep_hint_equivalents` say that the `max_weight` parameter should be ignored for all clauses that match hints in those ways (details are in the following subsections).

The hints strategy was introduced by Bob Veroff, who implemented it in a previous version of OTTER and used it in many applications [27, 28]. OTTER currently has two separate hints mechanisms, named "hints" and "hints2", both derived from Veroff's methods and ideas. The first is more general, and the second is much faster.

14.1 Hints (the General Version)

The hint clauses are given in one or more lists. All of the lists must be named "hints" as in the following example.

```
list(hints).
-p(x) | q(x) | r(x).
end_of_list.
```

A clause C can match a hint H in three ways.

1. C subsumes H. This is referred to as "bsub", in analogy to back subsumption.
2. C is subsumed by H. This is referred to as "fsub", in analogy to forward subsumption.
3. C is equivalent to H.

Six parameters and two flags determine the behavior of the hints mechanism.

`assign(equiv_hint_wt,`n`)`. Default ∞, range $[-\infty..\infty]$. If $n \neq \infty$, clauses that are equivalent to a hint receive a pick-given weight of n.

`assign(equiv_hint_add_wt,`n`)`. Default 0, range $[-\infty..\infty]$. Clauses that are equivalent to a hint have n added to their ordinary pick-given weight.

`assign(fsub_hint_wt,`n`)`. Default ∞, range $[-\infty..\infty]$. If $n \neq \infty$, clauses that are subsumed by a hint receive a pick-given weight of n.

`assign(fsub_hint_add_wt,`n`)`. Default 0, range $[-\infty..\infty]$. Clauses that are subsumed by a hint have n added to their ordinary pick-given weight.

`assign(bsub_hint_wt,`n`)`. Default ∞, range $[-\infty..\infty]$. If $n \neq \infty$, clauses that subsume a hint receive a pick-given weight of n.

`assign(bsub_hint_add_wt,`n`)`. Default 0, range $[-\infty..\infty]$. Clauses that subsume a hint have n added to their ordinary pick-given weight.

A clause can match more than one hint, and a clause can match a hint in more than one way, so the order of operations is relevant. The rules are as follows. (1) The first hint (as given in the input file) that matches the clause is used. (2) Equivalence is tested first, then fsub, then bsub. (3) Within match type (e.g., bsub), both types of weight adjustment can be applied (e.g., `bsub_hint_wt` and `bsub_hint_ad_wt`).

The hint-adjustment parameters can be overridden by attributes on individual hints. (This feature can be used, for example, if some hints are more important than others.) The attribute names for hints correspond to the six parameters listed above. For example, if the parameter `bsub_hint_add_wt` is set to -1000, that value can be overridden for a particular hint by giving it an attribute as follows.

```
f(x,f(x,f(x,y)))=f(x,y) # bsub_hint_add_wt(-2000).
```

The following two flags were introduced because the hints mechanism adjusts the pick-given weight of clauses and does not affect the purge-gen weight of clauses.

Flag `keep_hint_subsumers`. Default clear. If this flag is set, then the `max_weight` parameter is ignored for derived clauses that subsume any hints.

Flag `keep_hint_equivalents`. Default clear. If this flag is set, then the `max_weight` parameter is ignored for derived clauses that are equivalent to any hints.

Practical Advice. If one has a set of clauses that one wishes to use as hints, say from a proof of a related theorem, a good first attempt is to use the following settings.

```
assign(bsub_hint_add_wt, -1000).
set(keep_hint_subsumers).
```

If one has more than a few hints, and one wishes to use only the preceding settings, then we recommend using hints2 (the fast version).

14.2 Hints2 (the Fast Version)

The general hints mechanism described in the preceding paragraphs can be very slow if there are many hints, because each hint clause is tested until a match is found. The fast hints mechanism uses indexing to find hints. To activate the fast hints mechanism, hint clauses are placed in one or more lists named “hints2” as in the following example.

```
list(hints2).
-p(x) | q(x) | r(x).
end_of_list.
```

Only one type of matching can be used with hints2—a clause matches a hint if and only if it subsumes the hint. Two parameters and two flags apply to hints2.

`assign(bsub_hint_wt,`n`)`. Default ∞, range $[-\infty..\infty]$. If $n \neq \infty$, clauses that subsume a hint receive a pick-given weight of n.

`assign(bsub_hint_add_wt,`n`)`. Default 0, range $[-\infty..\infty]$. Clauses that subsume a hint have n added to their ordinary pick-given weight.

Flag `keep_hint_subsumers`. Default clear. If this flag is set, then the `max_weight` parameter is ignored for derived clauses that subsume any hints.

Flag `degrade_hints2`. Default clear. If this flag is set, Bob Veroff's hint-degradation strategy is applied. When a hint is first read, its `bsub_hint_add_wt` is associated with it; this value can come from the parameter or from an attribute. The hint-degradation strategy says that each time a hint is used to adjust the weight of a derived clause, its `bsub_hint_add_wt` is cut in half. Assuming that it initially has a large negative value, this strategy makes a hint progressively less important as it matches more derived clauses. This strategy was introduced because (contrary to intuition) many different generalizations of a hint can be derived.

14.3 Label Attributes on Hints

If a hint clause has a label attribute, for example,

```
f(x,f(x,f(x,y)))=f(x,y) # label("hint 32 from proof 12").
```

and if such a hint is used to adjust the pick-given weight of a clause C, the label is inherited by C. This feature, which is useful for tracking the application of hints, applies to both the general and the fast hints mechanisms.

14.4 Generating Hints from Proofs

The main source for hints is proofs of related theorems. (If the goal is to shorten a proof, hints often come from proofs of the same theorem.)

