Distribution Category:
Mathematics and Computers
(UC-32)

ANL-82-75

Argonne National Laboratory
9700 South Cass Avenue
Argonne, Illinois 60439

AN IMA-BASED THEOREM PROVER*

by

Ewing L. Lusk

and

Ross A. Overbeek

Mathematics and Computer Science Division

December 1982

DISCLAIMER

The views expressed in this report are those of the authors and may not reflect the position of the U.S. Department of Energy.

* This work was supported by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under Contract W-31-109-Eng-36, and also by National Science Foundation grant MCS 79-03870.
CONTENTS

Abstract ......................................................................................................................... 1

1. Introduction ............................................................................................................. 1
2. Basic Operation of the System ............................................................................... 2
3. Getting Started ......................................................................................................... 2
   3.1. Format of Clauses .............................................................................................. 2
   3.2. Some Elementary Commands ............................................................................ 3
4. A Sample Session ..................................................................................................... 4
5. Some More Commands ............................................................................................ 4
6. Introduction to Options ............................................................................................ 5
7. Inference Rules ......................................................................................................... 6
   7.1 Hyperresolution .................................................................................................. 6
   7.2 UR-resolution .................................................................................................... 8
   7.3 Binary Resolution .............................................................................................. 8
   7.4 Unit Resolution .................................................................................................. 9
   7.5 Factoring ........................................................................................................... 10
   7.6 Paramodulation ................................................................................................. 10
8. Subsumption ............................................................................................................. 12
9. Demodulation .......................................................................................................... 13
   9.1 Special Function Evaluation ............................................................................. 15
10. A Reference Manual for the Options ....................................................................... 17
   10.1. Inference Rules ............................................................................................... 18
   10.2. Options for Processing Generated Clauses .................................................... 19
   10.3. Paramodulation Options ................................................................................ 21
   10.4. Unused Option Family ................................................................................... 22
   10.5. Miscellaneous Options .................................................................................. 22
   10.6. I/O Language Options .................................................................................. 23
   10.7. Weighting Options ......................................................................................... 24
11. The Rest of the Commands ..................................................................................... 24
12. Weighting ............................................................................................................... 25
   12.1. Weighting Parameter Sets ............................................................................. 25
   12.2. Weighting Without Patterns .......................................................................... 26
   12.3. Weighting With Patterns ............................................................................... 28
   12.4. Use of Weighting Options ............................................................................. 29

References .................................................................................................................... 29
An LMA-Based Theorem Prover

Ewing L. Lusk
Ross A. Overbeek

ABSTRACT

We describe here a theorem prover constructed from the facilities provided by Logic Machine Architecture (LMA). This program is not part of LMA itself, but illustrates the level of inference-based system which can be constructed from the LMA package of tools. It is a clause-based theorem prover supporting a wide variety of techniques which have proven valuable over the years in a long-running automated deduction research project. In addition, it is designed to present a convenient, interactive interface to its user which includes a number of useful utility commands.

1. Introduction

Logic Machine Architecture (LMA) was defined in [7] and [8] as a layered collection of subroutines for the construction of inference-based systems. The precise definition of each routine is given in [9].

In this document we describe the first major system constructed from the LMA tools. It is a clause-based theorem prover which incorporates the results of a long-running research project in automated deduction [6, 10, 11, 12, 13, 15, 14, 20, 21, 23, 24]. It also provides to its user a convenient, interactive interface which facilitates experimentation with the system. Like LMA itself, it is written in an extremely portable dialect of PASCAL, and has run on such systems as VAX/UNIX, VAX/VMS, IBM/CMS, Perq, and Apollo. Because of its interactive user interface we call it intthp (pronounced "int-T-H-P"). This does not mean that it must be guided interactively in its search for a proof, only that it need not be run in batch mode.

The basic item managed by intthp is a clause. Roughly speaking, a clause corresponds to a statement. It is a disjunction of a set of literals, which are assertions or denials of facts. A theorem prover derives new clauses from existing clauses using one or more inference rules. A basic reference on theorem proving is [1]. Alternatives are [5] and [18]. A brief overview of terminology is given in [21]. There is also some introductory material in this document.

In order to use intthp, a user need not be familiar with LMA. In LMA terminology, intthp is a layer three program. Most of its own function is in providing the user interface. Database management of facts and inference procedures are provided to it by LMA routines in the lower layers.
the user interface. Database management of facts and inference procedures are provided to it by LMA routines in the lower layers.

