Prover9 Manual  *Version Aug-2007*

Introduction

Prover9 is a resolution/paramodulation automated theorem prover for first-order and equational logic. Prover9 is a successor of the Otter Prover [McCune-Otter33].

Getting Started

Prover9 has a fully automatic mode in which the user simply gives it formulas representing the problem. See the Section Clauses and Formulas.

An good way to learn about Prover9 is to browse and study the example input and output files that are available. *Users are encouraged to contribute examples from their own work with Prover9 (and Mace4).*

Related Programs

Several programs come bundled with Prover9. The most important is Mace4, which looks for finite models and counterexamples. Mace4 can help avoid wasting time searching for a proof with Prover9 by first finding a counterexample or by first helping to debug logical specifications.

Another useful program is Prooftrans, which can transform proofs found by Prover9 in various ways, including producing more detailed proofs, simplifying the justifications, renumbering the steps, producing proofs in XML, and producing proofs for input to other programs.

Terms of Use

Prover9, Mace4, related programs, and the LADR libraries (with which they were all constructed) are distributed under the terms of the **GNU General Public License (v2)**.

Other Theorem Provers

- E is a very good all-around prover.
- Waldmeister is a fast prover for equational logic.
- Vampire has lately been winning the MIX category of CASC.
- Paradox is an excellent program for finding finite models and counterexamples.

Format Conventions for this Manual

Many parts of this manual are displayed in boxes with different background colors.

A display like the following indicates part of an input or output file.

```
formulas(sos).
  all x all y (subset(x,y)  (all z (member(z,x) -> member(z,y)))).
end_of_list.

formulas(goals).
  all x all y all z (subset(x,y) & subset(y,z) -> subset(x,z)).
end_of_list.
```

A display like the following indicates a job that is run on a command line, for example, a command to run a Prover9 job.

```
prover9 -f subset_trans.in > subset_trans.out
```

A display like the following indicates some output that appears on the computer screen, for example, a message from Prover9.

```
-------- Proof 1 --------
THEOREM PROVED
------ process 3666 exit (max_proofs) ------
```

Displays like the following contain algorithms.

```
Simplify clause (c):
    demodulate c
    merge identical literals
```

A display like the following notes an important difference between Prover9 and Otter.

> Prover9's automatic mode is set by default. Otter's automatic mode must be explicitly set.

Next Section: Installation

Installing Prover9, Mace4, and Friends

Unix-like Systems

Here is a quick example for Unix-like systems, including Linux and Macintosh OS X. Visit the Prover9 Web page and download the current version of LADR. The filename should be something like `LADR-June-2006A.tar.gz`; make sure that file is in your current directory. Run the following commands.

```
% zcat LADR-June-2006A.tar.gz | tar xvf -
% cd LADR-June-2006A
% make all
```

Prover9, Mace4, Prooftrans, and several other programs should now be in the directory `LADR-June-2006A/bin`. You can either include that directory in your search path or copy those programs to some directory that is already in your search path.

Microsoft Windows

For now, see that the Prover9 Web page.

Next Section: Running Prover9

Running Prover9

The standard way of running Prover9 is to (1) prepare an input file containing the logical specification of a conjecture and the search parameters, (2) issue a command that runs Prover9 on the input file and produces an output file, (3) look at the output, and (4) maybe run Prover9 again with different search parameters.

A graphical user interface (GUI) for Prover9 is under development, but it is not described in this manual. Nearly all of the information in this manual applies also when using the GUI.

An Input File

Here is an input file; assume it is named `subset_trans.in`.
(Use a plain text editor, not a word processor, to create input files.)

```
formulas(sos).
  all x all y (subset(x,y)  (all z (member(z,x) -> member(z,y)))).
end_of_list.

formulas(goals).
  all x all y all z (subset(x,y) & subset(y,z) -> subset(x,z)).
end_of_list.
```

A Basic Prover9 Command

Here is a command to run Prover9 on the preceding file and send the output to a file called `subset_trans.out`.

```
prover9 -f subset_trans.in > subset_trans.out
```

When you run the preceding command, a message like the following should appear immediately on your screen.

```
-------- Proof 1 --------
THEOREM PROVED
------ process 3666 exit (max_proofs) ------
```

The output file subset_trans.out should contain the proof (and a lot of other information about the job).

Taking Input from Standard Input

Prover9 jobs can be run in a slightly different way, taking input from "standard input" instead of a named file, as follows.

```
prover9 < subset_trans.in > subset_trans.out2
```

The disadvantage of using this method is that the name of the input file is not given in the output file.

More Than One Input File

The input can occur in more than one file:

```
prover9 -f subset.in trans.in > subset_trans.out3
```

All arguments after the "`-f`" are taken as input filenames, and there can be as many as you like. When multiple filnames are given on the command line, a list of objects (clauses, formulas, or terms) cannot be split across more than one file.

Time Limit on the Command Line

Prover9 also accepts a time limit, in seconds, on the command line. The following command limits the job to about 10 seconds.

```
prover9 -t 10 -f subset_trans.in > subset_trans.out4
```

If "`-t`" and "`-f`" are both in the command, the "`-t`" must occur first.

Getting Statistics During the Search

This section applies to Unix-like systems only.

If a Prover9 process is running in the background, one can tell it to send search statistics (without killing the job) to the output file sending a "USR1" signal to the process. For example,

```
% prover9 -f p3a.in > p3a.outb &
    [1] 31613
% kill -USR1 31613
    A report (17.75 seconds) has been sent to the output.
```

Calling Prover9 From Another Program

If Prover9 is called from another program (e.g., a shell script, a Perl script, or a Python script), Prover9's exit codes can tell the other program the reason Prover9 terminates. The following table shows the exit codes.

Exit Code	Reason for Termination
0 (MAX_PROOFS)	The specified number of proofs (**`max_proofs`**) was found.
1 (FATAL)	A fatal error occurred (user's syntax error or Prover9's bug).
2 (SOS_EMPTY)	Prover9 ran out of things to do (sos list exhausted).
3 (MAX_MEGS)	The **`max_megs`** (memory limit) parameter was exceeded.
4 (MAX_SECONDS)	The **`max_seconds`** parameter was exceeded.
5 (MAX_GIVEN)	The **`max_given`** parameter was exceeded.
6 (MAX_KEPT)	The **`max_kept`** parameter was exceeded.
7 (ACTION)	A Prover9 action terminated the search.

101 (SIGINT)	Prover9 received an interrupt signal.
102 (SIGSEGV)	Prover9 crashed, most probably due to a bug.

The calling program will probably want to look in Prover9's output, for example, to extract a proof. See the page on Prover9 output files.

Next Section: Input Files

Prover9 Input Files

Prover9 takes its input from one or more (usually one) files. If there is more than one input file, lists of objects (formulas, weighting rules, etc.) cannot be split across more than one file. The page Running Prover9 shows how to specify the files in the commands to run Prover9.

Comments and Whitespace

There are two kinds of comment:

- *Line comment*. If the first '%' (percent sign) on a line is not the start of a block comment ('%BEGIN'), everything from that symbol through the end of the line is ignored.
- *Block comment*. If the parser sees the string '%BEGIN', that is not in a line comment, it will ignore everything up through the next occurrence of 'END%'. Line breaks are irrelevant. If there is no 'END%', the rest of the file is ignored, without causing an error.

Comments are not echoed to the output file. Clauses can have label attributes which can serve as different kind of comment which *does* appear in the output file.

Whitespace (spaces, newlines, tabs, etc.) is optional in most situations. The important exception is that whitespace is required around some operations in clauses and formulas (see the page Clauses and Formulas).

A Simple Example

The most basic kind of input file consists of list of clauses named "sos" representing the negation of the conjecture, as in the following example.

```
formulas(sos).           % clauses to be placed in the sos list
  -man(x) | mortal(x).
  man(george).
  -mortal(george).
end_of_list.
```

Prover9 will take the clauses, use its automatic mode to decide on the inference rules, and then search for a refutation.

The preceding example can also be stated in a more natural way by using a non-clausal formula for the man-implies-mortal rule and the goals list for the conclusion, as follows.

```
formulas(assumptions).   % synonym for formulas(sos).
  man(x) -> mortal(x).   % open formula with free variable x
  man(george).
end_of_list.
```

```
formulas(goals).         % to be negated and placed in the sos list
  mortal(george).
end_of_list.
```

Prover9 will transform the formulas in this input to the same clauses as in the basic input above before starting the search for a refutation.

> In Otter and in earlier versions of Prover9, "clauses" and "formulas" were distinct types of object, and formulas could not have free variables. Now, clauses are a subset of formulas, and Prover9 decides which formulas are non-clausal and takes the appropriate actions to transform them to clauses.

Types of Input

Prover9 input consists of lists of objects (formulas or terms) and commands.

Lists of Objects

Lists of objects start with a type (`formulas` or `terms`) and name (`sos`, `goals`, `weights`, etc.), and end with `end_of_list`. The following display show an example of each type of accepted list, with one object in each list.

```
formulas(sos).           p(x).     end_of_list.   % the primary input list
formulas(assumptions).   p(x).     end_of_list.   % synonym for formulas(sos)
formulas(goals).         p(x).     end_of_list.   % some restrictions (see Goals)
formulas(usable).        p(x).     end_of_list.   % seldom used
formulas(demodulators).  f(x)=x.   end_of_list.   % seldom used, must be equalities
formulas(hints).         p(x).     end_of_list.   % should be used more often  (see Hints)

list(weights).         weight(a) = 10.                        end_of_list. % see Weighting
list(kbo_weights).     a = 3.                                 end_of_list. % see Term Ordering
list(actions).         given = 100 -> set(print_kept).        end_of_list. % see Actions
list(interpretations). interpretation(2,[],[relation(p,[1])]). end_of_list. % see Semantics
```

If the input contains more than one list of a particular type/name, the lists are simply concatenated by Prover9 as they are read.

Commands

Eleven types of command are accepted. Here is an example of each.

```
op(400, infix_right, ["+", "--"]). % declare parse precedence and type (see Clauses and Formulas)

redeclare(negation, "~"]).         % change the negation symbol (see Clauses and Formulas)

set(print_kept).                   % set a flag

clear(auto_inference).             % clear a flag

assign(max_weight, 40).            % integer parameter

assign(stats, some).               % string parameter

assoc_comm(*).                     % not currently used for Prover9
```

```
commutative(g).                    % not currently used for Prover9

predicate_order([=,<=,P,Q).        % predicate symbol precedence (see Term Ordering)

function_order([0,1,a,b,f,g,*,+]). % function symbol precedence (see Term Ordering)

lex([0,1,a,b,f,g,*,+]).            % synonym for "function_order"

skolem([a,b,f,g]).                 % declare symbols to be Skolem functions (rarely used)
```

Order of Commands and Lists of Objects

For the most part, the order of things in the input file(s) is irrelevant. For example, commands can usually be mixed with lists of objects. The situations in which order matters are listed here.

- The `op(precedence, type, symbols)` commands must occur before any clauses or formulas that contain the affected symbols.
- Some of the flags and parameters alter other flags and parameters. The alterations can be undone by placing the appropriate command after the command that alters. The output file clearly shows what happens in these cases.

Note that changing the order of clauses or formulas within a list, changing the order of literals in a clause, or changing the order of subformulas in a formula can change the search, occasionally in substantial ways.

Next Section: Clauses & Formulas

Clauses and Formulas

The Glossary Page contains definitions of term, atomic formula, literal, clause, and formula from a logical point of view. This page contains descriptions of how those kinds of things are parsed and printed, and we refer to them collectively as *objects*.

> In Otter and in earlier versions of Prover9, "clauses" and "formulas" were distinct types of object, and "formulas" could not have free variables. Now, clauses are a subset of formulas.

Here are the important points about clauses and formulas.

- Clauses are a subset of formulas. All input formulas, including clauses, appear in a list headed by `formulas(`*`list_name`*`).`
- There is a rule for distinguishing variables from constants, because clauses and other formulas can have free variables (variables not bound by quantifiers). The default rule is that variables start with (lower case) `u` through `z`. For example, in the formula `P(a,x)`, the term `a` is a constant, and `x` is a variable. (See also the flag **`prolog_style_variables`**.)
- Free variables in clauses and formulas are assumed to be universally quantified at the outermost level.
- Prover9's inference rules operate on clauses. If non-clausal formulas are input, Prover9 immediately translates them clauses by NNF, Skolemization, and CNF conversions.

Parsing and Printing Objects

The *prefix standard form* of an object with an n-ary symbol, say `f`, at the root is

```
f( argument_1, ..., argument_n )
```

Whitespace (spaces, tabs, newline, etc.) is accepted anywhere except within symbols.

Prover9 will accept any term or formula written prefix standard form. However formulas and many terms can be written in more convenient ways, for example, "`a=b | a!=c'`" instead of "`|(=(a,b),-(=(a,'(c))))`".

Prover9 uses a general mechanism in which binary and unary symbols can have special parsing properties such as "infix", "infix-right-associated", "postfix". In addition, each of those symbols has a precedence so that many parentheses can be omitted. (The mechanism is similar to those used by most Prolog systems.)

Many symbols have built-in parsing properties (see the table below), and the user can declare parsing properties for other symbols with the "op" command.

Clauses and formulas make extensive use of the built-in parsing properties for the equality relation and the logic connectives. Instead of first presenting the general mechanism, we will present the syntax for formulas under the assumption of the built-in parsing properties. The general mechanism is described below in the section Infix,

Prefix, and Postfix Declarations.

Symbols

Symbols include variables, constants, function symbols, predicate symbols, logic connectives. Symbols do not include parentheses or commas.

Prover9 recognizes several kinds of symbol.

- An *ordinary symbol* is a (maximal) string made from the characters `a-z`, `A-Z`, `0-9`, `$`, and `_`.
- A *special symbol* is a (maximal) string made from the *special characters*: `{+-*/\^<>=` ~?@&|!#';}`.
- A *quoted symbol* is any string enclosed in double quotes.
- The *empty list symbol* is `[]`. This is a special case.

The reason for separating ordinary and special symbols is so that strings like `a+b`; that is, `+(a,b)`, can be written without any whitespace around the `+`.

A symbol cannot have both ordinary and special characters, for example `R+` (unless it is a quoted symbol).

Objects (terms or formulas) are constructed from symbols, parentheses, and commas.

Overloaded Symbols

In most cases, symbol overloading is not allowed. For example a symbol cannot be both a function symbol and a predicate symbol, or both a constant and a binary function symbol. There are a few exceptions.

- The logic connectives can also be used as function or predicate symbols of the same arity. For example, `-` is typically used as unary arithmetic minus well as for logical negation.

  Prover9 is much more strict about overloading symbols than Otter is.

Symbols With Meaning

Several symbols have built-in meaning. These are the equality symbols (`=`, `!=`) and logic connectives (`-`, `|`, `&`, `->`, `<-`, `<->`, `all`, `exists`). These symbols can be changed as described in the section Redeclaring Built-in Symbols. (Parentheses, comma, period, and the list construction symbols cannot be redeclared.)

Terms

Any term can be written in prefix standard form, for example, `f(g(x),y)` and `*('(x),y)`. If symbols in the term have parsing/printing properties (either built-in) or declared with the `op` command), the term can be written in infix/prefix/postfix form with assumed precedence, for example, `x'*y`, which represents `*('(x),y)` under the built-in parsing/printing properties.

A list notation similar to Prolog's can be used to write terms that represent lists. Note that the "cons" operator is "`:`", instead of "`|`" as in Prolog.

Term	Standard Prefix Form	What it Is

`[]`	`$nil`	the empty list
`[a,b,c]`	`$cons(a,$cons(b,$cons(c,$nil)))`	list of three objects
`[a:b]`	`$cons(a,b)`	first, rest
`[a,b:c]`	`$cons(a,$cons(b,c))`	first, second, rest

Lists are frequently used in Prover9 commands such as the `function_order` command, and they are sometimes also used in clauses and formulas.

Atomic Formulas

Equality is a built-in special case. The binary predicate symbol = is usually written as an infix relation. The binary symbol `!=` is an abbreviation for "not equal"; that is, the formula `a!=b` stands for `-(a=b)`, or more precisely, `-(=(a,b))`. From the semantics point of view, the binary predicate symbol = is the one and only equality symbol for the inference rules that use equality.

Clauses

The disjunction (OR) symbol is `|`, and the negation (NOT) symbol is `-`. The disjunction symbol has higher precedence than the equality symbol, so equations in clauses do not need parentheses. Every clause ends with a period. Examples of clauses follow (Prover9 adds some extra space when printing clauses).

```
formulas(sos).
    p|-q|r.
    a=b|c!=d.
    f(x)!=f(y)|x=y.
end_of_list.
```

Formulas

Meaning	Connective	Example
negation	`-`	`(-p)`
disjunction	`|`	`(p | q | r)`
conjunction	`&`	`(p & q & r)`
implication	`->`	`(p -> q)`
backward implication		

Automatic Modes

Prover9's automatic mode is set by default. Otter's automatic mode must be explicitly set.

If you simply give Prover9 a set of clauses and/or formulas, Prover9 will look at the clauses and decide which inference rules and clause-processing operations to use. If you don't like the automatic decisions that Prover9 makes, you can clear the flag **auto** or any of the secondary auto flags that depend on it. Prover9 output files show in detail the effects of changing these flags.

```
set(auto).    % default set
clear(auto).
```

This is the basic automatic mode of Prover9. The only direct effect of this flag is that it changes four secondary auto flags as follows.

```
set(auto) -> set(auto_inference).
set(auto) -> set(auto_process).
set(auto) -> set(auto_limits).
set(auto) -> set(auto_denials).

clear(auto) -> clear(auto_inference).
clear(auto) -> clear(auto_process).
clear(auto) -> clear(auto_limits).
clear(auto) -> clear(auto_denials).
```

Any of the secondary flags, as well as the entire automatic mode can be cleared by the user.

```
set(auto_inference).    % default set
clear(auto_inference).
```

If this flag is set, the input clauses are checked for several syntactic properties such as the presence of equality and non-Horn clauses. Based on the results of the checks, Prover9 decides which inference rules to use. In addition, changing this flag causes the following changes.

```
set(auto_inference) -> set(predicate_elim).
set(auto_inference) -> assign(eq_defs, unfold).

clear(auto_inference) -> clear(predicate_elim).
clear(auto_inference) -> assign(eq_defs, pass).
```

```
set(auto_process).    % default set
clear(auto_process).
```

This flag causes several other flags that affect clause processing to be altered based syntactic properties of the initial clauses.