Flag `print_proof_as_hints`. Default clear. If this flag is set, then whenever a proof is found, it is printed as a hints list in a form that can be input to a subsequent OTTER job. The proof that is printed contains more detail than the ordinary proofs printed by OTTER. In particular, when the proof contains demodulation, clauses are printed for each rewrite step of demodulation. This flag is independent of the hints mechanism.

15 Interaction during the Search

OTTER has a primitive interactive feature that allows the user to interrupt the search, modify the options, and then continue the search. The interrupt is triggered in two ways: (1) with OTTER running in the foreground, the user types the "interrupt" character (often DELETE or control-C), or (2) if the parameter `interrupt_given` is set to n, the search is interrupted after every n given clauses. When interrupted, OTTER immediately goes into a simple loop to read and execute commands. The accepted commands are listed in Table 8.

The following notes elaborate on the interactive feature.

- The flag `interactive_given` (Sec. 6.1.1) can be useful with the interactive feature. For example, if one thinks the search is going to fail, one can interrupt it, print list `sos`, set the `interactive_given` flag, then continue, selecting given clauses interactively.

Table 8: Interaction Commands

`help.`	Give simple help.
`set(`*flag-name*`).`	Set a flag.
`clear(`*flag-name*`).`	Clear a flag.
`assign(`*param-name*`,`*value*`).`	Assign a value to a parameter.
`stats.`	Send statistics to std. output and the terminal.
`usable.`	Print list `usable` on the terminal.
`sos.`	Print list `sos` on the terminal.
`demodulators.`	Print list `demodulators` on the terminal.
`passive.`	Print list `passive` on the terminal.
`fork.`	Fork and run the child process; resume parent when child finishes.
`continue.`	Continue the search.
`kill.`	Send statistics to standard output, and exit.

- The `fork` command creates a separate copy, called a *child*, of the entire OTTER process. Immediately after the fork, the child is running (waiting for more commands), and the original process, the *parent*, is waiting for the child to finish. When the child finishes, the parent resumes (waiting for more commands). Changes that the child makes to the clause space, options, and so forth, are not reflected in the parent; when the parent resumes, it is in exactly the same state as when the fork occurred. (The timing statistics are not handled correctly in child processes; CPU times are from the start of the current process; wall-clock time is correct; other timings are not reliable.)
- The interactive routine is an area where a user who is also a C programmer can easily add features. For example, most of the ordinary input commands could be made available in the interactive mode.
- This kind of interaction can be disabled by using the command `clear(sigint_interact).`

Warning. Do not interactively change any option that affects term or literal indexing.

16 Output and Exit Codes

OTTER sends most of its output to "standard output", which is usually redirected by the user to a file; we just call it the output file. The first part of the output file is an echo of most of the input and some additional information, including identification numbers for clauses and description of some input processing. Comments are not echoed to the output. The second part of the output file reflects the search. Various print flags determine what is output. Given clauses, generated clauses, kept clauses, and several messages about the processing of generated and kept clauses can be printed. Both statistics from the parameter `report` and proofs can also be printed during the search. The final part of the output file lists counts of various events (such as clauses given and clauses kept) and times for various operations.

Whenever a clause is printed, it is printed with its integer identifier (ID) and a justification list, which is enclosed in brackets. Examples:

```
4   [] -j(x,y)|j(x,0).
13 [hyper,11,8,eval,demod] j(3,1).
41 [31,demod] p([a,b,b,c,c,c,d,e,f]).
14 [new_demod,13] f(y,f(y,f(y,x)))=x.
71 [back_demod,58,demod,70,14,55,11,34,11] e!=e.
12 [demod,9] f(a,f(b,f(g(a),g(b))))!=e.
77 [binary,57.3,30.2] sm|mm| -sl.
33,32 [para_from,26.1.1,15.1.1.2,demod,21] g(x)=f(x,x).
36 [hyper,31,2,26,30,unit_del,19,18,20,19] p(k,g(k)).
4   [factor_simp,factor_simp]
   p(x)|p($f1(x))| -q($f2(y))| -q(y)|p($c6).
199 [binary,198.1,191.1,factor_simp] q($c14).
```

If the justification list is empty, the clause was input. Otherwise, the *first* item in the justification list is one of the following.

An inference rule. The clause was generated by an inference rule. The IDs of the parents are listed after the inference rule with the given clause ID listed first (unless `order_history is set`).

A clause identifier. The clause was generated by the `demod_inf` rule.

`new_demod`**.** The clause is a dynamically generated demodulator; it is a copy of the clause whose ID is listed after `new_demod`.

`back_demod`**.** The clause was generated by back demodulating the clause whose ID is listed after `back_demod`.

`demod`**.** The clause was generated by back demodulating an input clause.

`factor_simp`**.** The clause was generated by factor-simplifying an input clause. For example, `p(x)|p(a)` factor-simplifies to `p(a)`.

The sublist [`demod`, $id_1, id_2, \ldots$] indicates demodulation with $id_1, id_2, \ldots$. The sublist [`unit_del`, $id_1, id_2, \ldots$] indicates unit deletion with $id_1, id_2, \ldots$. The symbols `eval` indicates that a literal was "resolved" by evaluation (Sec. 9) during hyperresolution. The sublist [`factor_simp`, `factor_simp`, ...] indicates a sequence of factor-simplification steps (Sec. 6.1.5).

In proofs, some clauses are printed with two (consecutive) IDs. In such a case, the clause is a dynamically generated demodulator, and the two IDs refer to different copies of the same clause: the first ID refers to its use for inference rules, and the second to its use as a demodulator.