2. Basic Operation of the System

Central to the operation of intthp are four lists of clauses: the axioms list, the set of support list, the have-been-given list, and the demodulator list. Each of these plays a specific role in the fundamental operation of the theorem prover, which is repeated many times in the course of one execution of the program. The fundamental operation consists of the following steps:

1. Choose a clause from the set of support list. Call this clause "the given clause."
2. Infer a set of clauses that have the given clause as one parent and other parent clauses selected from the axioms list, the have-been-given list, and the demodulator list.
3. For each generated clause, "process" it (i.e., simplify it, perform subsumption checks, etc.).
4. Move the given clause from the set of the set of support list to the have-been-given list.

Very roughly speaking, a typical theorem-proving run consists of repeated execution of these four steps until either the set of support list has become exhausted or a contradiction has been found. The exact way in which steps 1, 2, and 3 (such as how the given clause is picked or which inference rules are used) is governed by user-controlled options, discussed in detail below.

3. Getting Started

In this section we will present the minimal set of commands necessary to run intthp. The full set of commands will be presented in sections to follow.

3.1. Format of Clauses

A variety of formats are supported for input and output of clauses. To begin with we will discuss only the default format. Other formats will be presented in the section below on input and output languages.

Constants are represented by strings of characters beginning with a letter between a and r or between A and Z, and containing no symbol from the set {():;&}. Variables are represented by strings beginning with a letter between s and z, inclusive, or with the "%" or the ".'" symbol. Function and predicate symbols should under ordinary circumstances follow the rules for constants. The negation symbol for a predicate is "-". Thus the following are valid literals:

\[
P \quad Q(x) \quad -R(f(x,a),b).
\]

The standard format of a clause uses the "|" symbol to represent the implied "OR" between literals. A clause may also be written in "if-then" format, with a conjunction in the "if" part and a disjunction in the "then" part. Variable names are not maintained internally; on input they are converted into variable numbers, which are then seen when the clause is output. For example, a clause may be entered in the form:

\[
\text{if } P(x,y) \& Q(x,y) \text{ then } R(x);
\]

but if it is subsequently displayed it will appear in its standard form:

\[
-P(x1,x2) \mid -Q(x1,x2) \mid R(x1)
\]
3.2. Some Elementary Commands

The interactive interface to intthp has the following structure. At the highest level, the prompt is "?". At this level, intthp is expecting a one-character command. The processes initiated by these commands may themselves be interactive; their prompts are designed to be as self-explanatory as possible. In this section we describe a small subset of the available commands. They will be further illustrated in the sample session in the next section.

- **a** (add a clause) prompts for a clause to be entered. The user then enters a clause, which may extend over several lines, terminated by `;` semicolon. The user is asked which list the clause should be added to, and should respond with one of the following characters:
  - `a` - to add the new clause to the axioms
  - `s` - to add the new clause to the set of support
  - `d` - to add the new clause to the demodulator list.

  Control is then returned to the highest level.

- **d** (delete a clause) prompts for the identifier of the clause to be deleted. If the user supplies the identifier of an existing clause, that clause will be deleted from all the lists it is in and from the clause space.

- **l** (list clauses) prompts for a list to be displayed. Valid responses are:
  - `a` - list the axioms
  - `s` - list the set of support
  - `h` - list the have-been-given list
  - `d` - list the demodulator list
  - `t` - list all the lists. Each list will be labeled as it is displayed.

- **n** (next inference) causes a clause to be selected, called "the given clause," from the set of support and as many new clauses to be derived from it as possible according to the current setting of various options. After each new clause is generated, the user will be asked whether or not he wants to keep it. He may then respond
  - `y` - the new clause will be added to the end of the set of support
  - `n` - the new clause will be discarded
  - `s` - the new clause will be discarded and generation of new clauses from the current given clause will be stopped. The given clause will be moved from the set of support to the end of the have-been-given list.

  If either "y" or "n" is entered, the process of generating new clauses from the given clause and prompting for whether or not to keep it will continue until no more clauses can be generated. Then the given clause will be moved to the have-been-given list. The "s" response aborts this process. If the newly generated clause is a unit equality clause, the user is asked whether or not it should become a demodulator. If he responds "y" then the clause is added to the demodulator list (as well as to the set of support) and back demodulation (see section below on demodulation) takes place.

- **q** (quit) exits from intthp.