If all clauses are Horn and there are negative nonunits, the flag **back_unit_deletion** is automatically set. If there are non-Horn clauses, the flags **back_unit_deletion** and **factor** are automatically set.

Unlike ordinary option dependencies, the options that are changed by **auto_process** cannot be undone by placing commands in the input file, because they depend on the structure of the clauses.

```
set(auto_limits).    % default set
clear(auto_limits).
```

The only effect of changing this flag is that two parameters are changed in the following ways.

```
  set(auto_limits) -> assign(max_weight, 100).
  set(auto_limits) -> assign(sos_limit, 10000).

  clear(auto_limits) -> assign(max_weight, INT_MAX).
  clear(auto_limits) -> assign(sos_limit, -1).
```

An Experimental Automatic Mode

```
set(auto2).
clear(auto2).    % default clear
```

This is an enhanced automatic mode, developed in preparation for CASC-2005. The only direct effect of changing this option is that it causes several other options to be changed. See an output file to see the effects of setting this flag.

Automatically Adjusting the sos_limit Parameter

```
assign(lrs_ticks, n).  % default n=-1, range [-1 .. INT_MAX]

assign(lrs_interval, n).  % default n=50, range [1 .. INT_MAX]

assign(min_sos_limit, n).  % default n=0, range [0 .. INT_MAX]
```

These three parameters work together and are used to automatically adjust the parameter **sos_limit** by means of a "limited resource strategy" [RV-lrs]. If **lrs_ticks** ≥ 0, the method is applied.

This is an experimental feature and is not recommended for general use.

Next Section: Term Ordering

Term Ordering

> Prover9's term ordering procedures and options are simpler than Otter's, but somewhat less flexible. We recommend that those who use Otter's "ad hoc" ordering try Prover9's KBO ordering.

Prover9 has available several methods for comparing terms. (Although atomic formulas, literals, and clauses are not, strictly speaking, terms, the term orderings we write about here apply to those objects as well.)

The term orderings are partial (and sometimes total on ground terms), and they are used in two ways.

- To orient equalities (positive and negative). If one side of the equality is greater than the other, the greater side is placed on the left-hand side, and the equality is marked as oriented.
- To decide which literals in clauses are admissible for application of inference rules. Several of the resolution and paramodulation rules require that some of the literals be maximal in their clause.

For many problems, a good term ordering can determine the difference between success and failure. The default settings work well in many cases, but many difficult problems require adjustments to the term ordering.

The primary choice (via parameter **`order`**) is type of ordering: LPO, RPO, or KBO. Each of those types uses a symbol precedence (see the `function_order` and `predicate_order` commands), and KBO also uses a symbol weighting function (see the `list(kbo_weights)` command). In addition, the options **`eq_defs`** and **`inverse_order`** cause changes to the term ordering.

See [Dershowitz-termination] for a survey on term ordering.

The Primary Choice

The *symbol precedence* is is a total order on function and predicate symbols (including constants). The symbol weighting function maps symbols to nonnegative integers.

- LPO (Lexicographic Path Ordering). The term ordering is determined entirely by the symbol precedence. It is total on ground terms.
- RPO (Recursive Path Ordering). The term ordering is determined entirely by the symbol precedence. It is not necessarily total on ground terms, because the order of subterms is not considered.
- KBO (Knuth-Bendix Ordering). This ordering uses a weighting function on symbols as well as the symbol precedence. The weighting function is used first, and the symbol precedence breaks ties. It is total on ground terms.

KBO is perhaps the most natural of the three, because it it based on weights of symbols, but it is more cumbersome to specify because it is determined both by the symbol weights and by the symbol precedence. However, if one of the two terms being compared has more occurrences of a variable, it cannot be smaller. For

example, the distributivity equation cannot be oriented so that it distributes (expands) terms.

LPO is perhaps the most powerful of the three, because it can usually orient more equations. However, it allows rewrite rules that expand terms in explosive ways, for example (this is from a real problem),

> `(x * y) * z` rewrites to `E * (x * (E * (x * (E * (y * (E * (x * (E * (x * (E * z)))))))))))`

RPO is perhaps the least useful of the three, because it is not necessarily total on ground terms. That is, not all ground equations can be oriented. Also, see the sections on demodulation options and on inference rules.

The reasonable choice is usually between LPO (the default) and KBO. For many problems, either one is good. The main reason LPO is the the default is that it is a bit faster than KBO.

Here is the primary option.

`assign(order, `*`string`*`).  % default `*`string`*`=lpo, range [lpo,rpo,kbo]`

> This option is used to select the primary term ordering to be used for orienting equalities and for determining maximal literals in clauses. The choices are `lpo` (Lexicographic Path Ordering), `rpo` (Recursive Path Ordering), and `kbo` (Knuth-Bendix Ordering).

Termination of Demodulation

If each member of a set of demodulators (rewrite rules) is oriented with respect to the current ordering (LPO, RPO, or KBO), then demodulation (term rewriting) is guaranteed to terminate (in theory) on all terms, regardless of the the order in which the demodulators are applied or the order in which the subject terms are demodulated. However, there are sets of demodulators that are intractable in practice.

The Default Term Ordering

The default symbol precedence (for LPO, RPO, and KBO) is given by the following rules (in order).

- function symbols < equality symbol < non-equality predicate symbols;
- if the term ordering is KBO, and if there is exactly one unary function in the problem, that function is greater than all other functions;
- function symbols: arity-0 < arity-2 < arity-1 < arity-3 < arity-4 ... (note the position of arity-1);
- non-equality predicate symbols: lower arity < higher arity;
- non-Skolem symbols < Skolem symbols;
- for Skolem symbols, the lower index is the lesser;
- for non-Skolem symbols, more occurrences < fewer occurrences;
- the lexical ASCII ordering (UNIX strcmp() function).

The specific symbol precedence for a problem is given in the output file in the section `PROCESS INPUT`.

The default symbol-weighting function for KBO is given by the following rules.

- Variables have weight 1;
- if there is exactly one unary function in the problem, it has weight 0;
- all other symbols have weight 1.

Adjustments to the Term Ordering

The Symbol Precedence

The `function_order` and `predicate_order` commands can be used to assign a symbol precedence. (The `lex` command is synonym for `function_order`.) They contain lists of symbols ordered by increasing precedence. For example,

```
predicate_order([=, <=, P, Q]).          % = < P < Q
function_order([a, b, c, +, *, h, g]).   % a < b < c < + < * < h < g
```

There are two separate commands, because presecate symbols are always greater than function symbols.

If there are function or predicate symbols in the problem that do not appear in the corresponding command, a warning is issued, and Prover9 will complete the precedence inserting the missing symbols at the beginning of the precedence using its default rules. In these cases, the user should check that Prover9 has constructed a reasonable precedence.

Note that Skolem symbols cannot appear in a function_order command, because Skolem symbols do not exist at the time the function_order command is written. If there is a function_order command, and if Skolem symbols are generated, each one will be inserted, in effect, into the function_order command at a position just before the first symbol of higher arity. This method gives a symbol precedence similar to the default in many cases.

> Otter's lex command has a syntax that shows the arities of the symbols; Prover9's function_order and predicate_order commands list only the symbols. The arities are not necessary for Prover9, because a string cannot represent two symbols with different arities.

The KBO Weights

If the term ordering is KBO, `assign(`**`order`**`, kbo)`, the user can change the default symbol-weighting function. For example,

```
list(kbo_weights).
  a = 3.
  b = 2.
  * = 5.
  j = 22.
end_of_list.
```

(This has no relationship to the term-weighting function for selecting the given clause and discarding inferred clauses.)

If any symbols are absent from the list, they retain their default KBO weights of 1. The symbol weights must be greater than 0, with the exception that there may be one unary symbol of weight 0. (The definition of KBO allows for one unary symbol of weight 0 which must also be greatest in the precedence. This special case allows an such as `g(f(x,y)) = f(g(y),g(x))` to be oriented as shown and used as a rewrite rule.)

Term Ordering Options

```
set(inverse_order).    % default set
```

`clear(inverse_order).`

If this flag is set, if there is no function order command (which defines the function symbol precedence), and if the term ordering is LPO or RPO, then Prover9 will attempt to adjust the default symbol precedence if there are any input equations that specify an inverse operation. For example, if `f(x,g(x)) = c` is input, `g` will be placed after `f` in the precedence. This allows an equation such as `g(f(x,y)) = f(g(y),g(x))` to be oriented as shown for demodulation and paramodulation. If this flag is set, the `PROCESS INPUT` section of the output file shows how the flag changes the symbol precedence.

`assign(eq_defs, `*`string`*`).  % default `*`string`*`=unfold, range [unfold,fold,pass]`

If *string*=unfold, and if the input contains an equational definition, say `j(x,y) = f(f(x,x),f(y,y))`, the defined symbol `j` will be eliminated from the problem before the search starts. This procedure works by adjusting the symbol precedence so that the defining equation becomes a demodulator. If there is more than one equational definition, cycles are avoided by choosing a cycle-free subset of the definitions. If the primary term ordering is KBO, this option may admit demodulators that do not satisfy the KBO ordering, because a variable may have more variables on the right-hand side. However, this exception is safe (does not cause non-termination).

If *string*=fold, and if the input contains an equational definition, say `j(x,y) = f(f(x,x),f(y,y))`, the term ordering will be adjusted so that equation is flipped and becomes a demodulator which introduces the defined symbol whenever possible during the search.

If *string*=pass, nothing special happens. In this case, functions may still be unfolded or folded if the term ordering and symbol precedence happen to arrange the demodulators to do so.

Next Section: More Prep

More Search Prep

```
set(expand_relational_defs).
clear(expand_relational_defs).    % default clear
```

If this flag is set, Prover9 looks for relational definitions in the assumptions and uses them to rewrite all occurrences of the defined relations elsewhere in the input, before the start of the search. The expansion steps are detailed in the output file and appear in proofs with the justification `expand_def`.

Relational definitions must be closed formulas for example,

```
formulas(assumptions).
  all x all y all z (A(x,y,z)  ((x <= y & y <= z) | (z <= y & y <= x))).
end_of_list.
```

If there are circular definitions, Prover9 will immediately exit with a fatal error.

This flag eliminates predicate symbols, and its effects overlap somewhat with the flag **`predicate_elim`**.

Here is a trivial example, using the transitivity-of-subset problem.

```
prover9 -f subset_trans_expand.in > subset_trans_expand.out
```

For more examples using this flag, see the problem set "Ternary Relations in Lattices", which is available from the Prover9 Web page.

```
set(predicate_elim).    % default set
clear(predicate_elim).
```

If this flag is set, Prover9 applies a procedure that attempts to eliminate predicate symbols from the problem before the start of the search. The eliminations occur by resolution, and those steps show up as ordinary resolution inferences in any proofs that are found. The procedure works by selecting an eliminable predicate symbol, say `P`, then doing some set of resolution inferences on `P`, then removing all clauses that contain `P`. The procedure is intended to preserve unsatisfiability.

The effects of this flag overlap somewhat with **`expand_relational_defs`**, which also eliminates predicate symbols.

```
assign(fold_denial_max, n).  % default n=0, range [-1 .. INT_MAX]
```

This parameter applies to negated ground input equalities in which neither side is a constant, say `f(a,b) != f(b,a)`. If the left-hand side has fewer than *n* symbols, a new constant is introduced and set equal to the left-hand side. This operation is applied to at most one clause in the input `sos` list.

```
set(sort_initial_sos).
clear(sort_initial_sos).    % default clear
```

If this flag is set, the `sos` list is sorted just before the start of the search. The order (somewhat arbitrary) is

- positive clauses < negative clauses < mixed clauses;
- fewer symbols < more symbols;
- fewer literals < more literals;
- shallower < deeper.

```
set(process_initial_sos).    % default set
clear(process_initial_sos).
```

If this flag is set, clauses in the initial `sos` list will be handled (with a few exceptions) as if they were inferred. For example, demodulation, subsumption, and the check for unit conflict will be applied. The exceptions are that none of **`max_weight`**, **`max_vars`**, **`max_depth`**, or **`max_literals`** will be applied. (These four parameters are never applied before the first given clause is selected.)

This flag should be cleared only in very rare circumstances.

Next Section: Search Limits

Search Limits

`assign(sos_limit, n).  % default n=20000, range [-1 .. INT_MAX]`

This parameter imposes a limit on the size of the `sos` list (*n*=-1 means there is no limit). It also activates a method for deleting clauses (in addition to, and after, application of the **`max_weight`** parameter).

This is a little bit tricky (and sometimes too clever for its own good). When the `sos` is half full, it starts being selective about keeping clauses, and as it fills up, it gradually becomes more selective. When it is full, it is very selective about keeping clauses. (The method is not applied to clauses that match hints.) When it decides to keep a clause, and the `sos` is already full, the "worst" clause in `sos` is deleted to make room for the new clause.

More details will be added later.

`assign(max_given, n).  % default n=-1, range [-1 .. INT_MAX]`

This parameter will stop the search after *n* given clauses have been used. A value of -1 means that there is no limit.

`assign(max_kept, n).  % default n=-1, range [-1 .. INT_MAX]`

The search will stop when more than *n* clauses have been retained.

`assign(max_megs, n).  % default n=200, range [-1 .. INT_MAX]`

The search will stop when about *n* megabytes of memory have been used.

`assign(max_seconds, n).  % default n=-1, range [-1 .. INT_MAX]`

The search will stop at about *n* seconds. For UNIX-like systems, the "user CPU" time is used.

`assign(max_minutes, n).  % default n=-1, range [-1 .. INT_MAX]`

Changing this parameter simply changes **`max_seconds`** to the corresponding value.

```
assign(max_hours, n).  % default n=-1, range [-1 .. INT_MAX]
```

Changing this parameter simply changes **max_seconds** to the corresponding value.

```
assign(max_days, n).  % default n=-1, range [-1 .. INT_MAX]
```

Changing this parameter simply changes **max_seconds** to the corresponding value.

Next Section: The Loop

The Inference Loop

The *main loop* for inferring and processing clauses and searching for a proof is sometimes called the *given clause algorithm*. It operates mainly on the `sos` and `usable` lists.

```
While the sos list is not empty:
    1. Select a given clause from sos and move it to the usable list;
    2. Infer new clauses using the inference rules in effect;
       each new clause must have the given clause as one of its
       parents and members of the usable list as its other parents;
    3. process each new clause;
    4. append new clauses that pass the retention tests to the sos list.
end of while loop.
```

Two Frequently Asked Questions

> *At some point in the search, Prover9 has all of the clauses needed to make an important inference, and one of the potential parents is selected as the given clause. But Prover9 fails to make the inference. Why is that?*
>
> *Why do all parents have to be in the `usable` list?*

The answer to both questions is the same, and it has to do with redundancy. Assume

- clause *C* can be inferred from clauses *A* and *B*;
- both *A* and *B* are in the `sos` list; and
- *A* is selected first.

According to the algorithm, *C* is not derived until *B* has also been selected. Otherwise, *C* would be derived twice from *A* and *B*.

Variations of the Loop

There are two common versions of the given clause algorithm that differ in how and when simplification (i.e., rewriting) occurs.

In the *Otter loop*, which Prover9 uses, clauses in the `sos` list can simplify new clauses, and new simplifiers are applied immediately to all clauses, including `sos` clauses.

In the *Discount loop*, clauses in the `sos` list (also called the *passive list*) cannot simplify or be simplified until they are selected as given clauses.

The tradeoff between the two versions is straightforward --- the Otter loop spends much more time simplifying with the possible benefit of making an important simplification sooner.

Next Section: Select Given

Selecting the Given Clause

At each iteration of the main loop, Prover9 selects a *given clause* from the `sos` list, moves it to the `usable` list, and makes inferences from it and other clauses in the `usable` list.

A basic way to select the given clause is to always select the lightest clause in `sos`. Otter has the ability to mix two methods of selecting the given clause in a ratio determined by a parameter --- selecting the lightest clause and selecting the oldest clause. This method adds a breadth-first component to the search. See the **`pick_given_ratio`** parameter below.

Prover9 uses six components in selecting the given clause. The following six options are used.

```
assign(age_part, n).     % default n=1, range [0 .. INT_MAX]

assign(weight_part, n).  % default n=0, range [0 .. INT_MAX]

assign(false_part, n).   % default n=4, range [0 .. INT_MAX]

assign(true_part, n).    % default n=4, range [0 .. INT_MAX]

assign(random_part, n).  % default n=0, range [0 .. INT_MAX]

assign(hints_part, n).   % default n=INT_MAX, range [0 .. INT_MAX]
```

These six parameters work together to specify a 6-way ratio for selection of the given clauses:

- **`age_part`** refers to the clause with lowest ID (the oldest clause),
- **`weight_part`** refers to the (oldest) clause with lowest weight,
- **`false_part`** refers to the (oldest) lightest *false* clause,
- **`true_part`** refers to the (oldest) lightest *true* clause,
- **`random_part`** refers to a (pseudo-) *random* clause,
- **`hints_part`** refers to the lightest clause that matches a hint.

The *false/true* distinction is determined by a set of interpretations. The default interpretation is that negative clauses are false and non-negative clauses are true. To use explicit interpretations, see the section on semantic guidance.

Under the default interpretation, for example, if **`age_part`** = 1, **`false_part`** = 2, and **`true_part`** = 3, given clauses will be selected in a cycle of size six: one clause by lowest ID, then two clauses because they are the lightest negative (i.e., false) clauses, then three clauses

because they are the lightest non-negative (i.e., true) clauses. And so on.

If a clause of required type is not available, that component of the ratio is simply skipped. For example, with ratio in the preceding paragraph, if no false clauses are available, the cycle has size four (one part age, 3 parts true clauses) until some false clauses become available.

Note that the default value of **hints_part** is INT_MAX. This means that whenever the selection cycle gets to the **hints_part**, clauses that match hints will be selected as long as any are available.