If the flag `detailed_history` is set, then for the inference rules `binary_res`, `para_from`, and `para_into`, the positions of the unified literals or terms are listed along with the parent IDs. For example, `[binary,57.3,30.2]` means that the third literal of clause 57 was resolved with the second literal of clause 30. For paramodulation, the "from" parent is listed as $ID.i.j$, where i is the literal number of the equality literal, and j (either 1 or 2) is the number of the unified equality argument; the "into" parent is listed as $ID.i.j_1.\cdots.j_n$, where i is the literal number of the "into" term, and $j_1.\cdots.j_n$ is the *position vector* of the "into" term; for example, `400.3.1.2` refers to the second argument of the first argument of third literal of clause 400. If the flag `para_all` is set, then the paramodulation positions are not listed.

When the flag `sos_queue` is set, the search is breadth first (level saturation), and OTTER sends a message to the output file when given clauses start on a new level. (Input clauses have level 0, and generated clauses have level one greater than the maximum of the levels of the parents. Since clauses are given in the order in which they are retained, the level of given clauses never decreases.)

Exit Codes. When OTTER stops running, it sends an exit code to the operating system, giving the reason for termination. The codes are useful when another program or system calls OTTER. Table 9 lists the exit codes. Note that we do not follow the UNIX convention of returning zero for normal and nonzero for abnormal termination.

Table 9: Exit Codes

101	Input error(s)
102	Abnormal end (compile-time limit or OTTER bug)
103	Proof(s) found (stopped by `max_proofs`)
104	`sos` list empty
105	`max_given` parameter exceeded
106	`max_seconds` parameter exceeded
107	`max_gen` parameter exceeded
108	`max_kept` parameter exceeded
109	`max_mem` parameter exceeded
110	Operating system out of memory
111	Interactive exit
112	Memory error (probable OTTER bug)
113	A USR1 signal was received
114	The splitting rule terminated with a possible model
115	`max_levels` parameter exceeded

17 Controlling Memory

In many OTTER searches, the `sos` list accumulates many clauses that never enter the search, possibly wasting a lot of memory. The normal way to conserve memory is to put a maximum on the weight of kept clauses. It can be difficult, however, to find an appropriate maximum. OTTER has a feature, enabled by the command `set(control_memory)`, that attempts to automatically adjust the maximum.

The memory-control feature operates as follows. When one third of available memory (`max_mem` parameter) has been filled, OTTER assigns or reassigns a maximum weight. The new maximum, say n, is such that 5% of all clauses in `sos` have weight $\leq n$. From then on, at every tenth iteration of the main loop, OTTER calculates a prospective new maximum n' in the same way. If $n' < n$, then the maximum is reset to n'. The values $1/3$ and 5%

were determined by trial and error. Perhaps these values should be parameters.

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. That is, the first few steps in the 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 parameters `change_limit_after` and `new_max_weight` to change the value of `max_weight` after a specified number of given clauses.

`assign(change_limit_after,`n`)`. Default 0, range [0..∞]. If n (the value) is not 0, this parameter has effect. After n given clauses have been used, the parameter `max_weight` is automatically reset to the value of the parameter `new_max_weight`.

`assign(new_max_weight,`n`)`. Default ∞, range [$-\infty..\infty$]. See the description of the preceding parameter.

Note that the memory-control feature (Sec. 17) can also address the complexity hump phenomenon.

18 Autonomous Mode

If the flag `auto` is set, OTTER will scan the input clauses for some simple syntactic properties and decide on inference rules and a search strategy. We think of the autonomous mode as providing a built-in metastrategy for selecting search strategies. The search strategy that OTTER selects for a particular set of clauses is usually refutation complete (except for the flag `control_memory`), but the user should not expect it to be especially effective. It will find proofs for many easy theorems, and even for cases in which it fails to find a proof, it provides a reasonable starting point.

In the input file, the command `set(auto)` must occur before any input clauses, and all input clauses must be in list `usable`; it is an error to place input clauses on any of the other lists when in autonomous mode. OTTER will move some of the input clauses to `sos` before starting the search. When OTTER processes the `set(auto)` command, it alters some options, even before examining the input clauses. If the user wishes to augment the autonomous mode by including some ordinary OTTER commands (including overriding OTTER's choices), the commands should be placed after `set(auto)` and before `list(usable)`.

After `list(usable)` has been read, OTTER examines the input clauses for several syntactic properties and decides which inference rules and strategies should be used, and which clauses should be moved to `sos`. The user cannot override the decisions that OTTER makes at this stage.

OTTER looks for the following syntactic properties of the set of input clauses: (1) whether it is propositional, (2) whether it is Horn, (3) whether equality is present, (4) whether equality axioms are present, and (5) the maximum number of literals in a clause. The program then considers six basic combinations of the properties: (1) propositional, (2) equality in which all clauses are units, and (3–6) the four combinations of {equality, Horn}. To see precisely what OTTER does for these cases, the reader can set up and run some simple experiments.

Please be aware that the autonomous mode reflects individual experience with OTTER; other users would certainly formulate different metastrategies. For example, one might prefer UR-resolution to hyperresolution or in addition to hyperresolution in rich Horn or nearly-Horn theories, and one might prefer to add few or no dynamic demodulators for equality theories.

19 Fringe Features

This section describes features that are new, not well tested, and not well documented. OTTER is not as robust when using these features, especially when more than fringe features is being used.

19.1 Ancestor Subsumption

OTTER does not necessarily prefer short or simple proofs—it simply reports the proofs that it finds. An 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.

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

When setting `ancestor_subsume`, we strongly recommend not clearing the flag `back_subsume`, because doing so can cause many occurrences of the same clause to be retained and used as given clauses.