These basic commands are enough to make use of intthp. In the next section we will use them in a sample session.
4. A Sample Session

To illustrate the above commands we present here a sample session. The prompts from intthp are shown in boldface type. The material in braces is a series of comments on the commands and is not part of the dialogue itself. At the beginning we suppose that you have invoked intthp according to the mechanism at your installation. It will respond with a date indicating the version you have installed and then with its highest-level prompt, which is a question mark. The session might then proceed as follows:

? a  \{We want to add a clause\}
clause > if Man(x) then Mortal(x); \{All men are mortal\}
add to axioms, set of support, or demodulators? a \{axioms\}
? a  \{We want to add another clause\}
clause > Man(Socrates); \{Socrates is a man\}
add to axioms, set of support, or demodulators? s \{set of support\}
? 1  \{We want to list the current clauses\}
axioms, set of support, have been given, or demodulators? t \{all of them\}
axioms:
1 -Man(x1) | Mortal(x1); \{Our first clause in normal form\}
support:
2 Man(Socrates); \{Our second clause\}
have been given:
3 Mortal(Socrates); \{New clause is kept\}
demodulators:
? n  \{We ask the system to make a deduction\}
given clause is: 2 Man(Socrates);
generated by hyperresolution: Mortal(Socrates); ancestors: 2 1
weight of new clause is 1 \{This will be explained later\}
keep this clause? y  \{We want to keep the new clause\}
3 Mortal(Socrates); ancestors: 2 1 \{exit from intthp\}

5. Some More Commands

A few more commands are described in this section. They make it easier to manage longer sessions and to interrupt sessions and then restore them.

h (help) displays a list of available commands and their functions. If the file "morehelp" is accessible to the program, then the user will be prompted for a command about which more information is desired. All lines in the file "morehelp" which begin with the command letter will be displayed, with a wait for a carriage return between screenfuls if necessary. By modifying this file, the user may add his own comments to the help file.

g (go) is a non-interactive form of the n command. It first prompts for a number of given clauses to use, then runs non-interactively until either that number of given clauses has been used or the set of support has been exhausted. If zero is entered in response to the prompt, then intthp will execute its basic loop until the set of support is exhausted. All generated clauses will automatically be kept unless forward subsumed or deemed too heavy (see section on weighting, below). All equality clauses will automatically become left-to-right demodulators.
(purge) removes all existing clauses from the clause space. They will be gone without a trace except that new clauses will be assigned identifiers greater than those of the purged clauses.

(save) saves the current clause space in a file, whose name is prompted for. It can be then be restored using the r command.

(restore) restores the clause space from a file which is in the format created by the s command. This format consists of four concatenated lists of clauses. A list consists of a set of clauses terminated by an empty clause (";"); only). When these lists of clauses are read in, they are assigned to the axioms, set of support, have been given, and demodulators, respectively.

(initialize lists) reads in clauses from files, one list at a time. File names are prompted for separately, one for axioms, one for set of support, and one for demodulators. The lists need not be empty when this command is invoked; the clauses read in are appended to the ends of the named lists.

6. Introduction to Options

The behavior of intthp is controlled by a collection of about fifty parameters called options. These range from fundamental choices, such as which inference rules are to be used, to such choices as whether or not to print the weight of each given clause as it is selected. Reasonable defaults exist for all of the options, and it is possible to run intthp without any knowledge of the options at all.

A complete guide to the options will be given below. Since many of them involve control of basic inference processing, we defer the details until after we have discussed these concepts in the next few sections. In this section we will describe the commands for maintaining the options. They consist of the commands for inspecting and modifying the current option settings and for saving and restoring a collection of option settings in a file.

The options are divided into families of (relatively) related options. One manipulates the options one family at a time, through menus displayed by intthp. A file of saved options is not in a form conveniently read by a user; it is anticipated that intthp will always be used to manage these files. The option management commands are:

(options) displays a menu of option families. To select one, type its number. You will then be presented with a submenu appropriate to that particular family. Several families may be inspected and modified with one invocation of the o command. To return to the highest level, enter a null line (carriage return only) in response to the "option family >" prompt. Each of the submenus lists its options and their current settings and gives a "modify >" prompt. To change an option, type its number. You will be prompted for a new value, and then for another option to modify. A null line returns the user to the "option family >" prompt, from which he can exit back to the highest level.

(save options; we ran out of mnemonics for the commands) prompts for a file name and then saves the current settings of the options in the named file, from which they can be restored using the k command.

(restore options) prompts for a file name and restores the options settings from that file. It must have been created using the j command.

Note: The r, i, and k commands all ask for names of existing files. If the file is not present, the user is assumed to have mistyped the file name, and is
prompted to enter it again. A carriage return in response to this prompt will abort this process and return control to the highest level.

The next three sections provide an introduction to the basic inference mechanisms used in intthp. The reader who is familiar with theorem-proving concepts and terminology may skip over them to the section which presents the options in more detail.