When a given clause is printed, its sequence number, the reason it was selected, its weight, and its ID are also shown as in the following excerpt.

```
given #1  (I,wt=7): 9 x v y = y v x.  [input].
...
given #18 (T,wt=5): 28 x v x = x.  [para(13(a,1),14(a,1,2))].
given #19 (A,wt=11): 18 x ^ (y ^ z) = y ^ (x ^ z).  [para(11(a,1),12(a,1,1)),demod(12(2))].
given #20 (F,wt=21): 43 x ^ (((x v y) ^ z) v ((x v z) ^ y)) = (x ^ z) v (x ^ y) # label(false).  [para(11(a
```

The selection codes are A=age, W=weight, F=false, T=true, H=hints, R=random, and I=input (see flag **input_sos_first**).

More selection criteria will likely be added in future versions of Prover9.

Other Options

```
set(default_parts).      % default set
clear(default_parts).
```

If this flag is cleared, all of the selection parts, *including hints*, are set to 0. If it is set, the selection parts are reset to their defaults. This flag operates by making the following changes.

```
clear(default_parts) -> assign(hints_part, 0).
clear(default_parts) -> assign(weight_part, 0).
clear(default_parts) -> assign(age_part, 0).
clear(default_parts) -> assign(false_part, 0).
clear(default_parts) -> assign(true_part, 0).
clear(default_parts) -> assign(random_part, 0).

set(default_parts) -> assign(hints_part, INT_MAX).
set(default_parts) -> assign(weight_part, 0).
set(default_parts) -> assign(age_part, 1).
set(default_parts) -> assign(false_part, 4).
set(default_parts) -> assign(true_part, 4).
set(default_parts) -> assign(random_part, 0).
```

```
assign(pick_given_ratio, n).  % default n=0, range [0 .. INT_MAX]
```

If $n>0$, the given clauses are chosen in the ratio one part by age, and n parts by weight. The false/true distinction is ignored. This parameter works by making the following changes. (Note that this parameter does not alter **hints_part**, so that clauses matching hints may still be selected.)

```
assign(pick_given_ratio, n) -> assign(age_part, 1).
assign(pick_given_ratio, n) -> assign(weight_part, n).
assign(pick_given_ratio, n) -> assign(false_part, 0).
assign(pick_given_ratio, n) -> assign(true_part, 0).
assign(pick_given_ratio, n) -> assign(random_part, 0).
```

```
set(lightest_first).
clear(lightest_first).    % default clear
```

If this flag is set, the given clauses are selected by weight, lightest first. This flag operates by making the following changes. (Note that this flag does not alter **hints part**, so that clauses matching hints may still be selected.)

```
set(lightest_first) -> assign(weight_part, 1).
set(lightest_first) -> assign(age_part, 0).
set(lightest_first) -> assign(false_part, 0).
set(lightest_first) -> assign(true_part, 0).
set(lightest_first) -> assign(random_part, 0).
```

```
set(breadth_first).
clear(breadth_first).    % default clear
```

If this flag is set, the sos list operates as a queue, giving a breadth-first search. That is, the oldest clause is selected as the given clause. This flag operates by making the following changes. (Note that this flag does not alter **hints part**, so that clauses matching hints may still be selected.)

```
set(breadth_first) -> assign(age_part, 1).
set(breadth_first) -> assign(weight_part, 0).
set(breadth_first) -> assign(false_part, 0).
set(breadth_first) -> assign(true_part, 0).
set(breadth_first) -> assign(random_part, 0).
```

```
set(random_given).
clear(random_given).    % default clear
```

If this flag is set, a random clause is selected from the sos list. This flag operates by making the following changes. (Note that this flag does not alter **hints part**, so that clauses matching hints may still be selected.)

```
set(random_given) -> assign(random_part, 1).
set(random_given) -> assign(age_part, 0).
set(random_given) -> assign(weight_part, 0).
set(random_given) -> assign(false_part, 0).
set(random_given) -> assign(true_part, 0).
```

```
assign(random_seed, n).  % default n=0, range [-1 .. INT_MAX]
```

This parameter determines the seed for the (pseudo-) random number generator, which is used for the parameter **random part** (and maybe also for other purposes). The system library functions rand() and srand() are used for random number generation.

If $n \geq 0$, it is used as the seed. If $n = -1$, Prover9 selects a seed, based on the value of the system clock; in this case, Prover9 jobs are not reproducibe.

```
set(input_sos_first).    % default set
clear(input_sos_first).
```

If this flag is set, the clauses in the initial `sos` list are selected as given clauses (in the order in which they occur in the `sos` list) before any derived clauses are selected. This flag allows heavy input clauses to enter the search right away. After the initial clauses have been selected, the ordinary **`selection ratio`**, takes over.

Next Section: Inference Rules

Inference Rules

When a given clause is selected, all of the enabled inference rules are applied to it. For each inference, one of the parents is the given clause, and all other parents must be in the `usable` list.

Most inference rules distinguish the parents by the roles they play in the inference, e.g., positive or negative literal for binary resolution, nucleus or satellite for hyper rules, and from and into for paramodulation. The given clause can play any role in the inference.

After an inference rule generates a new clause, the clause is processed, which includes simplification operations such as demodulation and **`unit_deletion`**, and retention tests, such as **`max_weight`**. Processing also includes several operations that might be considered inference rules, such as **`factor`** and **`new_constants`**.

Prover9 uses ordered resolution and paramodulation with literal selection. These methods restrict the literals that are eligible for inference. The resolution and paramodulation inference rules are intended to be complete (exceptions are given in the descriptions of the options below), but we have not done a rigorous analysis of the algorithms, so users should not make any assumptions about completeness. For an overview of ordered inference with literal selection, see the section Ordered Inference below.

Binary Resolution Rules and Options

```
set(binary_resolution).
clear(binary_resolution).    % default clear
```

If this flag is set, binary resolution will be applied to the given clause. The options **`literal_selection`**, **`ordered_res`**, and **`check_res_instances`** determine eligible literals.

```
set(neg_binary_resolution).
clear(neg_binary_resolution).    % default clear
```

If this flag is set, negative binary resolution is applied to the given clause. That is, the negative resolved literal must be in a clause in which all literals are negative. The options **`ordered_res`**, and **`check_res_instances`** are also used to determine eligible literals.

Note that there is no inference rule "pos_binary_resolution". Positive binary resolution can be achieved by using the parameter **`literal_selection`** so that at least one negative literal is always selected. Positive binary resolution is not the dual of **`neg_binary_resolution`**, because the **`literal_selection`** technique is not symmetric between positive and negative literals; in particular, selected literals are always negative. The **`literal_selection`** parameter is always ignored for negative binary resolution.

```
set(ordered_res).    % default set
clear(ordered_res).
```

This option puts restrictions on the binary and hyperresolution inference rules (but not on UR-resolution). It says that resolved literals in one or more of the parents must be maximal in the clause.

See the section Ordered Inference below.

```
set(check_res_instances).
clear(check_res_instances).    % default clear
```

This flag applies to the binary and hyperresolution inference rules if the flag **ordered_res** is also set. If **check_res_instances** is set, the **ordered_res** test is is applied after the substitution for the inference has been applied to the parents.

```
assign(literal_selection, string).  % default string=maximal_negative, range [maximal_negative, all_negativ
```

This parameter affects to the inference rules **binary_res** and **paramodulation**. It determines which literals are eligible for inference. Here are the accepted values.

- `maximal_negative`: negative literals that are maximal w.r.t. the negative literals of the clause are marked;
- `all_negative`: all negative literals are marked;
- `none`: no literals are marked;

If at least one negative literal is always selected (e.g., `maximal_negative` or `all_negative`), binary resolution will be positive binary resolution, and paramodulation will be positive paramodulation.

Literal selection is ordinarily used with ordered inference (flags **ordered_res** and **ordered_para**), but it can be used without ordered inference.

```
set(positive_inference).    % default set
clear(positive_inference).
```

The only direct effect of this flag is that setting it causes the **literal_selection** parameter to be changed as follows.

```
set(positive_inference) -> assign(literal_selection, maximal_negative).
```

This setting causes **binary_resolution** to be positive binary resolution, and **paramodulation** to be positive paramodulation.

Hyper and UR Resolution Rules and Options

The Hyper and UR resultion rules can resolve more than one literal of one of the parent clauses (the nucleus) with other parent clauses (the satellites), all in one step. An application of one of these inference rules can be viewed as a sequence of binary resolution steps.

```
set(pos_hyper_resolution).
clear(pos_hyper_resolution).    % default clear
```

If this flag is set, positive hyperresolution (usually called simply hyperresolution) is applied to the given clause. If the flag **ordered_res** is set, the resolved literals in the satellites (positive parents) must be maximal. If the flags **ordered_res** and **check_res_instances** are both set, the maximality check is done after the substitution for the inference has been applied to the parents. Literal selection is not applied to hyperresolution.

```
set(hyper_resolution).
clear(hyper_resolution).    % default clear
```

This flag is a synonym for **pos_hyper_resolution**. The only effect of changing this flag is to make the corresponding change to the flag **pos_hyper_resolution**.

```
set(neg_hyper_resolution).
clear(neg_hyper_resolution).    % default clear
```

If this flag is set, negative hyperresolution is applied to the given clause. If the flag **ordered_res** is set, the resolved literals in the satellites (negative parents) must be maximal. If the flags **ordered_res** and **check_res_instances** are both set, the maximality check is done after the substitution for the inference has been applied to the parents. Literal selection is not applied to hyperresolution.

```
set(ur_resolution).
clear(ur_resolution).    % default clear
```

If this flag is set, UR resolution (unit-resulting resolution) is applied to the given clause. In fact, the only effect of this flag is that it automatically sets the flags **pos_ur_resolution** and **neg_ur_resolution**

UR resolution may be incomplete when there are non-Horn clauses.

```
set(pos_ur_resolution).
clear(pos_ur_resolution).    % default clear
```

If this flag is set, positive UR resolution is applied to the given clause. That is, the resulting unit clause is a positive clause. Neither ordering constraints nor literal selection is applied to UR resolution.

```
set(neg_ur_resolution).
clear(neg_ur_resolution).    % default clear
```

If this flag is set, negative UR resolution is applied to the given clause. That is, the resulting unit clause is a negative clause. Neither ordering constraints nor literal selection is applied to UR resolution.

```
set(initial_nuclei).
clear(initial_nuclei).    % default clear
```

This flag puts a restriction on the nucleus for the hyperresolution and UR-resolution inference rules. It says that each nucleus must be an input clause (more precisely, an initial clause).

Setting this flag may cause incompleteness of the inference system.

```
assign(ur_nucleus_limit, n).  % default n=-1, range [-1 .. INT_MAX]
```

If *n* != -1, then the nucleus for each UR-resolution inference can have at most *n* literals.

This option may cause incompleteness of the inference system.

Paramodulation Rules and Options

```
set(paramodulation).
clear(paramodulation).    % default clear
```

If this flag is set, paramodulation is applied to the given clause. If the from literal is oriented (oriented equalities are always *heavy=light*), the paramodulation is applied left-to-right. If the from literal cannot be oriented Prover9 attempts to paramodulate from both sides of it. Unlike the inference rule superposition, this inference rule goes into "light" sides of equations.

If the flag **`ordered_para`** is also set, ordered paramodulation is used.

If paramodulation involves non-unit clauses, **`literal_selection`** is used to determine eligible literals.

Setting the flag **`paramodulation`** causes the flag **`back_demod`** to be automatically set. Back demodulation can be disabled by placing `clear(back_demod)` after `set(paramodulation)` in the input file.

```
set(ordered_para).    % default set
clear(ordered_para).
```

This flag places a restrictions on the **`paramodulation`** inference rule, based on maximal literals. See the section Ordered Inference.

```
set(check_para_instances).
clear(check_para_instances).    % default clear
```

This flag applies to the **`paramodulation`** inference rule and is analogous to the flag **`check_res_instances`** for **`binary_resolution`**. It says to apply the ordering tests *after* the substitution for the inference has been applied to the parent claues.

```
set(para_from_vars).    % default set
```

```
clear(para_from_vars).
```

This flag says that paramodulation may occur from variables. That is, a literal $x=t$, in which x does not ocur in t, may be used as the from literal, unifying arbitrary terms with x, and replacing them with t.

For (unit) equational problems, this flag is nearly always irrelevant.

Clearing this flag may cause incompleteness of the inference system.

```
assign(para_lit_limit, n).  % default n=-1, range [-1 .. INT_MAX]
```

If $n \neq -1$, each parent in paramodulation can have at most n literals. This option may cause incompleteness of the inference system.

```
set(para_units_only).
clear(para_units_only).    % default clear
```

This flag says that both parents for paramodulation must be unit clauses. The only effect of this flag is to assign 1 to the parameter **`para_lit_limit`**.

Setting this flag may cause incompleteness of the inference system.

```
set(basic_paramodulation).
clear(basic_paramodulation).    % default clear
```

This option hasn't been implemented yet.

Ordered Inference

This section contains a practical overview of ordered inference as implemented in Prover9. For theoretical presentations, see [Bachmair-Ganzinger-res] and [Nieuwenhuis-Rubio-para].

Prover9 uses ordered inference with literal selection.

- *Ordered inference.*Within a clause, there is a partial order on literals, determined by the term ordering. A subset of the literals is marked as *maximal*. (If the clause is ground, the order is total, and the greatest literal is marked as maximal.) The inference rules may be restricted in some cases so that they apply only to maximal literals.
- *Literal selection.* In each clause, a subset of the *negative* literals of a clause is marked as *selected*. (Different clauses may have the subset chosen by different methods.) The inference rules may be restricted in some cases so that they apply only to selected literals.

Ordered inference and literal selection are typically used together, but each can be used without the other, by changing the options **`ordered_res`** and **`literal_selection`**. In the following, if **`ordered_res`** is disabled, simply assume all literals are maximal. The setting `assign(literal_selection, none)` has the effect of disabling literal selection.

Ordered Binary Resolution with Literal Selection

```
A positive literal PL in a clause C is eligible for resolution if
  no literal is selected in C, and PL is maximal in C.

A negative literal NL in a clause C is eligible for resolution if
  (1) NL is selected in C, or
  (2) no literal is selected in C, and NL is maximal in C.
```

Note that if at least one negative literals is selected in every clause, we have a version of positive binary resolution, because no literal may be selected in the clause containing the positive resolved literal.

Ordered Factoring

Prover9 does not do ordered factoring. Instead, if factoring is enabled (see flag **factor**), factoring is applied as much as possible to all newly kept clauses. In theory, factoring can be restricted to maximal literals without losing completeness, but we believe applying it eagerly is more practical.

Ordered Paramodulation with Selection

For ordered paramodulation with selection, literal eligibility for the "from" literal is that same as eligibilty of the positive literal for ordered resolution with selection.

Literal eligibility for *positive* "into" literals is that same as eligibilty of the positive literal for ordered resolution with selection.

Literal eligibility for *negative* "into" literals is the same as eligibilty of the negative literal for ordered resolution with selection.

In other words,

```
A positive literal PL in a clause C is eligible for paramodularion
  (as the "from" or the "into" parent) if no literal is selected in C,
  and PL is maximal in C.

A negative literal NL in a clause C is eligible for paramodulation if
  (1) NL is selected in C, or
  (2) no literal is selected in C, and NL is maximal in C.
```

Negative Ordered Binary Resolution

```
A positive literal NL in a clause C is eligible for resolution if
  NL is maximal among the positive literals of C.

A negative literal NL in a clause C is eligible for resolution if
  C has no positive literals, and NL is maximal in C.
```

Note that negative ordered binary resolution is not the dual of positive ordered binary resolution, because the negative version ignores literal selection.

Next Section: Process Inferred

Processing Inferred Clauses

Processing of inferred clauses is separated into two stages: (1) simplifying the clause and deciding whether to keep it, and if it is kept, (2) using the clause to operate on other clauses.

Processing Initial Clauses

Initial clauses in the `sos` list are processed, for the most part, as if they were derived by some inference rule. This process helps to ensure that Prover9's working set of clauses starts out in a good state, in particular, that no clause subsumes another, and that all clauses are simplified according to the working set of demodulators. Note the following exceptions.

- The main exceptions to processing initial clauses is that the parameters **`max_literals`**, **`max_vars`**, **`max_weight`**, and **`max_depth`** are not applied.
- All processing of initial `sos` clauses can be disabled by clearing the flag **`process_initial_sos`**.
- Clauses in the initial `usable` list are never processed.
- If there is an initial `demodulators` list, the clauses therein will be checked. If an equation is orientable but backward, it will be flipped, and a warning message will be printed. Otherwise, it must satisfy all of the ordinary constraints on demodulators. Having an initial `demodulators` list is useful, along with clearing the flag **`back_demod`**, if the user wishes to have a set of demodulators that is fixed throughout the search.

Algorithm for Processing Clauses

Processing initial and inferred clauses.

```
Start with clause c:
    1.  Simplify c:
        1a.  demodulate
        1b.  orient equalities
        1c.  simplify literals
        1d.  merge identical literals
     unitedeletion
     cac1fedundancy
    2.  safe_unit_conflict check
    3.  max_literals, max_depth, max_vars, max_weight checks
    4.  evaluate for semantic selection
    5.  sos_limit check
    6.  subsumption check (forward)
    7.  assign an ID and keep the clause
    8.  unsafe unit conflict check
    9.  check if the clause should be a demodulator
    ---- (the following steps are delayed until finished with the given clause) ---
    10. factor c
    11. apply new_constants to c
```

```
    12. apply back subsume with c
    13. apply back demod with c
    14. apply back unit deletion with c
    15. move c to the sos list
```

Restricted denials (see flag **restrict_denials**) are not subject to the **max_weight** test.

Options for Processing Inferred Clauses

Demodulation Options

Dedmodulation is the process of using equations (demodulators) to rewrite terms. If a demodulator is oriented by the term ordering in effect (KBO, LPO, or RPO), it is applied unconditionally, heavy-to-light. If a demodulator is not oriented, it is applied only if the instance that would be used is oriented.

```
set(lex_order_vars).
clear(lex_order_vars).    % default clear
```

This flag allows an exception to the rule for applying nonorientable demodulators. If the flag is set, variables are treated as constants when comparing terms, with the precedence

```
function_order([x,y,z,u,v,w,v6,v7,v8, ...]).
```

For example, with the (nonorientable) demodulator `x*y = y*x`, the term `v7*v6` can be rewritten to `v6*v7`. Setting this flag can easily block proofs, but it can also drastically reduce the search space and still allow some proofs to be found.