19.2 The Hot List

The hot list is a strategy that can be used to emphasize particular clauses. It was invented by Larry Wos in the context of paramodulation, and it has been extended to most of OTTER's inference rules. To use the strategy, the user simply inputs one or more clauses in the special list named `hot`. Whenever a clause is generated and kept by OTTER's ordinary mechanisms, it is immediately considered for inference with clauses in the hot list.

Which Clauses Should Be Hot? Clauses input in the hot list are usually copies of clauses that occur also in `sos` or `usable`. They are typically clauses that the user believes will play a key role in the search for a proof, for example, special hypotheses.

Managing Hot-List Clauses. Input to the hot list is the same as input to other lists and can be in either clause or formula form, for example,

```
list(hot).
f(x,x) = x.  m(m(x)) = x.
end_of_list.
```

The flag `process_input` has no effect on hot-list clauses; they are never altered during input. Hot-list clauses are never deleted, for example by back subsumption or back demodulation. Even if a hot-list clause is identical to a clause in another list, it has a unique identifying number, and proofs that use hot-list clauses generally refer to two copies (with different ID numbers) of those clauses.

Hot Inference Rules. The inference rules that are applied to newly kept clauses and hot-list clauses are the same as the rules in effect for ordinary inference, with the exceptions `demod_inf`, `geometric_rule`, and `linked_ur_res`, which are never applied to hot-list clauses.

Applying Hot Inference. When hot inference is applied, the newly kept clause is treated as the given clause, and the hot list is treated as the usable list. (Note that the newly kept clause is not in the hot list, so it will not be considered for inference with itself, as happens with the given clause in ordinary inference.) For inference rules such as hyperresolution or UR-resolution that can use more than two parents, *all* of the other parents must be in the hot list; this generally means that the nucleus and other satellites must be in the hot list. Hot inference is not applied to clauses that are "kept" during processing of the input.

Level of Hot Inference (Parameter `heat`). To prevent long sequences of hot inferences (i.e., hot inference applied to a clause generated by hot inference, and so on) we consider the *heat level* of hot inference. The heat level of an ordinary inference is 0, and the heat level of a hotly inferred clause is one more than the heat level of the new-clause parent. The parameter `heat`, default 1, range [0..100], is the maximum heat level that will be generated. When a clause is printed, its heat level, if greater than 0, is also printed.

Dynamic Hot Clauses (Parameter `dynamic_heat_weight`). Clauses can be added to the hot list during a search. If the `pick_given` weight of a kept clause is less than or equal to the parameter `dynamic_heat_weight`, default $-\infty$, range $[-\infty..\infty]$, then the clause will be added to the hot list and used for subsequent hot inference. Input clauses that are "kept" during processing of the input are never made into dynamic hot clauses. Dynamic hot clauses can be added to an empty hot list (i.e., no input hot list).

19.3 Sequent Notation for Clauses

Two flags enable the use of sequent notation for clauses.

Flag `input_sequent`. Default clear. If this flag is set, clauses in the input file must be in sequent notation.

Flag `output_sequent`. Default clear. If this flag is set, then sequent notation is used when clauses are output.

Syntax:

- All sequent clauses have an arrow.
- The negative literals (if any) are written on the left side of the arrow, are written without the negation sign, and are separated by commas.
- The positive literals (if any) are written on the right side of the arrow and are separated by commas.

Table 10 lists some examples.

Table 10: Examples of Sequent Clauses

Ordinary Clause	Sequent Clause
`-p | -q | -r | s | t`	`p,q,r->s,t`
`p(a,b,c)`	`-> p(a,b,c)`
`a!=b`	`a=b ->`
`$F` (the empty clause)	`->`

Note that `p,q->r,s` is ordinarily thought of as (`p` *and* `q`) *implies* (`r` *or* `s`).

Sequent clauses are treated as (parsed as) a special case because they can't be made to fit within OTTER's ordinary syntax.

19.4 Conditional Demodulation

A conditional demodulator has the form

condition `->` *equality-literal*.

The equality is applied as a demodulator if and only if the instantiated *condition* evaluates to `$T`. The equality of a conditional demodulator is not subjected on input to being flipped or to being flagged as a lex-dependent demodulator, and conditional demodulators are never back demodulated. In other ways, conditional demodulators behave as ordinary demodulators. Examples are (`member` and `gcd` are defined in Sec. 9.)

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

19.5 Debugging Searches and Demodulation

The flag `very_verbose` causes too much output to be used with large searches. The following parameters can turn on verbose output for a segment of the search.

`assign(debug_first,`n`)`. Default 0, range [0 ..∞]. This parameter is consulted if the flag `very_verbose` is set. Verbose output will begin when a clause is kept and given an identifier of this value.

`assign(debug_first,`n`)`. Default -1, range [-1 ..∞]. This parameter is consulted if the flag `very_verbose` is set. Verbose output will end when a clause is kept and given an identifier of this value.

`assign(verbose_demod_skip,`n`)`. Default 0, range [0 ..∞]. This parameter is consulted during demodulation if the flag `very_verbose` is set. Verbose output will not occur during the first n rewrites.

19.6 Special Unary Function Demodulation

A feature, activated by the `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(_),f(_,_)]).

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 special-unary feature accomplishes just that. The command

```
special_unary([g(x)])
```

causes `g` to be ignored during term comparisons, and the expression will be demodulated to `a`. The `special_unary` command has no effect if the flag `lrpo` is set. *This is a highly experimental feature. Its behavior has not been well analyzed.*

19.7 The Invisible Argument

OTTER recognizes a built-in unary function symbol `$IGNORE(_)`. Forward subsumption treats each term that starts with `$IGNORE` as the constant `$IGNORE`, completely ignoring its argument. For example, `p(a,$IGNORE(b))` subsumes `p(a,$IGNORE(c))`. All other operations (in particular, inference rules, demodulation, and back subsumption) treat `$IGNORE` as an ordinary function symbol.