7. Inference Rules

In this section we describe the mechanisms which can be called upon to deduce new facts from existing ones. We assume that the reader knows what the expressions clause, literal, and term mean[1,4]. Inference rules are processes for producing new clauses from existing clauses. LMA supports a wide variety of inference rules. A key to effective use of the system is knowing which inference rules to apply in a given situation.

7.1. Hyperresolution

The most straightforward type of logical deduction is the following:

\[
\text{if } P \text{ then } Q \\
P \\
\hline
\text{therefore } Q
\]

In clause form this becomes

\[
-P \text{ or } Q \\
P \\
\hline
\text{therefore } Q
\]

The new clause, \( Q \), is formed from the clauses ( \(-P \text{ or } Q\) ) and \( P \) by clashing the literal \(-P\) against the literal \( P\). A more general form of this pattern occurs when there are more hypotheses in the "if-then" statement. A sentence like

\[
\text{if } P \text{ and } Q \text{ and } R, \text{ then } S
\]

becomes, when rendered into clausal form,

\[
-P \text{ or } -Q \text{ or } -R \text{ or } S
\]

We can deduce \( S \) if all of \( P, Q, \) and \( R \) are known to be true. Therefore from the four clauses

\[
-P \text{ or } -Q \text{ or } -R \text{ or } S \\
P \\
Q \\
R
\]

we can deduce \( S \). This is the pattern of deduction used in production systems and many of the systems described as "rule-based" systems.

Of course the literals in the above clauses may contain variables which may require instantiation in order for clashes to occur. For example, the sentence

"All men are mortal"

becomes
"Either x is not a man or x is mortal."

If we also know that

Socrates is a man

then we can deduce

"Socrates is mortal."

This is an example of the pattern:

\[-P(x) \text{ or } Q(x)\]
\[P(a)\]

therefore \[Q(a).\]

Hyperresolution is an inference rule which encompasses the above cases and more\[13, 16, 19, 20\]. It generalizes them in two ways. First, the "if-then" clause may have more than one conclusion literal. The clause

if \( P \) and \( Q \) then \( R \) or \( S \)

becomes

\[-P \text{ or } -Q \text{ or } R \text{ or } S.\]

Secondly, the clauses which contain literals which clash against the hypothesis literals in the "if-then" clause can have more than one literal, as long as all their literals are positive. A typical pattern might be:

\[-P \text{ or } -Q \text{ or } -R \text{ or } S\]
\[P \text{ or } T\]
\[Q \text{ or } W\]
\[R\]

\[T \text{ or } W \text{ or } S.\]

Note that hyperresolution requires that all of the negative literals in the "if-then" clause be clashed against corresponding literals in other clauses. For example, from

\[-P \text{ or } -Q \text{ or } -R\]

and

\[P \text{ or } S\]

hyperresolution would not deduce

\[S \text{ or } -Q \text{ or } R\]

(although binary resolution, described below, would do so). When variables are present their instantiations must be consistent. For example, from

\[-P(x, y) \text{ or } -Q(x) \text{ or } R(x, y)\]
\[P(z, b)\]
\[Q(a)\]

hyperresolution deduces

\[R(a, b).\]
Hyperresolution is perhaps the most commonly used inference rule in situations where equality substitutions do not play a major role. It corresponds to a natural mode of human reasoning. Its restriction that all negative literals must be clashed corresponds to the rule: "Don't draw any conclusions until all of the hypotheses are satisfied."

For a wide class of reasoning problems, hyperresolution is sufficient. It is the rule which most resembles the inference mechanism used in production systems. In intthp, hyperresolution is the default inference rule.

7.2. UR-Resolution

It is not hard to see that the use of hyperresolution by itself will lead to the derivation of clauses with only positive literals in them. While this is sufficient for a large class of problems, a number of reasoning tasks require the derivation of clauses containing negative literals.

Rather than abandon all restrictions on what kinds of clauses are allowed to be derived, we consider the desirability of clauses containing only one literal. Such clauses are called unit clauses or units. A unit clause can be regarded as a statement of fact, whereas multi-literal clauses represent conditional statements (if they contain both positive and negative literals) or statements of alternatives. Unit clauses are therefore more desirable in many situations. UR-resolution (Unit-Resulting resolution) [11] removes the restriction that derived clauses must have only positive literals, but imposes the restriction that derived clauses must be units. For example, from

\[-P \text{ or } -Q \text{ or } R \]
\[P\]
\[-R\]

UR-resolution would derive \(-Q\), whereas hyperresolution would be unable to derive anything. UR-resolution emphasizes units in another way as well: all but one of the clauses which participate in the deduction must be unit clauses, although they can be either positive or negative. One might say that UR-resolution emphasizes unit clauses in exactly the same way that hyperresolution emphasizes positive clauses. With variables present, another example might be:

\[P(x,y) \text{ or } -Q(a) \text{ or } R(x,z)\]
\[Q(x)\]
\[-R(b,c)\]

\[P(b,y).\]

7.3. Binary Resolution

Both hyperresolution and UR-resolution derive much of their power from the fact that many clauses can participate in the clash, which corresponds to taking several reasoning steps at once. Very occasionally it is necessary to employ resolution in very small steps. The form of resolution used in this case is called binary resolution; it corresponds to the smallest possible deductive step.