If you have a difficult problem that involves a commutative, associative-commutative, or some other permutative operation, we recommend trying this option.

```
assign(demod_step_limit, n).  % default n=1000, range [-1 .. INT_MAX]
```

This parameter limits the number of rewrite steps that are applied to a clause during demodulation. If *n*=-1, there is no limit.

```
assign(demod_size_limit, n).  % default n=1000, range [-1 .. INT_MAX]
```

This parameter limits the size (measured as symbol count) of terms as they are demodulated. If any term being demodulated has more than n symbols, demodulation of the clause stops. If *n*=-1, there is no limit.

```
set(back_demod).
clear(back_demod).    % default clear
```

If this flag is set, back demodulation is applied. If an orientable equation is derived, it is appended to the `demodulators` list. Non-orientable equations are appended based on the settings of the flags **lex_dep_demod** and **lex_dep_demod_sane** and the parameter **lex_dep_demod_lim**.

If an equation is added to `demodulators`, Then each clause in `usable` or `sos` that can be rewritten with the equation is copied and deleted, then the copy is treated as if it were generated by an inference rule. In particular, it will be processed, including demodulation, which will apply the new demodulator.

```
set(lex_dep_demod).    % default set
clear(lex_dep_demod).
```

If this flag is set, then non-orientable equations can become demodulators (via the flag **`back_demod`**).

```
assign(lex_dep_demod_lim, n).  % default n=11, range [-1 .. INT_MAX]
```

This parameter is a limit on the flag **`lex_dep_demod`**. A non-orientable equation cannot become a demodulator if it has more than *n* symbols. (The equation `(x*y)*z=x*(y*z)` has 11 symbols.) If *n* = -1, there is no limit.

```
set(lex_dep_demod_sane).    % default set
clear(lex_dep_demod_sane).
```

This flag is a restriction on the flag **`lex_dep_demod`**. If set, a non-orientable equation can become a demodulator only if its two sides have the same number of symbols.

```
set(unit_deletion).
clear(unit_deletion).    % default clear
```

This flag extends demodulation to include rewriting of literals with unit clauses. For example, if we have the unit clause `p(x,a)`, then we can use it to remove instances of `-p(x,a)` from generated clauses. This process is like using the unit clause as the demodulator `p(x,a) = TRUE`. (Unit deletion is not actually implemented as demodulation.)

```
set(back_unit_deletion).
clear(back_unit_deletion).    % default clear
```

This flag is analogous to back demodulation. If set, then each time a unit clause is kept, it is used to apply unit deletion to all clauses in `sos` and `usable` in the same way that `back_demodulation` works.

Simplifying and Deciding Whether to Keep Clauses

The options in this section appear in the order in which they are applied.

```
set(cac_redundancy).    % default set
clear(cac_redundancy).
```

If this flag is set, then an equational redundancy criterion is applied. If Prover9 finds that a binary operation is commutative or associative-commutative, it makes a note and uses that information

to simplify clauses that are derived later in the search.

If a derived clause contains an equality *alpha=beta*, in which *alpha* and *beta* are equal with respect to commutativity or associativity-commutativity of the previously noted operations, the equality is simplified to `TRUE`.

For example, if Prover9 notes that `x*y=y*x`, and then some time later a clause containing the literal `g(u*v)=g(v*u)` is derived, that literal will be simplified to TRUE and the clause will be deleted. (Demodulation will not rewrite the two sides to the same term unless the flag **`lex_dep_demod`** is set.)

```
assign(max_literals, n).  % default n=-1, range [-1 .. INT_MAX]
```

Clauses containing more than *n* literals will be deleted. If = -1, there is no limit. This parameter is never applied to initial clauses or to clauses that match hints.

```
assign(max_depth, n).  % default n=-1, range [-1 .. INT_MAX]
```

If the depth of the clause is more than *n*, it will be deleted. If = -1, there is no limit. This parameter is never applied to initial clauses or to clauses that match hints.

```
assign(max_vars, n).  % default n=-1, range [-1 .. INT_MAX]
```

Clauses containing more than *n* (distinct) variables will be deleted. If = -1, there is no limit. This parameter is never applied to initial clauses or to clauses that match hints.

```
assign(max_weight, n).  % default n=100, range [INT_MIN .. INT_MAX]
```

Derived clauses with weight greater then *n* will be discarded. If = -1, there is no limit. This parameter is never applied to initial clauses or to clauses that match hints.

```
set(safe_unit_conflict).
clear(safe_unit_conflict).    % default clear
```

This flag provides for a safe, but more expensive, unit conflict test. If set, the unit conflict test will be done *before* the **`max_weight`** test is applied. If the flag is clear, the test will be done *after* the **`max_weight`** test is applied, allowing the possibility that a proof will be missed, because the final step was deleted by the **`max_weight`** parameter.

Performing Operations with the New Clause

The options in this section appear in the order in which they are applied.

```
set(factor).
clear(factor).    % default clear
```

If this flag is set, binary factoring is applied to newly-kept clauses. Note that factoring is an inference rule rather than a simplification rule, because a child is generated and the parent is retained. (If the child happens to subsume the parent, the parent will be deleted by the back subsumption process). Unlike other inference rules such as resolution, factoring is applied to a clause when it is *kept*, not when it is *given*.

```
assign(new_constants, n).  % default n=0, range [-1 .. INT_MAX]
```

If this parameter is greater than 0, then Prover9 will apply a rule that introduces a new constant when it derives an equation that shows the existence of a constant. In particular, if a derived equation has the property that each side has variables and the two sides share no variables, a new constant will be introduced and set equal to one side of the equation. (Back demodulation will derive that the constant is equal to the other side.)

For example, if `x' * x = y * y'` is derived, the equation `x' * x = c` is produced, where the constant `c` does not occur anywhere else.

The value of the parameter limits the number of new constants that can be introduced by this rule.

(There is an extension to this rule that introduces (non-constant) function symbols based on the intersection of the variables of the two sides. We have not found the extension to be useful in practice, so we have not included it in Prover9.)

Unlike other inference rules such as resolution, the **`new_constants`** rule is applied to a clause when it is *kept*, not when it is *given*.

```
set(back_subsume).    % default set
clear(back_subsume).
```

If this flag is set, then back subsumption is applied with all new clauses. That is, when a new clause is kept, each clause subsumed by the new clause is deleted.

Next Section: Output Files

Output Files

Even when Prover9 fails to find a proof, its output file usually has lots of valuable information about the search. The output file can suggest many ways of improving the search for subsequent jobs as in the following examples.

- The output shows how equalities are oriented; different term ordering parameters may give better or more intuitive orientations.
- If Prover9 focused the search on uninteresting clauses (see the sequence of given clauses), different inference rules, a different **`pick_given_ratio`**, or a specialized weighting function can be used.
- If Prover9 ran out of time or memory with a huge `sos` list and small `usable` list (i.e., few given clauses were used), the **`sos_limit`** should be reduced.

Basic Structure of Output Files

Prover9 output files are divided into sections and subsections so the users (people and programs) can find what they are looking for. The delimiters are self-explanatory. A few comments about the sections are given here. For a specific example, see the output file subset_trans.out.

```
============================== Prover9 ===============================
    Version, date, host computer, command.
============================== end of head ===========================

============================== INPUT =================================
    Echo of the input.  Everything in this section that is not
    in the input is commented with "%", so copy-and-paste can be
    done on this section to create a new input file.
============================== end of input ==========================

============================== PROCESS GOALS =========================
    The search is always by refutation, and this section shows
    how goals are negated in preparation for the search.
============================== end of process goals ==================

============================== PROCESS INITIAL CLAUSES ===============
    This section shows the starting clauses (after Skomemization,
    if applicable) and then some of what Prover9 does in preparation
    for the search.  This includes predicate_elim, term ordering
    decisions, and auto_inference settings.  At this stage, clauses
    may be deleted by subsumption and equations may be copied to the
    list demodulators.  See the flag process_initial_sos.
============================== end of process initial clauses ========

============================== CLAUSES FOR SEARCH ====================
    This section shows the clauses just before the start of the
    search, that is, just before selection of the first given clause.

============================== end of clauses for search =============
```

```
============================== SEARCH ========================================
    This section typically shows the sequence of given clauses,
    and it may also include PROOF and STATISTICS sections.

============================== PROOF =========================================
    A proof in standard form.
============================== end of proof ==================================

============================== STATISTICS ====================================
    We encourage users to look at statistics!
============================== end of statistics =============================

============================== end of search =================================
```

Clause Justifications

After the initial stage of the output, each clause in the file has an integer identifier (ID) and a justification that may refer to IDs of other clauses. A justification is a list consisting of one primary step and some number of secondary steps. Most primary steps are inference rules applied to given clauses, and most secondary steps consist of simplification, rewriting, or orienting equalities.

Many of the types of step refer to positions of literals or terms in the parent clauses. Literals are identified by the characters 'a' (first literal), 'b' (second literal), etc. Terms are identified by the literal identifier followed by a sequence of integers giving the position of the term within the literal. For example, the position 'c,1,3,2' means third literal, first argument, third argument, second argument. Negation signs on literals are not included in the sequence.

Primary Steps.

- `assumption` -- input formula.
- `clausify` -- from CNF translation of a non-clausal assumption.
- `goal` -- input formula.
- `deny` -- from CNF translation of the negation of a goal.
- `resolve(59,b,47,c)` -- resolve the second literal of clause 59 with the third literal of clause 47.
- `hyper(59, b,47,a, c,38,a)` -- hyperresolution; interpret the list as a clause ID followed by a sequence of triples, <literal,clause-ID,literal> the inference is presented as a sequence of binary resolution steps. In the example shown, start with clause 59; then resolve literal b with clause 47 on literal a; with the result of the first step, resolve literal c with clause 38 on literal a. The special case "`xx`" means resolution with `x=x`.
- `ur(39, a,48,a, b,88,a, c,87,a, d,86,a)` -- unit-resulting resolution; the list is interpreted as in hyperresolution.
- `para(47(a,1),28(a,1,2,2,1))` -- paramodulate from the clause 47 into clause 28 at the positions shown.
- `copy(59)` -- copy clause 59.
- `back_rewite(59)` -- copy clause 59.
- `back_unit_del(59)` -- copy clause 59.
- `new_symbol(59)` -- introduce a new constant (see parameter **`new_constants`**).
- `factor(59,b,c)` -- factor clause 59 by unifying the second and third literals.
- `xx_res(59,b)` -- resolve the second literal of clause 59 with `x=x`.
- `propositional` -- not used in standard proofs.
- `instantiate` -- not used in standard proofs.

- `ivy` -- not used in standard proofs.

Secondary Steps (each assumes a working clause, which is either the result of a primary step or a previous secondary step).

- `rewrite([38(5,R),47(5),59(6,R)])` -- rewriting (demodulation) with equations 38, 47, then 59; the arguments (5), (5), and (6) identify the positions of the rewritten subterms (in an obscure way), and the argument R indicates that the demodulator is used backward (right-to-left).
- `flip(c)` -- the third literal is an equality that has been flipped by the term ordering. This does not necessarily mean that the equality is orientable by the primary term ordering, e.g., KBO.
- `merge(d)` -- the fourth literal has been removed because it was identical to a preceding literal.
- `unit_del(b,38)` -- the second literal has been removed because it was an instance of the negation clause 38 (which is a unit clause).
- `xx(b)` -- the second literal has been removed because it was an instance of `x!=x`.

Standard Proofs

Prover9 proofs may be transformed by separate programs, e.g., by Prooftrans.

Options That Say What Goes To the Output File

```
set(echo_input).    % default set
clear(echo_input).
```

Clearing this flag suppresses printing of clauses, formulas, weighting rules (and everything else that ends with `end_of_list`) that would ordinarily appear in the `INPUT` section of the output file.

```
set(quiet).
clear(quiet).    % default clear
```

Setting this flag causes most messages to the standard error file (usually the user's screen) to be suppressed. These messages include notifications about proofs and statistics reports, and warnings about demodulation limits. Setting this flag also suppresses several messages to the ordinary output file, and it clears the **`bell`** flag.

```
set(print_initial_clauses).    % default set
clear(print_initial_clauses).
```

If this flag is set, clauses are printed in the `PROCESS INITIAL CLAUSES` and `CLAUSES FOR SEARCH` sections of the output file.

```
set(print_given).    % default set
clear(print_given).
```

Clearing this flag prevents given clauses from being printed to the output file.

```
set(print_gen).
clear(print_gen).    % default clear
```

Setting this flag causes all generated clauses to be printed to the the output file. This can be output files to be really huge.

Setting this flag causes the flag **print_kept** also to be set, and clearing this flag causes the flag **print_kept** also to be cleared.

```
set(print_kept).
clear(print_kept).    % default clear
```

Setting this flag causes all kept clauses to be printed to the the output file.

```
set(print_labeled).
clear(print_labeled).    % default clear
```

Setting this flag causes kept clauses containing label attributes to be printed, even when the flag **print_kept** is clear. This flag is useful when using the hints strategy, because when a clause matches a hint containing a label, the label is copied to the clause. That is, clauses matching labeled hints will be printed.

```
set(print_clause_properties).
clear(print_clause_properties).    % default clear
```

Setting this flag causes several properties of clauses to be printed as "props" attributes on the clauses. The properties include which literals are maximal (counting from 1), which literals are maximal among literals of the same sign, and which literals are selected for application of inference rules.

```
set(print_proofs).    % default set
clear(print_proofs).
```

Clearing this flag prevents proofs from being printed to the output file. The proof message still goes to the standard error file (usually the user's screen), unless the flag **quiet** has been set.

```
set(default_output).    % default set
clear(default_output).
```

Setting this flag restores most of the output flags and parameters to their default values. Clearing this flag does nothing.

```
assign(report, n).  % default n=-1, range [-1 .. INT_MAX]
```

If $n > 0$, statistics are sent to the output file approximately every n seconds. (On Unix-like systems, one can also tell Prover9 to print statistics to the output file by sending the signal `USR1` to a running Prover9 process, e.g., `kill -USR1 4223`.)

```
assign(stats, string).  % default string=lots, range [none,some,lots,all]
```

This parameter determines how many statistics are sent to the output file.

```
set(clocks).
clear(clocks).    % default clear
```

If this flag is set, various operations during the Prover9 job are timed (e.g., inference, demodulation, and subsumption), and timing reports are sent to the output file.

Timing the operations can be expensive, especially in Solaris and Macintosh systems. On Linux systems, `set(clocks)` typically adds 5% -- 10% to the run time.

```
set(bell).    % default set
clear(bell).
```

If this flag is set, Prover9 beeps when important things happen, such as proofs and warnings. Some users run searches that find hundreds of proofs, and they clear this flag to prevent all of the beeping.

Next Section: Weighting

Weighting

Prover9's weighting function maps clauses to integers, and it is used primarily for two purposes:

- selecting the given clause, and
- discarding inferred clauses (with the parameter **max_weight**).

Otter accepts two weighting functions, one for selecting the given clause, and the other for discarding inferred clauses. Prover9 always uses the same weighting function for both purposes.

In Otter's weighting rules, a variable matches any variable and only variables. The role is similar to the anonymous variables "_" in Prover9's weighting rules.

Prover9 does not (yet) have anything analogous to Otter's `$DOTS` weighting feature.

Default Weights

The default weight of a clause is its symbol count, excluding commas, parentheses, negation symbols, and disjunction symbols. That is,

- the default weight of a constant or variable is 1,
- the default weight of a term or atomic formula is one more than the sum of the weights of its arguments,
- the default weight of a literal is the weight of its atomic formula,
- the default weight of a clause is the sum of the weights of its literals.

Weighting Rules

The weighting function can be modified by giving a list of rules in the input file. The list must start with `list(weights).` and end with `end_of_list`. Here is an example.

```
list(weights).

  weight(a) = 3.                                % the weight of the constant a is 3
  weight(f(a,x)) = 5 * weight(x).               % weight( f(a,term) ) = 5 * weight( term )
  weight(f(a,_)) = -1.                          % _ matches any variable
  weight(x | y) = 2 + (weight(x) + weight(y)). % add 2 for each "or" symbol

end_of_list.
```

Here is a summary of the weighting language.

- Each weighting rule is an equation. The left-hand side of the rule must be `weight(`*`pattern`*`)`. A rule applies to a term if its pattern matches the term in the ordinary sense of demodulation or term rewriting.

An exception is that the symbol "_" matches any variable and only a variable.

- The right-hand side of a rule consists of an integer-arithmetic expression applied to `weight(...)` terms. When applying a rule, the substitution of the pattern match is applied to the the `weight(...)` terms, which are then weighed recursively, and then the integer expression is evaluated to compute the weight of the term.
- The accepted integer operations are
  - ♦ binary: {`+, *, /, min, max`}
  - ♦ unary: {`-, depth`}
- The rules are parsed with the ordinary term-parsing code, so (unless the user as included an `op` command to change the parsing rules), the arithmetic expressions must be fully parenthesized, e.g., `a + (b + c)`.

Weighting rules are applied to a clause as follows.

- The clause is weighed top-down. That is, a term is weighed before its subterms are weighed.
- When weighing a term, the first rule that matches is applied.
- If no rule matches, the weight of the term is one more than the sum of the weights of its arguments.

Modifying the Default Weight

`assign(constant_weight, n).  % default n=1, range [INT_MIN .. INT_MAX]`

This parameter specifies the default weight of constants. It can be overridden with weighting rules for individual constants.

`assign(sk_constant_weight, n).  % default n=1, range [INT_MIN .. INT_MAX]`

This parameter specifies the default weight of Skolem constants. It takes precedence over **`constant_weight`**.

`assign(variable_weight, n).  % default n=1, range [INT_MIN .. INT_MAX]`

This parameter specifies the default weight of variables.

`assign(not_weight, n).  % default n=0, range [INT_MIN .. INT_MAX]`

The negation symbols on literals do not ordinarily contribute any weight to clauses. This parameter says that each negation symbol has weight *n*.

`assign(or_weight, n).  % default n=0, range [INT_MIN .. INT_MAX]`

The disjunction symbols between literals do not ordinarily contribute any weight to clauses. This parameter says that each disjunction symbol has weight *n*.

`assign(prop_atom_weight, n).  % default n=1, range [INT_MIN .. INT_MAX]`

This parameter specifies the default weight for propositional atoms, that is, predicate symbols of arity 0. They ordinarily have weight 1.