The purpose of `$IGNORE` is to record data about the derivation of a clause without having that data prevent the forward subsumption of clauses that would be subsumed without that data. The `$IGNORE` term is the term analog of the answer literal. For example, one can use `$IGNORE` terms in the jugs and water puzzle (Sec. 9.2) to record the sequence of pourings that leads to each state.

19.8 Floating-Point Operations

Table 11 lists a set of floating-point evaluable functions and predicates that are analogous to the integer arithmetic operations listed in Sec. 9. They operate in the same way as the integer operations.

Table 11: Floating-Point Operations

$float \times float \to float$	`$FSUM`, `$FPROD`, `$FDIFF`, `$FDIV`
$float \times float \to bool$	`$FEQ`, `$FNE`, `$FLT`, `$FLE`, `$FGT`, `$FGE`

The floating-point constants, however, are a little peculiar, both in the way they look and in the way they behave. They are written as quoted strings, using either single or double quotes. (Otherwise, they would not be able to contain decimal points.) Other than the quotation marks, the form of the floating-point constants accepted by OTTER is exactly the same as the form accepted by the C programming language (actually the C library used by the compiler). Examples are `"1.2"`, `"10e6"`, `"-3.333E-5"`. A floating-point constant must contain either a decimal point or an exponent character `e` or `E`.

The peculiar behavior comes from the fact OTTER stores the floating point numbers as character strings instead of directly as floating point numbers. To apply a floating-point operation, OTTER starts with the operand strings, translates them to true floating-point numbers (the C data type "double" is used), performs the operation, then translates the result into a string so that it can be an OTTER constant. As well as being inefficient, this scheme also has a problem with precision, because a fixed format is used to translate the results back into strings. The default format is `"%.12f"`, and it can be changed with a command such as

```
float_format("%17.8f")
```

Caution. OTTER does not check that the string in the `float_format` command is a well-formed format specification. This is the user's responsibility.

To fully understand how this works, see the standard C language reference [11, Appendix B]; in particular, the C library functions `sscanf` and `sprintf` are used to translate to and from strings.

19.9 Foreign Evaluable Functions

OTTER provides a general mechanism through which one can create one's own evaluable functions and predicates. The user (1) declares the function, its argument types, and its result type, (2) inserts a call to the function in the OTTER source code, (3) writes a C routine to implement the function, and (4) recompiles OTTER. The user must have a personal copy of the source code to use this feature. See the source code file `foreign.h` for step-by-step instructions, examples, templates, and test files.

Important note. Many times you can avoid having to do all of this by just writing your function with demodulators and using existing built-in functions. For example, if you need the maximum of two doubles, you can just use the demodulator `float_max(x,y) = $IF($FGT(x,y), x, y).`

19.10 The Inference Rule gL for Cubic Curves

Based on work of R. Padmanabhan and others, a new inference rule, gL ("geometric Law", or "Local to global"), was added to OTTER. The rule implements a local-to-global generalization principle that has a geometric interpretation for cubic curves. The article [20] and the monograph [18] contain descriptions of the rule, some details about its implementation in OTTER, and several new results obtained with its use.

The rule gL applies to single positive unit equalities, and it is implemented in two ways: as an inference rule, with unification, and as a rewrite rule, for when the target terms are already identical.

Flag `geometric_rule`. Default clear. When this flag is set, gL is applied as an inference rule (along with any other inference rules that are set) to each given clause. The rule gL applies to single positive unit equalities.

Flag `geometric_rewrite_before`. Default clear. When this flag is set, gL is applied as a rewrite rule, before ordinary demodulation, to each generated clause.

Flag `geometric_rewrite_after`. Default clear. When this flag is set, gL is applied as a rewrite rule, after ordinary demodulation, to each generated clause.

Flag `gl_demod`. Default clear. When this flag is set, ordinary demodulation is not applied to any derived clauses. Instead, after a clause is kept, it is copied, and the copy is demodulated and processed.

Our experience has shown that given two equalities of equal weight, one the result of gL and the other not, the gL result is usually more interesting. The following parameter can give preference to gL results.

`assign(geo_given_ratio,`n`).` Default 1, range $[-1..\infty]$. When this parameter is not -1, it affects selection of the given clause in a way similar to `pick_given_ratio`. If the ratio is n, then for each n given clauses selected in the normal way by weight, one given clause is selected because it is the lightest gL result available in `sos`. If pick_given_ratio and geo_given_ratio are both in effect, then clashes are resolved in favor of geo_given_ratio.

19.11 Linked UR-Resolution

OTTER has an inference rule, linked_ur_res, that is an application of the linked inference principle to UR-resolution. Linked inference rules can take much larger inference steps than the corresponding nonlinked rules, thereby avoiding the retention of many clauses that correspond to low-level deduction steps which can interfere with the overall proof search strategy.

We refer the reader to [29, 34, 26] for background on linked inference rules, and we focus here on specifying the constraints on linked UR-resolution for OTTER. The constraints are specified by six flags, two parameters, and annotations on input clauses.

Linked UR Flags

Flag linked_ur_res. Default clear. If this flag is set, linked UR-resolution is applied to all given clauses.

Flag linked_ur_trace. Default clear. If this flag is set, detailed information about the linking process is sent to the output file.

Flag linked_sub_unit_usable. Default clear. If this flag is set, intermediate unit clauses are checked for subsumption against the usable list.

Flag linked_sub_unit_sos. Default clear. If this flag is set, intermediate unit clauses are checked for subsumption against the sos list.