The only "restriction" on binary resolution is that exactly two clauses may participate in the clash[17]. Since both hyperresolution and UR-resolution can be thought of as sequences of binary resolutions, this is really not a restriction. An example might be:
Notice that this result could not have been obtained by hyperresolution (since it is not positive) nor by UR-resolution (since it is not a unit). However, any hyperresolvent or UR-resolvent can be obtained (eventually) by binary resolution. For example, the hyperresolution:

\[-P \lor -Q \lor R\]
\[P \lor S\]
\[Q\]

\[\underline{\neg P \lor \neg S \lor R}\]

\[\neg P \lor \neg Q \lor S\]

\[\neg Q \lor R \lor S\]
\[Q\]
\[\underline{\neg P \lor \neg Q \lor R} \lor \neg R\]

\[\neg P \lor -Q \lor R\]
\[P \lor S\]
\[Q\]

\[\underline{\neg P \lor \neg Q \lor -Q \lor S} \lor \neg R\]

\[\neg P \lor -Q \lor R\]
\[P \lor S\]
\[Q\]
\[\underline{\neg Q \lor R \lor S} \lor \neg Q\]

\[\neg P \lor -Q \lor R \lor T \lor -U \lor V\]

This derived clause will be shorter than the clauses which produced them. For example,

\[-P \lor -Q \lor R\]
\[P \lor S\]
\[Q\]
\[\underline{\neg Q \lor R \lor S} \lor \neg Q\]

\[\neg P \lor -Q \lor R \lor T \lor -U \lor V\]

It is easy to see how unrestricted use of binary resolution can lead to a very large collection of very weak clauses. (A clause having many literals can be thought of as making a weaker statement than one with few literals.)

7.4. Unit Resolution

One restriction that is sometimes placed on binary resolution is the requirement that one of the two clauses involved in the clash be a unit[2, 22]. The motivation for this restriction is that if one clause is a unit then the resulting resolvent will consist of the other participating clause with one of its literals removed (and perhaps some of its variables instantiated). Thus derived clauses will be shorter than the clauses which produced them. For example,

\[-P \lor -Q \lor R \lor S\]
\[R\]

\[\underline{\neg P \lor -Q \lor R} \lor \neg S\]

Or, with variables present.
These are not the only resolution-based inference rules supported by intthp, but they do represent the ones most often used. The complete set can be found in option family one.

### 7.5. Factoring

There is one inference rule that derives new clauses from a single clause, rather than from pairs of clauses. It is called **factoring**, and involves the unification of literals within the same clause\[1, 4\]. For example,

\[-P(x,y) \text{ or } Q(f(x),b) \text{ or } -R(x,c)
R(a,z)
\]

\[-P(a,y) \text{ or } Q(f(a),b).
\]

The new clause is said to be a **factor** of the original one.

Factoring is important because without it the resolution rules described above are incomplete, which means that given a set of contradictory clauses, a contradiction may not be derived. The classical example is:

\[P(x) \text{ or } P(x)
-P(x) \text{ or } -P(x)
\]

This set of clauses is contradictory, since \(P(x)\) is a factor of the first clause and \(-P(x)\) is a factor of the second clause. But without factoring, a rule like binary resolution will only derive the tautology \(P(x) \text{ or } -P(x)\). In intthp, factoring can be invoked in two ways: as automatic processing of newly generated clauses or as an inference rule in its own right.