`assign(nest_penalty, n).  % default n=0, range [0 .. INT_MAX]`

This parameter is used to penalize terms containing nested function symbols. If no weighting rule applies to a term *t*, then for each argument with the same function symbol as *t*, the value *n* is added to the weight of *t*. If *n*=0, there is no penalty.

`assign(skolem_penalty, n).  % default n=1, range [0 .. INT_MAX]`

This parameter is used to penalize terms containing non-constant Skolem function. If no weighting rule applies to a term *t*, then for each argument that contains a non-constant Skolem function, its weight is multiplied by *n*. If *n*=1, there is no penalty.

`assign(depth_penalty, n).  % default n=0, range [INT_MIN .. INT_MAX]`

This parameter is used to penalize (or prefer) clauses with deeper terms. It is applied to the entire clause after all of the literals and subterms have been weighed. The weight of the clause *C* is increased by *n* * *depth(C)*. Note that *n* may be negative, decreasing the weight of the clause.

`assign(var_penalty, n).  % default n=0, range [INT_MIN .. INT_MAX]`

This parameter is used to penalize (or prefer) clauses with more variables. It is applied to the entire clause after all of the literals and subterms have been weighed. If *v* is the number of (distinct) variable in the clause, the weight of the clause is increased by *n* * *v*. Note that *n* may be negative, decreasing the weight of the clause.

Adjustments to Clause Weight

The final weight of a clause is calculated in three steps. First, the weighting rules are applied. Second, if the weight is between **default_weight** and **max_weight**, the weight is reset to **default_weight**.

`assign(default_weight, n).  % default n=INT_MAX, range [INT_MIN .. INT_MAX]`

That is, all clauses with weight between **default_weight** and **max_weight** are treated equally.

Third, if the clause matches a hint, the weight may be adjusted by the flag **degrade_hints** and by the hint attribute `bsub_hint_wt`.

Debugging Weighting Rules and Options

Here is an example of using Prover9 to test weighting rules and parameters.

```
prover9 -f weight_test.in | grep 'given #' > weight_test.out
```

Next Section: Attributes

Attributes

Several kinds of attribute can be attached to input formulas with the # operator, for example,

```
x * y = y * x              # label(commutativity).
x * c != e                 # answer(x) # label("the denial").
-p(c) | -q(c)              # answer("here it is").
a * b != b * a             # action(in_proof -> exit) # answer(commutativity).
x * (y * z) = y * (x * z)  # bsub_hint_wt(500).
```

Each attribute has a data type of string, integer, or term. A string attribute is really just a term attribute that is a constant. If a string attribute is not a legal constant, it can be enclosed in double quotes to make it so.

Attributes can be attached only to the top level of a formula; they cannot be attached to proper subformulas. (This restriction might be lifted in future versions of Prover9.)

The accepted attributes are shown in the following table.

Name	Type	Inheritable	Purpose
label	string	No	Comment
answer	term	Yes	Record substitutions and what has been proved
action	term	No	Triggers action when clause is used
bsub_hint_wt	integer	No	Used for hints

Inheritable attributes are passed from parent to child during most inference rules.

Label Attributes

Label attributes are simply comments that can be attached to input clauses, including hint clauses.

Answer Attributes

Answer attributes on clauses are essentially answer literals. They are inherited during application of inference rules, and if they contain variables, the variables are instantiated by the substitution used in the inference.

Answer attributes (like all other attributes) contain exactly one argument. If you wish to record substitutions for more than one variable, you must use a term that contains all of the variables, for example, a list, as in the following clause.

```
-p(c,x,y,z)  # answer([x,y,z]).
```

Answer attributes need not contain variables. For example, when there are multiple goals, answer attributes can be used on the goal formulas to identify the goals that are proved.

Answer attributes on non-clausal formulas cannot contain variables. (This restriction might be lifted in future versions of Prover9.)

Action Attributes

Action attributes cause various things to happen when clauses are used in various ways. See the section on Actions.

Bsub_hint_wt Attribute

This attribute can be attached to a hint clause, and it is used to override ordinary weight assigned to clauses that match the hint. That is, if a hint matches a clause, and if the hint has a **`bsub_hint_wt`** attribute, the clause gets the value of the attribute as its weight instead of the weight that would be assigned by the ordinary weighting method.

Next Section: Actions

Actions

Prover9's actions allow the user to change the search strategy during the search. For example, after a certain number of given clauses have been used, the **max_weight** can be changed.

Actions can be triggered in two ways:

- by some counter, for example, after 100 clauses have been retained, and
- when a clause containing an action attribute is used, for example, when it is used in a proof.

Accepted Actions

The currently accepted actions are `exit` (which terminates the search) and a subset of the ordinary flags and parameters.

- Changable flags: **reuse_denials**, **print_gen**, **print_kept**, **print_given**, **breadth_first**. **lightest_first**. **breadth_first_hints**.
- Changable parameters: **demod_step_limit**, **demod_size_limit**, **new_constants**, **para_lit_limit**, **stats**, **max_given**, **max_weight**, **max_depth**, **max_vars**, **max_proofs**, **max_literals**, **constant_weight**, **variable_weight**, **not_weight**, **or_weight**, **prop_atom_weight**, **skolem_penalty**, **nest_penalty**, **depth_penalty**, **default_weight**, **pick_given_ratio**, **age_part**, **weight_part**, **hints_part**, **false_part**, **true_part**.

Actions Triggered by Counters

Counter actions are given as a list of rules *trigger -> action* in the input file. Here are the currently recognized triggers for counter actions.

- given: the number of given clauses that have been processed.
- generated: the number of clauses that have been inferred.
- kept: the number of clauses that have been inferred and retained.
- level: the search level (this applies to breadth-first searches).

The list must start with `list(actions).` and end with `end_of_list`.

Here is an example list of counter actions.

```
list(actions).

  given=10        -> set(print_kept).
  kept=1000       -> assign(max_weight, 30).
```

```
  generated=5000  -> assign(pick_given_ratio, 4).
  level=3         -> exit.

end_of_list.
```

Actions Triggered by Clauses

Clause actions occur as attributes on clauses, for example,

```
A * B != B * A  # action(in_proof -> assign(max_weight, 30)).
```

In this example (which only makes sense if **`max_proofs`** > 1), if the clause occurs in a proof, the action is applied.

The only trigger currently recognized for clause actions is `in_proof`. Others will likely be added.

Next Section: Goals and Denials

Goals and Denials

This section shows how the conclusion(s) of a conjecture can be stated in positive form, how one can search for direct proofs as opposed to bidirectional proofs, and how multiple conclusions are stated and handled.

Terminology

- *Conclusion*: this term is used informally.
- *Goal*: this term refers to a conclusion stated in positive form.
- *Denial clause*: this term refers to a negative clause in a Horn set, because such clauses usually correspond to the negation of a conclusion.

Goals: Stating Conclusions in Positive Form

> In Otter, the conclusions are always stated in negated form.

Prover9 allows the user to state conclusions in positive form by using the list `formulas(goals)`. However, Prover9 always works by refutation, so the clauses or formulas in the `goals` lists are negated as described below, and the results are appended to the `sos` clause list before the search starts. In other words, goals are "syntactic sugar" for input, and have nothing to do with the way Prover9 conducts its search for refutations.

When the conclusion is given in positive form, the user has no control over the Skolem symbols (if any) that Prover9 introduces. If the user needs some control of the Skolem symbols, for example, to insert them into the symbol precedence at a particular spot, or to include them in the weighting function, the user should do the Skolemizing and give the conclusion in negated form.

If there is just one formula in `formulas(goals)`, the meaning is clear: the formula is processed by first taking its universal closure, then negating. The formula is then handled exactly as if it had been input in `formulas(sos)`, that is, by Skolemizing and transforming to clauses.

Multiple Goals

If there is more than one formula in `formulas(goals)`, the meaning is not clear. Is the conclusion the disjunction of those formulas? Or the conjunction? *The answer: disjunction*: if any goal is proved, the proof is reported, printed, and counted.

Multiple *complex* goals are not allowed, because the quantification of free variables can be very confusing. Therefore Prover9 enforces the following rule.

> *If there is more than one formula in the goals list, each must be a positive universal conjunctive formula, that is a formula constructed from atomic formulas, universal quantification, and conjunction only.*

To avoid this restriction, one can always write the conclusion clearly as a single goal formula containing any of the logic connectives and quantification. However, if the conjecture involves multiple complex conclusions, we recommend, for search efficiency, separate Prover9 searches.

If there are multiple goals, each is processed separately by applying universal closure, negation, and transformation to clauses. After this processing, Prover9 forgets that there were multiple goals and simply searches for refutations.

When there are multiple goals, and when the user wishes to prove more than one goal, the parameter **max_proofs** should be set to an appropriate value. (The flag **auto_denials** (default set) can do so automatically.)

Multiple Proofs

```
assign(max_proofs, n).  % default n=1, range [-1 .. INT_MAX]
```

This parameter tells Prover9 to stop searching when the *n*-th proof has been found.

Denials: Negative Clauses in Horn Sets

Denial clauses (negative clauses in Horn sets) can be derived from goals, or they can be input directly as negative clauses.

Forward or Direct Proofs

The following flag restricts the use of negative clauses in Horn sets, with the aim of finding proofs that are more direct, that is, proofs that go forward from the hypotheses to the conclusion rather than proofs that reason backward from the conclusion.

```
set(restrict_denials).
clear(restrict_denials).    % default clear
```

This flag applies only to Horn sets. If the flag is set, negative clauses (clauses in which all literals are negative) in Horn sets are referred to as *restricted denials* and are given special treatment.

The inference rules (i.e., paramodulation and the resolution rules) will not be applied to restricted denials. However, restricted denials will be simplified by back demodulation and back unit deletion.

In addition, restricted denials will not be deleted if they are over the weight limit (**max_weight**).

The effect of setting **restrict_denials** is that proofs of Horn sets will usually be more forward or direct. This option can speed up proofs, it can delay proofs, and it can block all proofs.

Multiple Proofs of the Same Conclusion

```
set(reuse_denials).
```

```
clear(reuse_denials).    % default clear
```

If this flag is set, when a denial clause (a negative clause in a Horn set) is used in a proof, and when **max_proofs** says to search for more proofs, subsequent proofs may be of the same conclusion. (Multiple proofs of the same conclusion may be useful when one is searching for *short* proofs.)

If this flag is clear, then when a proof is found, the denial and all of its descendants are disabled so that they will not appear in subsequent proofs.

This flag is independent of the flag **restrict_denials**.

Auto_denials

```
set(auto_denials).    % default set
clear(auto_denials).
```

If this flag is set (the default), negative clauses in Horn sets receive some special initial processing.

If a Horn set has more than one denial (negative) clause, we assume they correspond to separate conclusions, and the user wishes to have a separate proof of each conclusion. Therefore, if **max_proofs** has not been changed from its default value of 1, we assign to **max_proofs** the number of negative clauses. (Note that when **reuse_denials** is clear (the default), Prover9 prevents multiple proofs of the same conclusion.)

Also, if a negative clause in a Horn set has label attribute but no answer attribute, the clause is given an answer attribute corresponding to the first label attribute. This saves the user from changing "label" to "answer" when moving formulas from the `sos` list to the `goals` list.

An Example

The following example illustrates multiple goals (including a goal that is a combination of other goals), **auto_denials**, and **restrict_denials**.

```
prover9 -f olsax.in > olsax.out
```

Next Section: Hints

Hints

Hint clauses can be used to help guide Prover9's search. Prover9's input can contain any number of hint lists (which are simply concatenated by Prover9).

Each list of hint clauses must start with `formulas(hints).` and end with `end_of_list`. Any clause is acceptable as a hint. For example (the label attributes are optional),

```
formulas(hints).
    x ' * (x * y) = y       # label(6).
    x * (x * y) = y         # label(7).
    x * (y * (x * y)) = e   # label(8).
    x ' ' * e = x           # label(9).
    x ' * e = x             # label(10).
    x ' = x                 # label(11).
    x * e = x               # label(12).
    x * (y * x) = y         # label(13).
    x * y = y * x           # label(14).
end_of_list.
```

A derived clause *matches* a hint if it subsumes the hint. If a clause matches more than one hint, the first matching hint is used.

> In Otter, "matching a hint" can mean (depending on the parameter settings) subsumes, subsumed by, or equivalent to. These other types of matching may be added to Prover9 if there is any demand for them.

Hints are used primarily when selecting given clauses. The mechanism for doing this is the given-clause selection procedure. In short, the default value of the **`hints_part`** parameter says to select clauses that match hints (lightest first) whenever any are available.

Hints are also used when deciding to keep a new clause. Clauses that match hints are not deleted by any of the parameters **`max_weight`**, **`max_vars`**, **`max_literals`**, or `max_depth`.

Where do Hints Come From?

Hints frequently consist of proofs, perhaps many, of related theorems.

Bob Veroff developed the concept, installing code for hints in an early version of Otter, to experiment with his method of *proof sketches* [Veroff-hints, Veroff-sketches]. In the proof sketches method, a difficult conjecture is attacked by first proving several (or many) weakened variants of the conjecture, and using those proofs as hints to guide searches for a proof of the original conjecture.

The program Prooftrans, which is distributed along with Prover9, can be used to extract proofs from a Prover9 output file and transform the proofs to lists of hints suitable for input to subsequent Prover9 jobs.

An Example

This example consists of four jobs. The first is a proof of a nontrivial theorem called "hard". The other three jobs prove the hard theorem indirectly by first proving an easier theorem (in this case, the easier theorem simply the harder theorem with an extra assumption); then using the proof of the easier theorem as hints to help prove the hard theorem.

1. A Prover9 job that proves the hard theorem.

   ```
   prover9 -f hard.in > hard.out
   ```
2. A proof of the easier thorem.

   ```
   prover9 -f easy.in > easy.out
   ```
3. A Prooftrans job converts the proof of the easier theorem into a list of hints.

   ```
   prooftrans hints -f easy.out > easy.hints
   ```
4. A Prover9 job that uses the hints to prove the harder theorem.

   ```
   prover9 -f hard.in easy.hints  > hard-hints.out
   ```

Proving the hard theorem indirectly (jobs 2,3,4) takes about 1/4 the time as proving it directly (job 1). Of course the difficult and interesting part of working this way is finding good "easy" theorems.

Special Weight Assignments

When the given clause selection procedure calls for a clause that matches a hint, the lightest such clause is chosen. Ordinarily, clauses that match hints are weighed just as any other clause is weighed. However, if one believes some hints are more important that others, one can, in effect, say "any clause that matches this hint gets a specific weight". This is accomplished by attaching a `bsub_hint_wt` attribute to the hint, as in the following example.

```
formulas(hints).
  x ' * (x * y) = y     # label("very important hint") # bsub_hint_wt(-100).
end_of_list.
```

Another way to assign a special weight is with the following flag.

```
set(breadth_first_hints).
clear(breadth_first_hints).    % default clear
```

> Setting this flag causes all clauses that match hints to receive weight 0. The effect is as if each hint had the attribute `bsub_hint_wt(0)`. This causes clauses that match hints to be selected in the order they are generated.

The weight assigned by any of the preceding methods may be modified if the flag **`degrade_hints`** is set.

Hint Degradation

In many searches that use hints, a given hint can match many different derived clauses. As a hint matches more and more clauses, we wish its influence to diminish. This is the idea behind Veroff's *hint degradation* method.

```
set(degrade_hints).    % default set
clear(degrade_hints).
```

> If this flag is set, a weight penalty is added to clauses that match hints that have been previously matched. The following procedure is used. Given a newly derived clause, say *C*, assume we find a hint that matches the clause; let *n* be the number of times the hint has already been matched; then the weight of *C* is increased by (*n* * 1000). In other words, 1000 is added for each previous match of the hint.
>
> The effect of this procedure is (usually) that clauses matching hints are selected in the following order: clauses matching hints that have *not* been matched before, clauses matching hints that have been matched *once* before, and so on.

Keeping/Limiting Clauses the Match Hints

Ordinarily, when a clause matching a hint is derived, the clause will be retained even if it violates limits such as `max_weight`. Setting the following flag will cause those limits to be applied to such clauses, and it may be useful with trying to simplify known proofs.

```
set(limit_hint_matchers).
clear(limit_hint_matchers).    % default clear
```

> If this flag is set, the parameters `max_weight`, `max_literals`, `max_depth`, and `max_vars` will be applied to clauses that match hints (as well as to clauses that don't match hints).
>
> Otherwise (the default), those limits will not be applied to clauses that match hints.

Back Demodulation of Hints

When hints come from proofs in which equality and rewriting play a major role, they may have trouble guiding a search, because the rewriting may occur in different ways in the new search. In particular, a hint may fail to match a clause, because the clause has been rewritten and the hint has not. This is the motivation for the following feature.

```
set(back_demod_hints).    % default set
clear(back_demod_hints).
```

> If this flag is set, hints are back demodulated. That is, they are kept simplified with respect to the current set of demodulators.

Labels on Hints

Label attributes on hint clauses get special treatment. When a hint containing a label matches a clause, the label attribute is copied to the clause.

The following flag addresses the situation in which the input contains sets of equivalent hints. (This situation frequently occurs when the hints contain many proofs of similar theorems.)

```
set(collect_hint_labels).
clear(collect_hint_labels).    % default clear
```

If this flag is set, and the hints list contains a set of equivalent hints, only the first copy of the hint is retained. However, the labels from all of the other equivalent hints are collected and put on the retained copy. When a clause matches the retained hint, it gets copies of all of the labels from the equivalent hints.

If this flag is clear, when a clause matches a set of equivalent hints, it receives the label (if any) from the first copy only.

Next Section: Semantics

Semantic Guidance

Prover9 has a method of using finite interpretations to guide the search for a proof; in particular, to help select the given clause.

To use semantic guidance the user gives one or more interpretations along with the ordinary Prover9 input. All clauses (input and derived) that are eligible to be selected as given clauses are evaluated in the interpretations. If a clause is false in all of the interpretations, it is marked as "false" and given the attribute `label(false)`; if it is true in any of the interpretations, it is marked as "true". (There is an exception: see the parameter **`eval_limit`** below.)

If a clause being evaluated contains a symbol that is not in an interpretation, a warning message is given, and the clause receives the value "true".

When selecting the given clause, Prover9 may use the parameters **`true_part`**,and **`false_part`**, as described on the page Selecting the Given Clause. With semantic guidance (explicit interpretations), the "true_part" and "false_part" refer simply to clauses marked as "true" and "false" with respect to the interpretations.