Flag linked_unit_del. Default clear. If this flag is set, unit deletion is applied to intermediate clauses.

Flag linked_target_all. Default clear. If this flag is set, any literal can be a target.

Linked UR Parameters

assign(max_ur_depth,n). Default 5, range [0 .. 100]. This parameter limits the depth of linked UR-resolution. Note that the depth of ordinary UR-resolution is 0.

assign(max_ur_deduction_size,n). Default 20, range [0 .. 100]. This parameter limits the size of linked UR-resolution inferences, that is, the number of corresponding binary resolution steps. In other words, the size of a linked inference step is one less than the number of clauses that participate.

Linked UR Annotations

Each clause that participates in a linked UR-resolution inference is classified as a *nucleus* (the nonunit clause containing the target literal), a *link* (nonunit clauses all of whose literals are resolved), or a *satellite* (unit clauses).

Input clauses can be annotated with special literals specifying the role(s) they can play in linked UR inferences. The clause annotations are as follows.

`$NUCLEUS([`*list-of-literal-numbers*`])` — the clause (assumed to be nonunit) can be a nucleus. The argument is a list of positive integers identifying the literals that can act as targets.

`$LINK([])` — the clause (assumed to be nonunit) can act as a link. The argument must be the empty list.

`$BOTH([`*list-of-literal-numbers*`])` — the clause (assumed to be nonunit) can be either a nucleus or a link, and when it is used as a nucleus, the admissible target literals are given in the list.

`$SATELLITE([])` — the clause (assumed to be unit) can act as a satellite. The argument must be the empty list.

For example, the annotation on the following four-literal clause says that it can act as a nucleus with the fourth literal as the target.

```
$NUCLEUS([4]) | -go | -P31 | -Q31 | R3_LD1_DS5.
```

Input clauses on the `usable` list must be annotated to participate in linked UR. Units on the list `sos` are assumed to be satellites and need not be annotated.

Most experiments with linked UR-resolution have been done under the following constraints. (1) Linked UR is the only inference rule being used, (2) every input clause in the `usable` list is annotated, and (3) every clause in the `sos` list is a unit and is *not* annotated. Linked UR seems to behave correctly under these constraints, but several problems have been noticed with other initial conditions.

Acknowledgment. The linked UR-resolution rule was implemented by Nick Karonis, with collaboration from Bob Veroff and Larry Wos.

19.12 Splitting

To address OTTER's poor performance on many non-Horn problems, a splitting rule was installed in OTTER (in November 1997). By "splitting" we mean that the search is divided into two or more independent branches such that if each of the branches is refuted, then the state before the split has been refuted. Splitting is typically recursive.

OTTER's splitting implementation uses the UNIX fork() system call, which creates copies of the state of the OTTER process. An additional hypothesis is asserted on the first branch, and the first branch continues executing while the second branch waits. If the first branch is refuted, the second branch starts running with its additional hypothesis. This method avoids explicit backtracking.

Two splitting methods are available: splitting on ground clauses, and splitting on ground atoms. In both methods, the parameter `split_depth` can be used to limit the depth of splitting. For example, with the command

```
assign(split_depth, 3).
```

a case such as [1.1.1.1] will not occur.

19.12.1 Splitting on Ground Clauses

Clause splitting can be triggered in two ways: either periodically or when a ground clause is selected as the given clause. In both methods, the clauses on which splitting occurs can be constrained by any of the following three flags (all clear by default).

```
set(split_pos).     % split on positive clauses only
set(split_neg).     % split on negative clauses only
set(split_nonhorn). % split on non-Horn clauses only
```

These flags determine eligibility. If none of the flags is set, all ground nonunit clauses are eligible.

Splitting Periodically on Clauses. To enable the periodic splitting method, one uses the following command.

```
set(split_clause).
```

The default period is every 5 given clauses. To change the period, say to 10 given clauses, use the following command.

```
assign(split_given, 10).
```

Instead of splitting after some number of given clauses, one can split after some number of seconds, say 4, with the following command.

```
assign(split_seconds, 4).
```

The clause on which to split can be selected from the set of eligible clauses in two ways. The default method is to select the first, lightest (using the pick-given scale) eligible clause from the sequence `usable+sos`. Instead, one can use the command

```
set(split_min_max).
```

which says to use the following method to compare two eligible clauses. Prefer the clause with the lighter heaviest literal (using the pick-given scale); if the heaviest literals have the same weight, use the lighter clause; if the clauses have the same weight, use the first in `usable+sos`.

Splitting When Given. To specify that clause splitting should be occur whenever an eligible clause is selected as the given clause, one uses the following command.

```
set(split_when_given).
```

The Branches for Clause Splitting. If OTTER decides to split on a clause, say `P | Q | R`, the assumptions for the three cases are

```
Case 1: P.
Case 2: -P & Q.
Case 3: -P & -Q & R.
```

One system fork occurs, and Case 1 executes. If it succeeds, a second fork occurs, and Case 2 executes. If that succeeds, Case 3 executes. If any of the cases fails to produce a refutation, the failure is propagated to the top, and the entire search fails.

19.12.2 Splitting on Atoms

To split on atoms, OTTER periodically selects a ground atom, say `P`, and considers two branches, one with assertion `P`, and the other with `-P`. The following command specifies splitting on atoms.

```
set(split_atom).
```

The parameters `split_given` and `split_seconds` determine (just as for clause splitting) when atom splitting occurs. If all input clauses are ground, and if the parameter `split_given` is assigned 0, then the resulting procedure is essentially a (very slow) Davis-Putnam-Loveland-Logemann SAT procedure.