### 7.6. Paramodulation

The next inference rule we consider is not based on resolution at all. Instead, it is based on the substitution properties of the equality relation. For example, if we know that John's wife is sick, and that John's wife is Sue, then we know that Sue is s.ck. This is an instance of the pattern

\[Equal(a,b)
\]

\[P(b).
\]

In this example, the result \(P(b)\) is called a **paramodulant** rather than a resolvent\[15, 3, 14\]. The clause \(P(b)\) is said to be obtained by paramodulating into the clause \(P(a)\) from the equality clause \(Equal(a,b)\). The terms in the from clause and in the into clause are identical in the above example, but in general are only required to be unifiable. Here is an example in which a substitution must be made in the into clause:
and here is one in which the substitution must be made in the from clause:

\[ \text{P}(g(a), b) \]
\[ \text{Equal}(g(x), x) \]
\[ \text{P}(a, b). \]

Sometimes, substitutions are made in both terms:

\[ \text{P}(f(a, x), x) \]
\[ \text{Equal}(f(y, b), y) \]
\[ \text{P}(a, b). \]

In the above examples, both the into and from clauses are units, but this is not a requirement for paramodulation. For example,

\[ \text{P}(f(x, g(y))) \text{ or } \text{Q}(x, y) \]
\[ \text{Equal}(f(a, g(b)), c) \]
\[ \text{P}(c) \text{ or } \text{Q}(a, b). \]

Note that, as usual, when a substitution is made for a variable, it must be made for all occurrences of the variable in the clause. The from clause can also have extra literals:

\[ \text{P}(f(a, x)) \]
\[ \text{Equal}(f(y, b), c) \text{ or } \text{Q}(y) \]
\[ \text{P}(c) \text{ or } \text{Q}(a). \]

The expressions into and from can also refer to the terms being matched as well as to the clauses in which they occur. In the above example, one would say that paramodulation occurred from the term \( f(y, b) \) into the term \( f(a, x) \).

The terms paramodulated into or from may even be variables, although this is sometimes considered undesirable. An example of paramodulation into a variable would be

\[ \text{P}(f(x), x) \]
\[ \text{Equal}(g(b), h(a)) \]
\[ \text{P}(f(h(a)), h(a)). \]

and an example of paramodulating from a variable would be
Various kinds of restrictions are sometimes imposed on paramodulation. These include blocking paramodulation into variables or from variables, and restricting the from term to be either the left or right hand side of the equality literal. In the above examples, the left hand side was always used as the from term, but this is not necessary. In the following example, we are not paramodulating from the variable, but rather from the right hand side of the equality.

\[
P(f(a,a)) \text{ or } Q(x) \\
\text{Equal}(y,f(y,y))
\]

\[
P(a) \text{ or } Q(a)
\]

Another type of restriction limits the kinds of substitutions that are allowed. For example, one might require that the into term be an instance of the from term, or that the from term be an instance of the into term. In non-complexifying paramodulation, variables in the into term can only be replaced by other variables or constants, unless they occur nowhere else in the into clause. All of these variations are supported by intthp and controlled by the paramodulation family of options.

8. Subsumption

What Subsumption Means

Subsumption is the mechanism by which unnecessary clauses are discarded [1, 4, 17]. The simplest situation occurs when a clause is derived which is already present in the clause space. In this case we want to discard the newly derived clause.

More generally, the newly derived clause may be recognizably less general than some existing clause without being identical to it. There are two basic ways this can happen.

The first is that the literals of the new clause may form a subset of the literals of the existing clause. For example, if we already know

\[
P \text{ or } Q
\]

and derive

\[
Q \text{ or } S \text{ or } P
\]

then we may discard the new clause, since it is logically weaker than the original one.

The second is that the new clause may be an instance of the original clause. For example,

Old clause: \( P(a,x) \)

New clause: \( P(a,b) \)

Since any resolution in which the new clause might participate will occur with the old clause anyway, we discard the new clause.
These two ways in which a new clause may be less general than an existing clause may of course be combined. For example,

Old clause: \( P(a,x) \) or \( Q(y) \) or \( P(y,b) \)
New clause: \( Q(a) \) or \( P(a,b) \)

So in general, clause \( A \) subsumes clause \( B \) if there is a substitution for the variables in clause \( A \) such that after the substitution, the literals of clause \( B \) form a subset of the literals of clause \( A \).

This process of discarding new clauses which are subsumed by existing clauses is called \textit{forward subsumption}. The subsumption process also can occur in the opposite direction. That is, a newly derived clause may subsume one or more existing clauses in which case we probably want to keep the new clause and discard the subsumed clauses. This process is called \textit{backward subsumption}.

In intthp, forward and backward subsumption is done for all newly generated clauses, as part of default processing. The user can optionally turn off either one or both.

Note that under ordinary circumstances any factor of a clause would be subsumed by its parent. To prevent this, intthp utilizes an option available within LMA to prevent any clause from subsuming another clause shorter than itself.