Format of Interpretations for Semantic Guidance

The interpretations are finite and must be in the format produced by Mace4. They must appear in a list that starts with `list(interpretations).` and ends with `and_of_list.` The following example is a lattice in terms of the meet and join operations.

```
list(interpretations).
interpretation(6, [], [
    function(^(_,_), [
        0,0,0,0,0,0,
        0,1,2,3,4,5,
        0,2,2,0,0,0,
        0,3,0,3,5,5,
        0,4,0,5,4,5,
        0,5,0,5,5,5]),
    function(v(_,_), [
        0,1,2,3,4,5,
        1,1,1,1,1,1,
        2,1,2,1,1,1,
        3,1,1,3,1,3,
        4,1,1,1,4,4,
        5,1,1,3,4,5])]).
end_of_list.
```

An Example of Semantic Guidance

Here a job that uses the preceding interpretation for semantic guidance.

```
prover9 -f LT-82-2.in > LT-82-2.out
```

Notes about the preceding job.

- The interpretation is the only additional input needed to give semantic guidance. The default values of the parameters **age_part**, **true_part**, **false_part**, and **eval_limit**, work well for this job (and many others).
- The interpretation does not contain Skolem constants that appear in the denial, and warning messages are given when Prover9 attempts to evaluate clauses containing those Skolem constants. (They receive the value "true".)
- One of the input clauses is assigned the attribute `label(false)`, because it is false in the interpretation.
- The "false" given clauses (#12, #13, #17, #18, #22, #23, ...) are mostly heavier than the "true" given clauses, showing that they would likely not enter the search at such an early stage without semantic guidance.
- At given clause #138, there are no more false clauses to select; and then several more are inferred and given near the end of the search, leading to a proof.
- This job takes about 11 seconds. A similar job without semantic guidance takes about 25 minutes to find a proof.

Advice on Selecting Interpretations

If the conjecture formulates naturally as

theory, hypotheses -> conclusion,

a good first step is to try the smallest model of the theory in which the conclusion is false. The preceding example has that form, and the interpretation used in the that example can be easily found with the following Mace4 job.

```
mace4 -N10 -f LT-82-2-interp.in > LT-82-2-interp.out
```

If the conjecture formulates naturally as

theory -> conclusion,

with no obvious hypothesis, one can try to slightly weaken the theory in some way that relates to the conclusion, and use a model of the weakened theory in which the conclusion is false.

Options for Semantic Guidance

Aside from the parameters **true_part**,and **false_part**, which may be used regardless of whether semantic guidance is in effect, there is just one option, **eval_limit**, to control semantic guidance.

If an interpretation is large, or if a clause being evaluated has many variables, evaluation can take too long, because it must consider each instance of the clause over the domain of the interpretation. That is if an

interpretation has size *d*, and a clause has *v* variables, evaluation has to consider d^v instances of the clause to determine that it is true. The following parameter causes large evaluations to be skipped.

```
assign(eval_limit, n).  % default n=1024, range [-1 .. INT_MAX]
```

This parameter applies when explicit interpretations are being used to select the given clause. When a clause is being evaluated in an interpretation, if the number of ground instances that would be considered is greater than *n*, the evaluation is skipped and the clause is assigned the value *true*.

The default value of 1024 allows

- clauses with up to 3 variables to be evaluated in interpretations up to size 10,
- clauses with up to 4 variables to be evaluated in interpretations up to size 5,
- clauses with up to 5 variables to be evaluated in interpretations up to size 4,
- clauses with up to 6 variables to be evaluated in interpretations up to size 3, and
- clauses with up to 10 variables to be evaluated in interpretations of size 2.

Next Section: Mace4

Mace4 (Models And CounterExamples)

The program Mace4 [McCune-Mace4] searches for finite structures satisfying first-order and equational statements, the same kind of statement that Prover9 accepts. If the statement is the denial of some conjecture, any structures found by Mace4 are counterexamples to the conjecture.

Mace4 can be a valuable complement to Prover9, looking for counterexamples before (or at the same time as) using Prover9 to search for a proof. It can also be used to help debug input clauses and formulas for Prover9.

For the most part, Mace4 accepts the same input files as Prover9. If the input file contains commands that Mace4 does not understand, then the argument "-c" must be given to tell Mace4 to ignore those commands.

For example, say we're learning group theory, and we're wondering whether all groups are commutative. We can run the following two jobs in parallel, with Prover9 looking for a proof, and Mace4 looking for a counterexample.

```
prover9  -f x2.in > x2.prover9.out
mace4 -c -f x2.in > x2.mace4.out
```

Most of the options accepted by Mace4 can be given either on the command line or in the input file. The following command lists the command-line options accepted by Mace4.

```
mace4 -help
```

Terminology. We use the terms *interpretation*, *model*, and *structure* for the objects that Mace4 produces. From a logic point of view, Mace4 produces interpretations which are models of the input formulas. From a math point of view, Mace4 produces structures satisfying the input formulas.

What Mace4 Does

Mace4 searches for *unsorted finite structures* only. That is, a structure (model) has one underlying finite set, called the *domain* (the members are always 0,1,...,*n-1* for a set of size *n*), and structures are functions and relations (tables) over the domain, corresponding to the operations and relation symbols in the specification.

By default, Mace4 starts searching for a structure of domain size 2, and then it increments the size until it succeeds or reaches some limit.

The Original Mace4 Manual

The original Mace4 manual [McCune-Mace4] (PDF) is out of date with respect to features and options, but it contains useful information on the history of Mace4, details on the search methods, and the differences between Mace2 and Mace4.

Next Section: Mace4 Input

Mace4 Input

Mace4 has been designed so that it accepts most Prover9 input files. This allows users to prepare one input file which can be used by Prover9 (to search for proofs) and by Mace4 (to search for counterexamples).

Mace4 Options

Mace4 and Prover9 accept different sets of flags and parameters. In order to use the same input files for both programs, we let Mace4 take its options from the command line instead of from the input file. If Mace4 is given a Prover9 input file, along with the command-line option `-c`, it will ignore any unrecognized (e.g., Prover9) options in the input file. The Mace4 options are described on the next page.

Formulas (including Clauses)

Mace4 accepts the same formulas and clauses as Prover9. See the page Prover9 Clauses and Formulas.

A Caveat: Domain Elements

In one important case, formulas have different meanings in Prover9 and Mace4:

If a formula contains constants that are natural-numbers, {0,1,...}, Mace4 assumes they are members of the domain of some structure, that is, they are distinct objects; in effect, Mace4 operates under the assumptions $0 \neq 1$, $0 \neq 2$, ...

To Prover9, natural numbers are just ordinary constants. For example, to Prover9 the statement 0=1 is satisfiable, and to Mace4 it is unsatisfiable.

Because Mace4 assumes that natural-number constants are members of the domain, if a formula contains a natural number that is out of range ($\geq n$, when searching for a structure of size n), Mace4 will terminate with a fatal error.

Lists of Formulas (including Clauses)

Prover9 accepts a fixed set of lists of formulas (e.g., `assumptions`, `usable`, `goals`, `hints`).

Mace4 accepts any lists of formulas. All are treated as ordinary formulas *except the following two lists*.

- `formulas(hints)`. These are intended to help Prover9 find proofs and are ignored by Mace4.
- `formulas(goals)`. These are negated by Mace4, just as they are by Prover9.

formulas(goals)

Prover9 has several restrictions on the goals it accepts (see Prover9 Goals and Denials), and Mace4 has the same restrictions. Mace4 negates goals and translates them to clauses in the same way as Prover9. (The term "goal" is not particularly intuitive for Mace4 users, because Mace4 does not prove things. It makes more sense, however, when one thinks of Mace4 as searching for a counterexample to the goal.)

When there are multiple goals, Mace handles them the same as Prover9. For example, consider the following goals.

```
formulas(goals).
  x * y = y * x              # label(commutativity).
  (x * y) * z = x * (y * z)  # label(associativity).
end_of_list.
```

Logically, this is a disjunction: Prover9 gives a proof if either goal is proved, and Mace4 gives a counterexample if both are falsified. In particular, this pair of goals is equivalent (for both Prover9 and Mace4) to the following pair of assumptions.

```
formulas(assumptions).
  exists x exists y (x * y != y * x).
  exists x exists y exists z (x * y) * z != x * (y * z).
end_of_list.
```

Next Section: Mace4 Options

Mace4 Options

Mace4 accepts `set`, `clear`, and `assign` commands in the input file. Several of these are in common with Prover9 (e.g., `assign(max_seconds, 30)`), but most are specifically for Mace4.

If Mace4 is called with the command-line option `-c` (compatability mode), it will ignore any `set`, `clear`, and `assign` that it does not recognize, assuming they are meant for some other program (Prover9).

Each Mace4 option can be specified on the command line instead of in the input file. In practice, Mace4 options are *usually* specified on the command line, so that the input files can be used also for Prover9.

When Mace4 options are specified on the command line, single-character codes are used. For example, the command-line option `-t 30` means the same as `assign(max_second, 30)` in the input file. If an option is given in *both* places, the one on the command line takes precedence. Command-line options for Boolean-valued options (flags) always take an argument: 1 means "set", and 0 means "clear". For example, `-V 1` means `set(prolog_style_vaiables`, and `-V 0` means `clear(prolog_style_variables)`.

The command "`mace4 -help`" shows the correspondence between the command-line codes and the option names, and it shows the default values.

Option Listing

Basic Options

```
assign(domain_size, n).  % default n=2, range [2 .. 200]  % command-line -n n

assign(iterate_up_to, n).  % default n=10, range [-1 .. 200]  % command-line -N n

assign(increment, n).  % default n=1, range [1 .. INT_MAX]  % command-line -i n
```

These three parameter work together to determine the domain sizes to be searched. The search starts for structures of size `domain_size`; if that search fails, the size is incremented, and another search starts. This continues up through the value `iterate_up_to` (or until some other limit terminates the process).

For example, the command-line options "`-n 5 -N 11 -i 2`" say to try domain sizes 5,7,9,11.

```
set(iterate_primes).                      % command-line -q 1
clear(iterate_primes).    % default clear  % command-line -q 0
```

This flag overrides the parameter `iterate`. It says to try the sizes that are prime numbers, from `domain_size` up through `iterate_up_to`.

For example, the command-line options "`-n 10 -N 30 -q 1`" say to try domain sizes 11, 13, 17, 19, 23, 29.

```
assign(max_models, n).  % default n=1, range [-1 .. INT_MAX]  % command-line -m n
```

The parameter max_models says to stop searching when the *n*-th structure has been found.

```
assign(max_seconds, n).  % default n=INT_MAX, range [0 .. INT_MAX]  % command-line -t n
```

The parameter **max_seconds** says to stop searching after *n* seconds.

```
assign(max_seconds_per, n).  % default n=INT_MAX, range [0 .. INT_MAX]  % command-line -s n
```

The parameter allows at most *n* seconds for each domain size. The parameter **max_seconds** can be used (together with max_seconds_per) to given an overall time limit.

```
assign(max_megs, n).  % default n=200, range [0 .. INT_MAX]  % command-line -b n
```

The parameter **max_megs** says to stop searching when (about) *n* megabytes of memory have been used.

```
set(prolog_style_variables).                        % command-line -V 1
clear(prolog_style_variables).    % default clear  % command-line -V 0
```

A rule is needed for distinguishing variables from constants in clauses and formulas with free variables. If this flag is clear, variables start with (lower case) 'u' through 'z'. If this flag is set, variables in clauses start with (upper case) 'A' through 'Z' or '_'.

```
set(print_models_standard).      % default set    % command-line -P 1
clear(print_models_standard).                    % command-line -P 0
```

If this flag is set, all structures that are found are printed in "standard" form, which means they are suitable as input to other LADR programs such as isofilter and interpformat.

```
set(print_models).                        % command-line -p 1
clear(print_models).    % default clear  % command-line -p 0
```

If this flag is set, and if print_models_standard is clear, all structures that are found are printed in a tabular form.

```
set(integer_ring).                        % command-line -R 1
clear(integer_ring).    % default clear  % command-line -R 0
```

If this flag is set, a ring structure is is applied to the search. The operations {+,-,*} are assumed to be the ring of integers (mod domain_size). This method puts a tight constraint on the search, allowing much larger structures to be investigated. Here is an example.

```
mace4 -f ring41.in > ring41.out
```

For further information on the integer_ring flag, see slides from a workshop presentation.

```
set(verbose).                        % command-line -v 1
clear(verbose).    % default clear  % command-line -v 0
```

If the verbose flag is set, the output file receives information about the search, including the initial partial model (the part of the model that can be determined before backtracking starts) and timing and other statistics for each domain size. (It does not give a trace of the backtracking, so it does not consume a lot of file space.)

```
set(trace).                         % command-line -T 1
clear(trace).    % default clear     % command-line -T 0
```

If the `trace` flag is set, detailed information about the search, including a trace of all assignments and backtracking, is printed to the standard output. *This flag causes a lot of output, so it should be used only on small searches.*

Advanced Options

These options are used for experimentation with search methods. They can be ignored by nearly all users. For descriptions of most of these options, see the original Mace4 manual [McCune-Mace4] (PDF).

```
set(lnh).      % default set     % command-line -L 1
clear(lnh).                     % command-line -L 0

assign(selection_order, n).  % default n=2, range [0 .. 2]  % command-line -O n

assign(selection_measure, n).  % default n=4, range [0 .. 4]  % command-line -M n

set(negprop).      % default set     % command-line -G 1
clear(negprop).                     % command-line -G 0

set(neg_assign).      % default set     % command-line -H 1
clear(neg_assign).                      % command-line -H 0

set(neg_assign_near).      % default set     % command-line -I 1
clear(neg_assign_near).                       % command-line -I 0

set(neg_elim).      % default set     % command-line -J 1
clear(neg_elim).                     % command-line -J 0

set(neg_elim_near).      % default set     % command-line -K 1
clear(neg_elim_near).                      % command-line -K 0

set(skolems_last).                        % command-line -S 1
clear(skolems_last).    % default clear   % command-line -S 0
```

Next Section: Interpformat

Interpformat

The models (structures) in Mace4 output files can be transformed in various ways with the program Interpformat.

The transformations are listed here.

- `standard`: This transformation simply extracts the structure from the file and reprints it in the same (standard) format, with one line for each operation. The result should be acceptable to any of the LADR programs that take standard structures.
- `standard2`: This is similar to `standard`, except that the binary operations are split across multiple lines to make them more human-readable. The result should be acceptable to any of the LADR programs that take standard structures.
- `portable`: This form is list of ... of lists of strings and natural numbers. It can be parsed by seveal scripting systems such as GAP, Python, and Javascript. See the section Portable Format.
- `tabular`: This form is designed to be easily readable by humans. It is not meant for input to other programs.
- `raw`: This form is a sequence of natural numbers.
- `cooked`: This form is a sequence of ground terms.
- `xml`: This is an XML form. Here is a DTD for LADR interpretations, and here is an XML stylesheet for transforming the XML to HTML.
- `tex`: This generates LaTeX source for the interpretation.

Examples

The following Mace4 job creates an output file containing one model in "standard" (the default) format.

```
mace4 -c -f x2.in > x2.mace4.out
```

The following Interpformat jobs take the Mace4 output file, extract the model, and transform it as described above.

```
interpformat standard  -f x2.mace4.out > x2.standard
interpformat standard2 -f x2.mace4.out > x2.standard2
interpformat portable  -f x2.mace4.out > x2.portable
interpformat tabular   -f x2.mace4.out > x2.tabular
interpformat raw       -f x2.mace4.out > x2.raw
interpformat cooked    -f x2.mace4.out > x2.cooked
interpformat xml       -f x2.mace4.out > x2.xml
interpformat tex       -f x2.mace4.out > x2.tex
```

Portable Format

The portable format for interpretations can be parsed by several scriping languages including Python and GAP. Here is a counterexample on ternary relations in lattice theory. The result contains one interpretation of size 4 containing two binary functions (meet and join), one binary relation (less-or-equal), two ternary relations, and three constants.

```
mace4 -c -f LT-port.in | interpformat portable > LT-port.out
```

The result is a list of interpretations:

- each interpretation is a triple: [size-of-interpretation (say n), comments, list-of-operations];
- each operation is a 4-tuple: ["function" | "relation", name-of-operation, arity, values];
- values of operations (domain elememts are [0 ... n-1]):
  - ♦ constant (nullary function): domain element;
  - ♦ unary function: list of domain elements;
  - ♦ binary funcion: 2-dimensional list (list of lists) of domain elements;
  - ♦ ternary funcion: 3-dimensional list of domain elements;
  - ♦ etc.
  - ♦ relations are similar but with values of 0 (FALSE) or 1 (TRUE).

Here is a simple Python script that reads a list of portable interpretations and prints them in a different form.

```
port.py < LT-port.out > LT-port.out2
```

Here is a simple GAP session that reads and prints a list of portable interpretations.

```
% gap -b
GAP4, Version: 4.4.7 of 17-Mar-2006, i486-pc-linux-gnu-i486-linux-gnu-gcc
gap> interpretations := EvalString(StringFile("LT-port.out"));;
gap> interpretations;
[ [ 4, [ "=(number,1)", "=(seconds,0)" ],
      [ [ "relation", "<=", 2, [ [ 1, 1, 1, 1 ], [ 0, 1, 0, 0 ],
                  [ 0, 1, 1, 0 ], [ 0, 1, 0, 1 ] ] ],
          [ "function", "^", 2, [ [ 0, 0, 0, 0 ], [ 0, 1, 2, 3 ],
                  [ 0, 2, 2, 0 ], [ 0, 3, 0, 3 ] ] ],
          [ "function", "v", 2, [ [ 0, 1, 2, 3 ], [ 1, 1, 1, 1 ],
                  [ 2, 1, 2, 1 ], [ 3, 1, 1, 3 ] ] ],
          [ "function", "c1", 0, 2 ], [ "function", "c2", 0, 0 ],
          [ "function", "c3", 0, 3 ],
          [ "relation", "A", 3, [ [ [ 1, 1, 1, 1 ], [ 0, 1, 0, 0 ],
                      [ 0, 1, 1, 0 ], [ 0, 1, 0, 1 ] ],
                  [ [ 1, 0, 0, 0 ], [ 1, 1, 1, 1 ], [ 1, 0, 1, 0 ],
                      [ 1, 0, 0, 1 ] ],
                  [ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 1, 1, 1, 0 ],
                      [ 0, 0, 0, 0 ] ],
                  [ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 0, 0, 0, 0 ],
                      [ 1, 1, 0, 1 ] ] ] ],
          [ "relation", "B", 3, [ [ [ 1, 1, 1, 1 ], [ 0, 1, 0, 0 ],
                      [ 0, 1, 1, 0 ], [ 0, 1, 0, 1 ] ],
                  [ [ 1, 0, 0, 0 ], [ 1, 1, 1, 1 ], [ 1, 0, 1, 0 ],
                      [ 1, 0, 0, 1 ] ],
                  [ [ 1, 0, 0, 1 ], [ 0, 1, 0, 1 ], [ 1, 1, 1, 1 ],
                      [ 0, 0, 0, 1 ] ],
                  [ [ 1, 0, 1, 0 ], [ 0, 1, 1, 0 ], [ 0, 0, 1, 0 ],
                      [ 1, 1, 1, 1 ] ] ] ] ] ]
gap>
```

Next Section: Isofilter

Isofilter

If Mace4 produces more than one structure, some of them are very likely to be isomorphic to others. The program Isofilter can be used to remove isomorphic structures.