An atom is eligible for splitting if it occurs in an eligible nonunit ground clause. Clause eligibility is determined just as in the clause splitting case, that is, by the flags `split_pos`, `split_neg`, and `split_nonhorn`.

All clauses in `usable+sos` are considered when deciding the best eligible atom. The default method select the lightest atom in the lightest clause (using the pick-given scale). An optional method for selecting an atom considers the number of occurrences of the atom. The command

```
set(split_popular).
```

says to prefer the atom that occurs in the greatest number of clauses.

Instead of having OTTER decide the atoms on which to split, the user can specify them in the input file with a command such as

```
split_atoms([P, a=b, R]).
```

which says to split, in order, on those atoms. In this example, we get eight cases, and then no more splitting occurs. The time at which the splitting occurs is determined, as above, by the parameters `split_given` and `split_seconds`.

19.12.3 More on Splitting

If OTTER fails to find a proof for a particular case (e.g., the list `sos` empties or some limit is reached), the whole attempt fails. If the search strategy is complete, then an empty `sos` list indicates satisfiability, and the set of assumptions introduced by splitting give a partial model. It is up to the user, however, to complete the model.

When OTTER finds a refutation by splitting, the output file does not contain an overall proof. A proof is given for each leaf in the tree, and those proofs contain clauses such as

```
496 [264,split.1.1.1.1] nop(C,D)!=nop(A,A).
```

in which the justification indicates that a split occurred on clause 264, and this clause is the assertion for case [1.1.1.1]. Other information about splitting is given in the output file, for example, when a split occurs, the case numbers, the case assertions, and when forked processed begin and end.

When splitting is enabled, the parameter `max_seconds` (for the initial process and all descendant processes) is checked against the wall clock (from the start of the initial process) instead of against the process clock. This is problematic if the computer is busy with other processes.

OTTER's splitting rule is highly experimental, and we do not have much experience with it. A general strategy that may be useful for non-Horn problems is the following.

```
set(split_when_given).
set(split_pos).           % Also try it without this command.
assign(split_depth, 10).
```

The OTTER distribution packages should contain a directory of sample input files that use the splitting rule.

Acknowledgment The splitting rule was developed in collaboration with Dale Myers, Rusty Lusk, and Mohammed Almulla.

20 Soundness and Completeness

20.1 Soundness

OTTER has a very good record with respect to soundness, but (as far as we know) no parts of it (the C code) have been formally verified. If anything depends on proofs found by OTTER, the proofs should be carefully checked, by hand or by an independent program.

The IVY project [19] contains a component that checks the proof objects (Sec. 6.1.8) produced by OTTER. The main result of the IVY project is a hybrid system, constructed in the ACL2 verification environment [10], that takes a first-order conjecture, translates it to clauses, sends the clauses to OTTER, and checks any proof objects that are returned. ACL2 has been used to prove various soundness properties of the clause translator, the proof checker, and their composition as a hybrid system.

20.2 Completeness

If the clause set does not involve equality, or if it involves equality and includes the equality axioms, then many of the common refutation-complete resolution search strategies can be easily achieved with OTTER. For example, hyperresolution and factoring, with positive clauses in the list `sos` and nonpositive clauses in the list `usable`, is complete. If the input clause set is Horn, then factoring is not required. The default method of selecting the given clause (take one with the fewest symbols) does not interfere with completeness, and neither forward nor back subsumption, as implemented in OTTER, interferes with completeness of the basic inference rules.

Completeness issues are more complex when paramodulation is the inference rule, especially when the set of support strategy is considered. A simple and complete paramodulation strategy for OTTER is (1) paramodulate *from* and *into* the given clause, (2) paramodulate *from* and *into* both sides of equality literals, (3) paramodulate *from* (but not *into*) variables, and (4) place all input clauses in the list `sos`. The equality `x=x` is required, but the functionally reflexive axioms are not required.

Completeness of the basic inference rules is important, but incomplete restrictions and refinements are frequently required to find proofs. For example, we almost always use the `max_weight` parameter; strictly speaking, it is incomplete, but it saves a lot of time and memory, and careful use of it does not prevent OTTER from finding proofs in practice. For paramodulation, we generally use the flag `anl_eq` with additional restrictions—some are known to be incomplete, and others have not been analyzed. We sometimes use UR-resolution on non-Horn sets, which is incomplete. And we make extensive use of weighting to purge uninteresting clauses and the options `delete_identical_nested_skolem`, `max_distinct_vars`, and `max_literals`, all of which interfere with completeness.

21 Limits, Abnormal Ends, and Fixes

OTTER has 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. (Of course, one must have one's own copy of the source code to do this.) Some of the limits are as follows.

`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 the forward subsumption discrimination tree.

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

Conserving Memory. Several steps can be taken if OTTER is using too much memory.

- Use max_weight to discard (more) generated clauses. This is a very effective way to save memory (and time).
- Set the flag control_memory (Sec. 6.1.10), or use the parameters change_limit_after and new_max_weight (Sec. 17).
- Decrease (down to 0) the value of the fpa_literals and fpa_terms parameters.
- Set the for_sub_fpa flag to switch forward subsumption indexing from discrimination tree to FPA indexing.
- If the inference rules being used are binary resolution or paramodulation, clear the flag detailed_history.
- If a lot of back subsumption or back demodulation is expected, set the flag really_delete_clauses (Sec. 6.1.10).
- If applicable, set no_fapl or no_fanl (Sec. 6.1.9).
- If back demodulation is being used, clear the flag index_for_back_demod.
- Run an OTTER job until memory runs out, collect interesting lemmas from the output file, then rerun the job including the lemmas as input clauses. Repeat. (This can be a good strategy even when memory is not a problem.)