9. Demodulation

The Meaning of Demodulation

\textit{Demodulation} is the process of rewriting a clause in place using an equality substitution\cite{23}. The rewriting is controlled by unit equality clauses called \textit{demodulators}. In intthp, demodulators are exactly those unit equality clauses on the demodulator list. Clauses derived during the run may also become demodulators, and be dynamically added to the demodulator list. For example,

\begin{align*}
P(f(a),b) \\
\text{Equal}(f(a),c)
\end{align*}

\[ P(c,b) \]

The clause \( P(c,b) \), called a \textit{demodulant}, replaces the existing clause \( P(f(a),b) \), which is deleted. (The clause \( P(c,b) \) could also be derived by paramodulation, but the parent clause would not be deleted.)

Variables may be present in the demodulators, and in the clauses they demodulate, but instantiation of variables can occur only in the term in the equality. For example,

\begin{align*}
P(f(g(a)),g(a)) \\
\text{Equal}(f(g(x)),h(x))
\end{align*}

\[ P(f(h(a)),g(a)). \]

The demodulated clause need not be a ground clause (that is, it may contain variables):
In general, one can specify that a demodulator apply left-to-right, right-to-left, or either way. In LMA, a user variable in the demodulator controls the direction of demodulation.

In the presence of multiple demodulators, many may apply, and each may apply more than once. For example,

\begin{align*}
P(f(g(a)), f(a)) \\
\text{Equal}(g(x), h(x)) & \quad \text{(left-to-right)} \\
\text{Equal}(a, h(a)) & \quad \text{(right-to-left)} \\
\text{Equal}(f(a), b) & \quad \text{(left-to-right)}
\end{align*}

\[ P(b, b). \]

Since a demodulator may apply more than once, it is possible for looping to occur\cite{[12]}. This possibility occurs naturally in demodulators which express commutativity, such as

\[ \text{Equal}(f(x, y), f(y, x)) \]

In the presence of this demodulator, a clause like \( P(f(a, b)) \) would demodulate to \( P(f(b, a)) \), then to \( P(f(a, b)) \), then \( P(f(b, a)) \), etc. This is prevented in the following way.

When a clause is designated as an "either-way" demodulator, then whether it is applied or not depends on the lexical ordering of the instantiations of its variables. Lexical ordering of symbols can be allowed to default, or can be specified by use of the LEX predicate. Depending on the lexical ordering of \( a \) and \( b \), the demodulator

\[ \text{Equal}(f(x, y), f(y, x)) \]

will demodulate \( P(f(a, b)) \) to \( P(f(b, a)) \) or leave it unchanged. In this way canonical forms for expressions can be maintained.

When existing demodulators are applied to a newly derived clause, the process is called forward demodulation. It is also possible for new demodulators to be added to the clause space, in which case one may want to apply them to some or all of the existing clauses in the clause space. This process is called back demodulation. An example would be the following situation. Suppose the set of existing clauses contains

\begin{align*}
P(f(h(a))) \\
\text{Equal}(f(b), c)
\end{align*}

and the new demodulator

\[ \text{Equal}(h(a), b) \]

is derived. Then by back demodulation the clause

\[ P(f(b)) \]

is derived, which immediately demodulates to

\[ P(c). \]
The clause $P(f(h(a)))$ is replaced by $P(c)$.

9.1. Special Function Evaluation

In forming the demodulant of a clause we not only apply equality transformations - we also perform "function evaluations". For example, $SSUM(1,1)$ would be rewritten as 2, even though no demodulator existed to cause the reduction.

To understand the behavior of the demodulation-simplification routine, it is necessary to understand the meanings attached to the following system-defined symbols:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SSUM(n1,n2)$</td>
<td>if $n1$ and $n2$ are self-defining numeric values, this simplifies to $n1+n2$</td>
</tr>
<tr>
<td>$SNEG(n1)$</td>
<td>if $n1$ is a self-defining numeric value, this simplifies to $-n1$</td>
</tr>
<tr>
<td>$SPROD(n1,n2)$</td>
<td>if $n1$ and $n2$ are self-defining numeric values, this simplifies to $n1*n2$.</td>
</tr>
<tr>
<td>$SDIV(n1,n2)$</td>
<td>if $n1$ and $n2$ are self-defining numeric values, and if $n2 &lt;&gt; 0$, then this evaluates to $n1/n2$</td>
</tr>
<tr>
<td>$SMOD(n1,n2)$</td>
<td>if $n1$ and $n2$ are self-defining integers, then this evaluates to $n1$ modulo $n2$</td>
</tr>
<tr>
<td>$SCOMP(n1,n2)$</td>
<td>if $n1$ and $n2$ are ground values, then this evaluates to</td>
</tr>
<tr>
<td></td>
<td>0 if $n1=n2$</td>
</tr>
<tr>
<td></td>
<td>1 if $n1&lt;n2$</td>
</tr>
<tr>
<td></td>
<td>2 if $n1&gt;n2$</td>
</tr>
<tr>
<td>$SOUT(t)$</td>
<td>if this occurs in a unit clause, and $t$ is ground (contains no variables), $t$ is written to the terminal and this evaluates to NIL</td>
</tr>
<tr>
<td>$SIN$</td>
<td>if this occurs in a unit clause, this evaluates to an object entered from the terminal</td>
</tr>
<tr>
<td>$SOUTIN(t)$</td>
<td>if this occurs in a unit clause, and if $t$ is ground, then $t$ is written to the terminal and the whole term is replaced with an object entered from the terminal</td>
</tr>
<tr>
<td>$SCHR(n)$</td>
<td>this symbol is only evaluated when a $SOUT$ or a $SOUTIN$ causes something to be written to the terminal. In that case this expression evaluates to &quot;chr(n)&quot;, the ASCII character represented by the value $n$.</td>
</tr>
</tbody>
</table>
This expression evaluates only if it is ground. In that case it evaluates to TRUE if \( t_1 > t_2 \). Else, it evaluates to FALSE.