Determining whether two structures are isomorphic is a hard problem in general, but isofilter can cope with some large structures in reasonable time. It depends on the type of the strucure. For example, quasigroups usually take more time than lattices.

Isofilter accepts structures in LADR standard format, which is the default format produced by Mace4. However, the Mace4 output contains a lot of extraneous information, which can be stripped out with the command `interpformat standard`. Isofilter also accepts the following command-line arguments, which are described in the examples below.

- `ignore_constants`: ignore all constants during the isomorphism tests.
- `check '<operations>'` : consider only the listed operations in the isomorphism tests.
- `wrap`: enclose the resulting structures in `list(interpretations). ... end_of_list.`

Examples

We start with a Mace4 job and extract the interpretations; then we run Isofilter.

```
mace4 -N6 -m -1 -f BA2.in | interpformat standard > BA2.interps
isofilter < BA2.interps > BA2.interps2
```

Note that the two models in BA2.interps2 differ only in one of the constants. In this case the constants come from existentially quantified variables in the goal, and all we really care about is the lattice. We can tell Isofilter to ignore differences in constants by giving it the argument `ignore_constants` as in the following command.

```
isofilter ignore_constants < BA2.interps > BA2.interps3
```

Another way to use only a subset of the operations is the `check` argument, which us used to specify *exactly* which operations to test for isomorphism. In the following example, only the meet and join operations are checked. (If there is more than one operation, or if the operation may be interpreted by the shell, they should be enclosed in single quotes.)

```
mace4 -N6 -m -1 -f MOL.in | interpformat standard > MOL.interps
isofilter check '^ v' < MOL.interps > MOL.interps2
```

The output of isofilter is frequently used as input to a program that expects the interpretations to be in a list of interpretations; we can tell isofilter to put the output in that form by giving it the argument `wrap` as in the following command.

```
isofilter ignore_constants wrap < BA2.interps > BA2.interps4
```

Finally, we can string together some of the preceding commands as follows.

```
mace4 -N6 -m -1 -f BA2.in | interpformat standard | isofilter ignore_constants wrap > BA2.interps5
```

Next Section: Prooftrans

Prooftrans

When Prover9 proves a theorem, it sends the proof to its output file in a standard form. The standard form contains, for each step, justifications with enough detail to reconstruct or check the proof without any search.

Prover9 proofs may contain non-clausal assumptions and goals, as well as ordinary clauses. Non-clausal assumptions are translated to clauses, and goals are negated and then translated to clauses. See the proof in following example

```
prover9 -f subset_trans.in > subset_trans.out
```

Prooftrans can extract proofs from Prover9 output files and transform them in various ways, including the following.

- No transformation,
- renumber steps,
- simplify justifications,
- expand all steps, turning secondary justifications into explicit steps,
- produce proofs in XML,
- produce proofs for checking by the IVY proof checker, and
- produce hints for guiding subsequent searches.

Prooftrans is part of the LADR/Prover9/Mace4 package. When the package is installed, the Prooftrans program should be in the same directory as Prover9 and Mace4.

Using Prooftrans

The Prover9 output file containing the proof(s) is usually given to Prooftrans with the argument "`-f <filename>`". If there is no "`-f <filename>`" argument, Prooftrans takes its input from the standard input.

The arguments that tell Prooftrans what to do with the proof(s) are described in the following sections, using the output file subset_trans.out as a running example.

If there is more than one proof in the file, the transformations will be applied to each proof. The `hints` transformation collects all of the clauses in the proof(s) into one list of hints. The other transformations produce one proof for each proof in the input file.

Here is a synopsis of the Prooftrans command; the arguments in square brackets are optional.

```
prooftrans [parents_only] [expand] [renumber] [striplabels] [-f file]
prooftrans xml            [expand] [renumber] [striplabels] [-f file]
prooftrans ivy                     [renumner]               [-f file]
prooftrans hints [-label label] [expand]      [striplabels] [-f file]
```

Note that more than one transformation can be applied in several cases. The option "striplabels" tells prooftrans to remove all label attributes on clauses.

Unfortunately, the output of Prooftrans usually cannot be used as the input to another Prooftrans job, because Prooftrans expects its input to have specific keywords and standard-form proofs.

No Transformation

If no additional argument is given, Prooftrans simply extracts the proof from the Prover9 output file.

```
prooftrans -f subset_trans.out > subset_trans.proof1
```

Renumber the Steps

The argument `renumber` tells Prooftrans to renumber the steps of each proof consecutively, starting with step 1. The `expand`, `parents_only`, and `xml` transformations can be used with the `renumber` transformation.

```
prooftrans renumber -f subset_trans.out > subset_trans.proof2
```

Simplify Justifications

The argument `parents_only` tells Prooftrans list only the parents in the justifications, not the details about inference rules or positions. The `expand` and `renumber` transformations can be used with the `parents_only` transformation.

```
prooftrans parents_only -f subset_trans.out > subset_trans.proof3
```

Expand Steps

The argument `expand` tells Prooftrans to produce more detailed proofs in which

- all hyper- and UR-resolution steps are replaced with binary resolution steps,
- all demodulation sequences are replaced with paramodulation steps, and
- all unit deletion simplifications are replaced with resolution steps.

Note to author: this is a bad example, because only one step gets expanded.

```
prooftrans expand -f subset_trans.out > subset_trans.proof4
```

Note that when a step is expanded (step 22 in this example), the new steps are identified by appending 'A', 'B', etc. to the number of the original step.

The `renumber`, `parents_only`, and `hints` transformations can be used with the `expand` transformation.

XML Proofs

The options `xml` or `XML` tell Prooftrans to produce proofs in XML. The options `expand` and `renumber` can be used with the XML transformation.

```
prooftrans xml -f subset_trans.out > subset_trans.proof5.xml
```

The preceding output is displayed by your browser not as XML, but as some transformation of the XML, because the XML refers to an XML stylesheet, telling the browser how to transform the XML into HTML.

To see the XML source, click "View -> Frame Source" (or something like that) in your browser while viewing the proof.

Here is the DTD for Prover9 XML proofs. (If you get an error, click "View -> Page Source".)

IVY Proofs

The options `ivy` or `IVY` tell Prooftrans to produce very detailed proofs that can be checked with the Ivy proof checker.

```
prooftrans ivy -f subset_trans.out > subset_trans.proof6
```

Ivy proofs have a only 5 types of step: `input`, `propositional`, `new_symbol`, `flip`, `instantiate`, `resolve`, and `paramod`. The `resolve` and `paramod` do not involve unification; instances are generated first as separate steps, and then `resolve` or `paramod` are applied to identical atomic formulas or terms.

The Ivy proof checker cannot check steps justified by `new_symbol`.

Proofs to Hints

The option `hints` tells Prooftrans to take all of the proofs in the file and produce one list of hints that can be given to Prover9 to guide subsequent searches on related conjectures.

```
prooftrans hints -f subset_trans.out > subset_trans.proof7
```

If there is more than one proof in the file, the proofs will probably share many steps. The list of hints that Prooftrans produces will be the union of the steps in the proofs; that is, the duplicate steps will be removed.

The `expand` transformation can be used with the `hints` transformation.

The label option tells prooftrans to attach label attributes to the hint clauses. The labels consist of the string given on the command line and a sequence number generated by prooftrans. The user's command shell may require that the label be quoted, and if the the label is not a legal LADR constant, prooftrans will enclose the label in double quotes.

```
prooftrans hints -label 'job8' -f subset_trans.out > subset_trans.proof8
```

Next Section: FOF-Prover9

FOF-Prover9

FOF (First-Order Formula) reduction is a method of attempting to simplify a problem by reducing it to an equivalent set of independent subproblems. A trivial example is to reduce the conjecture `A B` to the pair of problems `A -> B` and `B -> A`.

The problem reduction uses a miniscope method [Champeaux-miniscope] that is quite powerful in some cases, but it can easily blow up on complex formulas. Therefore, if the reduction procedure fails to terminate within a few seconds, or if the subproblems it produces are more complex than the original problem, the reduction attempt is aborted (and the user may wish to try the ordinary Prover9 program instead). If the reduction succeeds, each subproblem is given to the ordinary Prover9 search module.

Input files accepted by FOF-Prover9 are the same as those accepted by Prover9, and when FOF-Prover9 runs the search module on a subproblem, is uses all of the options given in the input file.

An Example of FOF Reduction

This example was given by Peter Andrews as a challenge problem for resolution systems in the 1970s. FOF-Prover9's miniscope procedure reduces it to 32 trivial subproblems. (More powerful miniscope methods can solve the problem by reducing it to 0 subproblems.)

```
fof-prover9 -f andrews.in > andrews.out
```

Here is the same input run with ordinary Prover9.

```
prover9 -f andrews.in > andrews.out2
```

The preceding example is artificial and seems tailor-made for FOF reduction. Other, more realistic examples can be found in the standard set of Prover9 examples.

Next Section: More Programs

Other LADR Prograns

The page describes several other programs that have constructed with the same code base (LADR) as Prover9, Prooftrans, Mace4, and Interpformat.

When we write that a program takes a "stream" of objects, we mean that it reads them from the standard input, and the objects are *not* enclosed in `objects(..)` and `end_of_list`.

When we write that a program takes a "set" of objects, we mean that the filename containing the objects is an argument to the program, and the objects are *not* enclosed in `objects(..)` and `end_of_list`.

When we write that a program takes a "list" of objects, we mean that the filename containing the objects is an argument to the program, and the objects are enclosed in `objects(..)` and `end_of_list`.

Contents

Clausefilter

Given a set of interpretations, a test to perform, and a stream of formulas, this program outputs the formulas that pass the test. (If non-clausal formulas with free variables are given, their universal closures are used and output.)

The accepted tests are `true_in_all`, `true_in_some`, `false_in_all`, and `false_in_some`.

Example: given a set of non-modular orthomodular lattices and a stream of identities, print the identities that are false in all of the lattices. This job was used when searching for modular ortholattice (MOL) single axioms: any MOL single axiom is false in all non MOLs.

```
clausefilter non-MOL-OML.interps false_in_all < MOL-cand.296 > MOL-cand.238
```

Clausetester

This program takes a set of interpretations and stream of formulas. For each formula, the interpretations in which the formula is true are shown, and at the end the number of formulas true in each interpretation is shown. (If non-clausal formulas with free variables are given, their universal closures are used and output.)

Example:

```
clausetester uc-18.interps < uc-hunt.clauses > uc-hunt.out
```

Interpfilter

Given a set of formulas, a test to perform, and a stream of interpretations, this program outputs the interpretations that pass the test. (If non-clausal formulas with free variables are given, their universal closures are used.)

The accepted tests are `all_true`, `some_true`, `all_false`, and `some_false`.

Example: from a list of quasigroups, extract the associative-commutaive ones.

```
interpfilter assoc-comm.clauses all_true < qg4.interps > qg4-ac.interps
```

Rewriter

Rewrite a stream of terms with a list of demodulators. The demodulators are used left-to-right as given, and they are not checked for termination.

Basic example that canonicalizes group expressions:

```
rewriter group.demods < group-terms.in > group-terms.out
```

This example canonicalizes Boolean ring expressions. It uses associative-commutative (AC) operations and the op command to change the parsing rules.

```
rewriter bool-ring.demods < bool-ring.in > bool-ring.out
```

This example rewrites identities in terms of {meet,join,complementation} into the Sheffer stroke.

```
rewriter BA-Sheffer.demods < BA4.in > BA4.out
```

TPTP Translators

The TPTP Problem Library contains thousands of problems for theorem provers, and the TPTP Language is widely used in the community. LADR has two programs to translate between the LADR and TPTP languages. These programs are new and experimenal, and they do not support all of the language features.

TPTP_to_LADR This program takes a TPTP problem file and produces a bare input file suitable for input to Prover9 or Mace4. For example,

```
tptp_to_ladr < PUZ031-1.p > PUZ031-1.in
prover9 -f PUZ031-1.in > PUZ031-1.out
```

If you prefer, those two processes can be piped together:

```
tptp_to_ladr < PUZ031-1.p | prover9 > PUZ031-1.out2
```

Some of the TPTP language features that are *not yet supported* are comment blocks, system comments, real numbers, natural numbers as distinct objects, and distinct objects with double quotes.

Some future version of LADR will likely support direct input of TPTP files to Prover9 and Mace4, without having to invoke a translator.

LADR_to_TPTP This program takes a Prover9 input file and produces a TPTP problem file. A difficulty with this kind of translation is that TPTP accepts a more restriced class of function and predicate symbols. When the translator sees symbols that are not accepted by TPTP, it introduces new symbols, and it gives the symbol mapping as comments in the output. Ordinary TPTP constant, function, and predicate symbols must start with lower case a-z, and any remaining characters must be alphanumeric or _ (underscore). That is, they must match the (Unix-style) regular expression "`[a-z][a-zA-Z0-9_]*`".

Here is an example that contains several unacceptable symbols.

```
ladr_to_tptp < RBA-2.in > RBA-2.p
```

Next Section: All Options

TPTP_to_LADR This program takes a TPTP problem file and produces abare input file suitable for input to Prov

lphanumeric or _(underscore). That is, they must match the (Unix-style) regular expression"[a-z][a-zA-Z0-9_]*".

Prover9 Options

There are three kinds of options:

- *Flags* are Boolean-valued options which can be changed with the *set* and *clear* commands, e.g.,
  `set(clocks).`
  `set(print_given).`
- *Parms* are integer-valued options which can be changed with the *assign* command, e.g.,
  `assign(max_weight, 30).`
- *Stringparms* are string-valued options which can be changed with the *assign* command, e.g.,
  `assign(order, kbo).`

Option Dependencies

Several of the flags and parameters cause other flags and parameters to be changed. In some cases, that is the only direct effect they have. For example, if you `clear(auto)`, you will see the following in the output.

```
clear(auto).
    % clear(auto) -> clear(auto_inference).
    % clear(auto_inference) -> clear(predicate_elim).
    % clear(auto_inference) -> assign(eq_defs, pass).
    % clear(auto) -> clear(auto_limits).
    % clear(auto_limits) -> assign(max_weight, 2147483647).
    % clear(auto_limits) -> assign(sos_limit, -1).
```

The lines starting with "`%`" are the dependent options that are changed in behalf of `clear(auto)`. Note the sub-dependencies in this example.

The option dependencies can be undone by simply changing the dependent option afterward, as in the following example input.

```
clear(auto).
set(predicate_elim).
```

Option Listing

The option names below are links to the sections containing the descriptions.