22 Obtaining and Installing OTTER

OTTER 3 is free, and there are no restrictions on copying or distributing it. The main means of distribution is from the OTTER Web site at http://www.mcs.anl.gov/AR/otter/.

Once one has a copy of the OTTER 3 distribution directory, one should look at the file README for instructions on installing and testing OTTER. On UNIX-like systems, OTTER may have to be compiled. There may also be executable versions for Microsoft Windows available on the OTTER Web site.

Acknowledgments

Much of my work over the past few years has been in collaboration with Larry Wos. Toward our goal of creating programs that are expert assistants for mathematicians, logicians, engineers, and other scientists, we have worked together on many applications of automated deduction, and that work has led to many of OTTER's current features.

The basic design of the program, including the data structures and the use of indexing, descends mostly from theorem provers designed and implemented by Ross Overbeek. The indexing mechanisms, which are in large part responsible for the performance of the program, have benefited from discussions with Overbeek, Mark Stickel, and Rusty Lusk.

For many years Bob Veroff has maintained his own versions of OTTER. Many of his enhancements have been adopted for the official versions of OTTER.

The expert users of OTTER, including Larry Wos, Bob Veroff, John Kalman, Ken Kunen, Art Quaife, Dale Myers, Johan Belinfante, Michael Beeson, Branden Fitelson, and Zac Ernst, have tracked down bugs and suggested useful enhancements.

References

[1] W. Bledsoe and D. Loveland, editors. *Automated Theorem Proving: After 25 Years*, volume 29 of *Contemporary Mathematics*. AMS, 1984.

[2] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988.

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

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

[5] C. A. R. Hoare and M. J. C. Gordon, editors. *Mechanized Reasoning and Hardware Design*. Prentice Hall, 1992.

[6] J S. Moore, editor. Special Issue on System Verification. *J. Automated Reasoning*, 5(4), 1989.

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

[8] J. Kalman. *Automated Reasoning with Otter*. Rinton Press, Princeton, New Jersey, 2001.

[9] D. Kapur and H. Zhang. RRL: Rewrite Rule Laboratory User's Manual. Technical Report 89-03, Department of Computer Science, University of Iowa, 1989.

[10] M. Kaufmann, P. Manolios, and J Moore. *Computer-Aided Reasoning: An Approach*. Advances in Formal Methods. Kluwer Academic, 2000.

[11] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, second edition edition, 1988.

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

[13] A. G. Kurosh. *The Theory of Groups*, volume 1. Chelsea, New York, 1956.

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

[15] E. Lusk and R. Overbeek. The Automated Reasoning System ITP. Tech. Report ANL-84/27, Argonne National Laboratory, Argonne, IL, April 1984.

[16] W. McCune. Skolem functions and equality in automated deduction. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 246–251, Cambridge, MA, 1990. MIT Press.

[17] W. McCune. MACE 2.0 Reference Manual and Guide. Tech. Memo ANL/MCS-TM-249, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, June 2001.

[18] W. McCune and R. Padmanabhan. *Automated Deduction in Equational Logic and Cubic Curves*, volume 1095 of *Lecture Notes in Computer Science (AI subseries)*. Springer-Verlag, Berlin, 1996.

[19] W. McCune and O. Shumsky. IVY: A preprocessor and proof checker for first-order logic. In M. Kaufmann, P. Manolios, and J Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 16. Kluwer Academic, 2000.

[20] R. Padmanabhan and W. McCune. Automated reasoning about cubic curves. *Computers and Mathematics with Applications*, 29(2):17–26, 1995.

[21] A. Quaife. Automated development of Tarski's geometry. *J. Automated Reasoning*, 5(1):97–118, 1989.

[22] A. Quaife. *Automated Development of Fundamental Mathematical Theories*. PhD thesis, University of California at Berkeley, 1990.

[23] J. Siekmann and G. Wrightson, editors. *Automation of Reasoning: Classical Papers on Computational Logic*, volume 1 and 2. Springer-Verlag, Berlin, 1983.

[24] B. Smith. Reference Manual for the Environmental Theorem Prover: An Incarnation of AURA. Tech. Report ANL-88-2, Argonne National Laboratory, Argonne, IL, March 1988.

[25] G. Sutcliffe and C Suttner. The TPTP Problem Library for Automated Theorem Proving. `http://www.tptp.org/`.

[26] R. Veroff. An Algorithm for the Efficient Implementation of Linked UR-resolution. Tech. Report CD92-17, Department of Computer Science, University of New Mexico, 1992.

[27] R. Veroff. Using hints to increase the effectiveness of an automated reasoning program: Case studies. *J. Automated Reasoning*, 16(3):223–239, 1996.

[28] R. Veroff. Solving open questions and other challenge problems using proof sketches. *J. Automated Reasoning*, 27(2):157–174, 2001.

[29] R. Veroff and L. Wos. The linked inference principle, i: The formal treatment. *J. Automated Reasoning*, 8(2):213–274, 1992.

[30] L. Wos. *Automated Reasoning: 33 Basic Research Problems*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[31] L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning: Introduction and Applications,* 2nd edition. McGraw-Hill, New York, 1992.

[32] 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. *J. Automated Reasoning*, 1(1):5–48, 1985.

[33] L. Wos and G. Pieper. *A Fascinating Country in the World of Computing: Your Guide to Automated Reasoning*. World Scientific, Singapore, 1999.

[34] L. Wos, R. Veroff, B. Smith, and W. McCune. The linked inference principle II: The user's view. In R. Shostak, editor, *Proceedings of the 7th International Conference on Automated Deduction, Lecture Notes in Computer Science, Vol. 170*, pages 316–332, Berlin, 1984. Springer-Verlag.