From Page Clauses and Formulas

```
set(prolog_style_variables).
clear(prolog_style_variables).    % default clear
```

From Page Automatic Modes

```
set(auto).    % default set
clear(auto).


set(auto_inference).    % default set
clear(auto_inference).


set(auto_process).    % default set
clear(auto_process).


set(auto_limits).    % default set
clear(auto_limits).


set(auto2).
clear(auto2).    % default clear


assign(lrs_ticks, n).  % default n=-1, range [-1 .. INT_MAX]


assign(lrs_interval, n).  % default n=50, range [1 .. INT_MAX]


assign(min_sos_limit, n).  % default n=0, range [0 .. INT_MAX]
```

From Page Term Ordering

```
assign(order, string).  % default string=lpo, range [lpo,rpo,kbo]


set(inverse_order).    % default set
clear(inverse_order).


assign(eq_defs, string).  % default string=unfold, range [unfold,fold,pass]
```

From Page More Search Prep

```
set(expand_relational_defs).
clear(expand_relational_defs).    % default clear


set(predicate_elim).    % default set
clear(predicate_elim).
```

```
assign(fold_denial_max, n).  % default n=0, range [-1 .. INT_MAX]


set(sort_initial_sos).
clear(sort_initial_sos).    % default clear


set(process_initial_sos).    % default set
clear(process_initial_sos).
```

From Page Search Limits

```
assign(sos_limit, n).  % default n=20000, range [-1 .. INT_MAX]


assign(max_given, n).  % default n=-1, range [-1 .. INT_MAX]


assign(max_kept, n).  % default n=-1, range [-1 .. INT_MAX]


assign(max_megs, n).  % default n=200, range [-1 .. INT_MAX]


assign(max_seconds, n).  % default n=-1, range [-1 .. INT_MAX]


assign(max_minutes, n).  % default n=-1, range [-1 .. INT_MAX]


assign(max_hours, n).  % default n=-1, range [-1 .. INT_MAX]


assign(max_days, n).  % default n=-1, range [-1 .. INT_MAX]
```

From Page Selecting the Given Clause

```
assign(age_part, n).     % default n=1, range [0 .. INT_MAX]


assign(weight_part, n).  % default n=0, range [0 .. INT_MAX]


assign(false_part, n).   % default n=4, range [0 .. INT_MAX]


assign(true_part, n).    % default n=4, range [0 .. INT_MAX]


assign(random_part, n).  % default n=0, range [0 .. INT_MAX]
```

```
assign(hints part, n).   % default n=INT_MAX, range [0 .. INT_MAX]


set(default parts).      % default set
clear(default_parts).


assign(pick given ratio, n).  % default n=0, range [0 .. INT_MAX]


set(lightest first).
clear(lightest_first).    % default clear


set(breadth first).
clear(breadth_first).    % default clear


set(random given).
clear(random_given).    % default clear


assign(random seed, n).  % default n=0, range [-1 .. INT_MAX]


set(input sos first).    % default set
clear(input_sos_first).
```

From Page Inference Rules

```
set(binary resolution).
clear(binary_resolution).    % default clear


set(neg binary resolution).
clear(neg_binary_resolution).    % default clear


set(ordered res).    % default set
clear(ordered_res).


set(check res instances).
clear(check_res_instances).    % default clear


assign(literal selection, string).  % default string=maximal_negative, range [maximal_negative, all_negativ


set(positive inference).    % default set
clear(positive_inference).


set(pos hyper resolution).
```

```
clear(pos_hyper_resolution).    % default clear


set(hyper_resolution).
clear(hyper_resolution).    % default clear


set(neg_hyper_resolution).
clear(neg_hyper_resolution).    % default clear


set(ur_resolution).
clear(ur_resolution).    % default clear


set(pos_ur_resolution).
clear(pos_ur_resolution).    % default clear


set(neg_ur_resolution).
clear(neg_ur_resolution).    % default clear


set(initial_nuclei).
clear(initial_nuclei).    % default clear


assign(ur_nucleus_limit, n).  % default n=-1, range [-1 .. INT_MAX]


set(paramodulation).
clear(paramodulation).    % default clear


set(ordered_para).    % default set
clear(ordered_para).


set(check_para_instances).
clear(check_para_instances).    % default clear


set(para_from_vars).    % default set
clear(para_from_vars).


assign(para_lit_limit, n).  % default n=-1, range [-1 .. INT_MAX]


set(para_units_only).
clear(para_units_only).    % default clear


set(basic_paramodulation).
clear(basic_paramodulation).    % default clear
```

From Page Processing Inferred Clauses

```
set(lex_order_vars).
clear(lex_order_vars).    % default clear


assign(demod_step_limit, n).  % default n=1000, range [-1 .. INT_MAX]


assign(demod_size_limit, n).  % default n=1000, range [-1 .. INT_MAX]


set(back_demod).
clear(back_demod).    % default clear


set(lex_dep_demod).    % default set
clear(lex_dep_demod).


assign(lex_dep_demod_lim, n).  % default n=11, range [-1 .. INT_MAX]


set(lex_dep_demod_sane).    % default set
clear(lex_dep_demod_sane).


set(unit_deletion).
clear(unit_deletion).    % default clear


set(back_unit_deletion).
clear(back_unit_deletion).    % default clear


set(cac_redundancy).    % default set
clear(cac_redundancy).


assign(max_literals, n).  % default n=-1, range [-1 .. INT_MAX]


assign(max_depth, n).  % default n=-1, range [-1 .. INT_MAX]


assign(max_vars, n).  % default n=-1, range [-1 .. INT_MAX]


assign(max_weight, n).  % default n=100, range [INT_MIN .. INT_MAX]


set(safe_unit_conflict).
clear(safe_unit_conflict).    % default clear
```

```
set(factor).
clear(factor).    % default clear


assign(new constants, n).  % default n=0, range [-1 .. INT_MAX]


set(back subsume).    % default set
clear(back_subsume).
```

From Page Output Files

```
set(echo input).    % default set
clear(echo_input).


set(quiet).
clear(quiet).    % default clear


set(print initial clauses).    % default set
clear(print_initial_clauses).


set(print given).    % default set
clear(print_given).


set(print gen).
clear(print_gen).    % default clear


set(print kept).
clear(print_kept).    % default clear


set(print labeled).
clear(print_labeled).    % default clear


set(print clause properties).
clear(print_clause_properties).    % default clear


set(print proofs).    % default set
clear(print_proofs).


set(default output).    % default set
clear(default_output).


assign(report, n).  % default n=-1, range [-1 .. INT_MAX]
```

```
assign(stats, string).  % default string=lots, range [none,some,lots,all]


set(clocks).
clear(clocks).    % default clear


set(bell).    % default set
clear(bell).
```

From Page Weighting

```
assign(constant_weight, n).  % default n=1, range [INT_MIN .. INT_MAX]


assign(sk_constant_weight, n).  % default n=1, range [INT_MIN .. INT_MAX]


assign(variable_weight, n).  % default n=1, range [INT_MIN .. INT_MAX]


assign(not_weight, n).  % default n=0, range [INT_MIN .. INT_MAX]


assign(or_weight, n).  % default n=0, range [INT_MIN .. INT_MAX]


assign(prop_atom_weight, n).  % default n=1, range [INT_MIN .. INT_MAX]


assign(nest_penalty, n).  % default n=0, range [0 .. INT_MAX]


assign(skolem_penalty, n).  % default n=1, range [0 .. INT_MAX]


assign(depth_penalty, n).  % default n=0, range [INT_MIN .. INT_MAX]


assign(var_penalty, n).  % default n=0, range [INT_MIN .. INT_MAX]


assign(default_weight, n).  % default n=INT_MAX, range [INT_MIN .. INT_MAX]
```

From Page Goals and Denials

```
assign(max_proofs, n).  % default n=1, range [-1 .. INT_MAX]


set(restrict_denials).
clear(restrict_denials).    % default clear
```

```
set(reuse_denials).
clear(reuse_denials).    % default clear


set(auto_denials).    % default set
clear(auto_denials).
```

From Page Hints

```
set(breadth_first_hints).
clear(breadth_first_hints).    % default clear


set(degrade_hints).    % default set
clear(degrade_hints).


set(back_demod_hints).    % default set
clear(back_demod_hints).


set(collect_hint_labels).
clear(collect_hint_labels).    % default clear
```

From Page Semantic Guidance

```
assign(eval_limit, n).  % default n=1024, range [-1 .. INT_MAX]
```

Next Section: Glossary

Glossary

Under construction. (Send suggestions of terms to include.)

Terms, Clauses, Formulas, Interpretations

These definitions apply to first-order unsorted logic. See a book on first-order logic for more formal definitions of these concepts.

Term

A recursive definition of first-order unsorted *terms*.

- A variable is a term,
- a constant is a term, and
- an *n*-ary function symbol applied to *n* terms is a term.

Atomic Formula

An *n*-ary predicate symbol applied to *n* terms is an *atomic formula*.

Formula

- An atomic formula is a formula,
- if `F` and `G` are formulas, then the following are formulas.
  - `(-F)`
  - `(F | G)`
  - `(F & G)`
  - `(F -> G)`
  - `(F G)`
- if `F` is a formula and `x` is a variable, then the following are formulas.
  - `(all x F)`
  - `(exists x F)`

When writing formulas for Prover9, many of the parentheses can be omitted; see the page Clauses and Formulas tor the parsing rules.

Free Variables

A *free variable* is a variable not bound by any quantifier. A *closed formula* has no free variables. An *open formula* has at least one free variable.

Prover9's default rule for distinguishing free variables from constants is that free varaibles start with (lower case) 'u' through 'z'.

Literal

A *literal* is either an atomic formula or the negation of an atomic formula.

Clause

A *clause* is a formula consisting of a disjunction of literals. All variables in a clause are assumed to be universally quantified.

Interpretation

An *interpretation* of a first-order language consists of

- of a set of objects called the *domain*,
- an *n*-ary function over the domain into the domain for each *n*-ary function symbol in the language,
- an *n*-ary relation over the domain for each *n*-ary predicate symbol in the language.

Given an interpretation, each term in the language evaluates to a member of the domain, and each clause or closed formula in the language evaluates to TRUE or to FALSE.

Types of Clause

Unit Clause

A *unit clause* has exactly one literal.

Positive Clause, Negative Clause, Mixed Clause

A *positive clause* has no negative literals. A *negative clause* has no positive literals. Note that the empty clause is both positive and negative. A *mixed clause* has at least one literal of each sign.

Horn Clause, Horn Set

A *Horn clause* has at most one positive literal. A Horn set is a set of Horn clauses.

Definite Clause

A *definite clause* has exactly one positive literal.

Logic Transformations

Negation Normal Form (NNF)

A formula is in *negation normal form* if the only logic connectives are negation, conjunction, disjunction, quantification (universal or existential), and if all negation operations are applied directly to atomic formulas.

Conjunctive Normal Form (CNF)

This definition applies to quantifier-free formulas.

A formula is in *conjunctive normal form* if (1) the only logic connectives are negation, conjunction, and disjunction, (2) no negation is applied to a conjunction or a disjunction, and (3) no disjunction is applied to a conjunction.

Alternate definition: A formula is in CNF if it is a clause or a conjunction of clauses.

Skolemization

Skolemization is the process of replacing existentially quantified variables in a formula with new constants (called *Skolem constants*) or functions (called *Skolem functions*). If an existential quantifier is in the scope of some universal quantifiers, the new symbol is a function of the corresponding universally quantified variables. The result of Skolemization is not, strictly speaking, equivalent to the original formula, because new symbols may have been introduced, but the result is inconsistent iff the the original formula is inconsistent.

Clausification

Clausification is the process of translating a formula into a conjunction of clauses. A standard way is NNF conversion, Skolemization, moving universal quantifiers to the top (renaming bound variables if necessary), CNF conversion, and finally removing universal quantifiers. The variables in each resulting clause are implicitly universally quantified.

Each step produces an equivalent formula, except for Skolemization, which produces an equiconsistent formula, so the result of Clausification is inconsistent iff the original formula is inconsistent.

Universal Closure

The *universal closure* of a formula is constructed by universally quantifying, at the top of the formula, all free variables in the formula.

Term Ordering Terminology

Knuth-Bendix Ordering (KBO)

Lexicographic Path Ordering (LPO)

Recursive Path Ordering (RPO)

Maximal Literal

A literal is *maximal* in a clause, with respect to some term ordering, if no literal in the clause is greater. The terms orderings used by Prover9 (LPO, KBO, RPO) are, in general, only partial, so clauses do not necessarily have greatest literals.

Inference and Simplification Rules

Completeness

An inference system is *complete* if it is capable (given enough time and memory) of proving any theorem (in the language of the inference system).

Binary Resolution

The inference rule *binary resolution* takes two clauses containing unifiable literals of opposite sign and produces a clause consisting of the remaining literals to which the most general unifying substitution has been applied. The rule can be viewed as a generalization of modus ponens.

Restrictions on Binary Resolution.

- *Positive resolution*: one of the parents is is a positive clause.
- *Negative resolution*: one of the parents is is a negative clause.
- *Unit resolution*: one of the parents is is a unit clause.

Ordered Inference, Literal Selection

Ordered Inference is a restriction of resolution or paramdulation based on literal ordering. In some cases, inferences can be restricted to maximal literals.

Literal selection is a restriction of resolution or paramdulation. In each clause, some subset of the negative literals are marked as selected (the selection may be arbitrary), and in some cases inferences can be restricted to selected literals.

Factoring

The inference rule *factoring* takes one clause containing two or more literals (of the same sign) that all unify. The most general unifying substitution is applied to the parent, and the unified literals become identical and are merged into one.

Factoring in Prover9 is binary, operating on two literals at a time.

Hyperresolution

The *hyperresolution* inference rule (also called positive hyperresolution) takes a non-positive clause (called the *nucleus*) and simultaneously resolves each of its negative literals with positive clauses (called the *satellites*), producing a positive clause. Hyperresolution can be viewed as a sequence of positive binary resolution steps ending with a positive clause.

Negative hyperresolution is similar to hyperresolution but with the signs reversed.

UR-Resolution

The *UR-resolution* (unit-resulting resolution) inference rule takes a nonunit clause (called the *nucleus*) and resolves all but one of its literals with unit clauses (called the *satellites*), producing a unit clause.

Positive UR-resolution is UR-resolution with the constraint that the result must be a positive unit clause.

Negative UR-resolution is UR-resolution with the constraint that the result must be a negative unit clause.

"From" and "Into" in Paramodulation

A *paramodulation* inference consists of two parents and a child. The parent containing the equality used for the replacement is the *from* parent or the *from clause*, the equality is the *from* literal, and the side of the equality that unifies with the term being replaced is the *from* term.

The replaced term is the *into* term, the literal containing the replaced term is the *into* literal, and the parent containing the replaced term is the *into* parent or *into* clause.

Superposition is a restricted form of paramodulation in which substitution is not allowed into the lighter side of an equation.

Positive Paramodulation

Positive paramodulation is a restriction of paramodulation in which the "from" clause is positive, and if the "into" literal is positive, the "into" clause is also positive.

Demodulation, Back Demodulation

Demodulation (also called *rewriting*) is the process of using a set of oriented equations (demodulators) to rewrite (simplify, canonicalize) terms. If the demodulators have good properties, demodulation terminates.

Forward demodulation (or just demodulation) is the process of using a set of demodulators to rewrite newly generated clauses.

Back demodulation is the process of using a new demodulator to simplify previously stored clauses.

Unit Deletion, Back Unit Deletion

Unit deletion is analogous to demodulation. The difference is that unit clauses, rather than equations, are used for simplification.

Unit deletion is the process of using unit clauses to remove literals from newly generated clauses.

Back unit deletion is the process of using a new unit clause to remove literals from previously stored clauses.

Subsumption, Forward and Backward Subsumption

Clause C *subsumes* clause D if the variables of C can be instantiated in such a way that it becomes a subclause of D. If C subsumes D, then D can be discarded, because it is weaker than or equivalent to C. (There are some proof procedures that require retention of subsumed clauses.)

Forward subsumption (or just subsumption) is the process of deleting a newly derived clause if it is subsumed by some previously stored clause.

Back subsumption is the process of deleting all previously stored clauses that are subsumed by a newly derived clause.

Unit Conflict

Unit Conflict is an inference rule that derives a contradiction from unit clauses, for example, from `P(a,b)` and `-P(x,b)`.

Prover9 Terminology

Given Clause

The *given clause loop* drives the inference process int Prover9. At each iteration of the loop, a *given clause* is selected from the sos list, moved the the usable list, and then inferences are made using the given clause and other clauses in the usable list.

Sos List, Assumptions List, Usable List

During the search, the *usable* list is the list of clauses that are available for making inferences with the given clause, and the *sos list* is the list of clauses that are waiting to be selected as given clauses. Clauses in the sos list are not available for making primary inferences, but they can be used to simplify inferred clauses by demodulation and unit deletion.

The *assumptions list* is identical to the sos list; that is, "assumptions" is a synonym for "sos".

Prover9 also accepts non-clausal formulas in lists named usable or sos. Such formulas are transformed to clauses which are placed in the clause list of the same name.

The name "sos" is used because it can be employed to achieve the set-of-support strategy, which requires that all lines of reasoning start with a subset of the input clauses called the set of support. The clauses or formulas in the initial set of support are placed the sos list, and the rest of the

clauses or formulas are placed in the usable list.

Goal, Goals List

A *goal* is the conclusion of a conjecture, stated in positive form. Each goals is transformed to clauses by constructing its universal closure, negation, then clausification.

If there is more than one goal, Prover9 may impose restrictions on the structure of the goals.

Hint, Hints List

Hints are clauses that are intended to guide Prover9 toward proofs. Hints are not part of the problem specification; that is, they are not used for making inferences. They are used only as a component of the weighting function for selecting given clauses.

Initial Clause

A clause is an *initial clause* if it exists at the time when the first given clause is selected. Initial clauses are not necessarily input clauses, because they may be created during preprocessing, for example, by rewriting or clausification.

Denial

In Prover9 terminology, a negative clause in a Horn set is called a *denial*, because such clauses usually come from the negation of a conclusion. (The term does not apply to non-Horn sets, because a proof of a non-Horn set may require more than one negative clause.)

FOF Reduction

FOF (First-Order Formula) reduction is a method of attempting to simplify a problem by reducing it to an equivalent set of independent subproblems. A trivial example is to reduce the conjecture `A B` to the pair of problems `A -> B` and `B -> A`.

Lex-Dependent Demodulator

A *lex-dependent demodulator* is one that cannot be oriented by the primary term ordering (LPO or KBO). An example is commutativity of a binary operation. During demodulation, a lex-dependent demodulator is applied only if it produces a term that is smaller in the primary term ordering.

Depth of Term, Atom, Literal, Clause

- depth of variable, constant, or propositional atom: 0;
- depth of term or atom with arguments: one more than the maximum argument depth;
- depth of literal: depth of its atom (negation signs don't count);
- depth of clause: maximum of depths of literals;

For example, `p(x) | -p(f(x))` has depth 2.

Relational Definition

A *relational definition* for an *n*-ary relation (say `P` with *n*=3) is a closed formula of the form (using Prover9 syntax)

```
all x all y all z (P(x,y,z) f)
```

for some formula *f* that does not contain the symbol `P`.

Equational Definition

An *equational definition* for an *n*-ary function (say `f` with *n*=3) is an equation (using Prover9 syntax)

```
f(x,y,z) = t
```

for some term *t* that does not contain the symbol `f` and that does not contain free variables other than `x`, `y`, and `z`.

Next Section: References

References

Not done yet.

[Bachmair-Ganzinger-res]
L. Bachmair and H. Ganzinger. "Resolution Theorem Proving". Chapter 2 in *Handbook of Automated Reasoning* (ed. A. Robinson and A. Voronkov), 2001. Preliminary version.

[Nieuwenhuis-Rubio-para]
R. Nieuwenhuis and A. Rubio. "Paramodulation-Based Theorem Proving". Chapter 7 in *Handbook of Automated Reasoning* (ed. A. Robinson and A. Voronkov), 2001.

[Champeaux-miniscope]
D. Champeaux. "Sub-problem finder and instance checker, two cooperating modules for theorem provers". *J. ACM*, 33(4):633--657, 1986.

[Dershowitz-termination]
N. Dershowitz. "Termination of rewriting". *J. Symbolic Computation*, 3:69-116, 1987.

[McCune-Otter33]
W. McCune. *Otter 3.3 Reference Manual*. Tech. Memo ANL/MCS-TM-263, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, August 2003.

[McCune-Mace4]
W. McCune. *Mace4 Reference Manual and Guide*. Tech. Memo ANL/MCS-TM-264, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, August 2003.

[RV-lrs]
A. Riazanov and A. Voronkov. "Limited resource strategy in resolution theorem proving". *J. Symbolic Computation*, 36(1-2):101-115, 2003.

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

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