This expression evaluates only if it is ground. In that case it evaluates to TRUE if \( t_1 \geq t_2 \). Else, it evaluates to FALSE.

This expression evaluates only if it is ground. In that case it evaluates to TRUE if \( t_1 < t_2 \). Else, it evaluates to FALSE.

This expression evaluates only if it is ground. In that case it evaluates to TRUE if \( t_1 = t_2 \). Else, it evaluates to FALSE.

This expression evaluates only if it is ground. In that case it evaluates to TRUE if \( t_1 \neq t_2 \). Else, it evaluates to FALSE.

This expression evaluates to FALSE.

This expression evaluates to TRUE.

Besides the above, the following special symbols have been defined:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\leq(t_1,t_2)$</td>
<td>This expression evaluates only if it is ground. In that case it evaluates to TRUE if ( t_1 \leq t_2 ). Else, it evaluates to FALSE.</td>
</tr>
<tr>
<td>$\geq(t_1,t_2)$</td>
<td>This expression evaluates only if it is ground. In that case it evaluates to TRUE if ( t_1 \geq t_2 ). Else, it evaluates to FALSE.</td>
</tr>
<tr>
<td>$\lt(t_1,t_2)$</td>
<td>This expression evaluates only if it is ground. In that case it evaluates to TRUE if ( t_1 &lt; t_2 ). Else, it evaluates to FALSE.</td>
</tr>
<tr>
<td>$\neq(t_1,t_2)$</td>
<td>This expression evaluates only if it is ground. In that case it evaluates to TRUE if ( t_1 \neq t_2 ). Else, it evaluates to FALSE.</td>
</tr>
</tbody>
</table>
Demodulation is routinely invoked in intthp as part of the standard processing of newly generated clauses, although it can be turned off. Forward and backward demodulation can also be specified as inference rules. In that situation forward demodulation means "apply all existing demodulators to the given clause" and backward demodulation means "if the given clause is a demodulator, apply it to all existing clauses."

10. A Reference Manual for the Options

In this section we discuss each of the currently available options, give their defaults, and describe the effects of alternative settings.

When the o command is invoked, it presents the following menu:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIL</td>
<td>used to mark the end of lists</td>
</tr>
<tr>
<td>C</td>
<td>used as &quot;concatenate&quot;, a binary operator to form lists</td>
</tr>
<tr>
<td>$JUNK</td>
<td>any clause that contains this symbol will evaluate to TRUE, if simplified</td>
</tr>
<tr>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>currently not used in simplification</td>
</tr>
<tr>
<td>OR</td>
<td>used (along with NOT) in the representation of clauses</td>
</tr>
<tr>
<td>0</td>
<td>the constant 0</td>
</tr>
<tr>
<td>$LT0</td>
<td>$LT0(t) means &quot;t is less than 0&quot;</td>
</tr>
<tr>
<td>$LTE</td>
<td>$LTE0(t) means &quot;t is less than or equal to 0&quot;</td>
</tr>
<tr>
<td>$CURVARS</td>
<td>$CURVARS(C(v1,C(v2, ...))) means that v1,v2,... are &quot;current variables&quot;, a concept that is used occasionally in Bledsoe's inequality manipulation rules</td>
</tr>
</tbody>
</table>
Select option family:
1 - inference rules
2 - generated clause processing options
3 - paramodulation options
4 - unused option family
5 - miscellaneous options
6 - I/O language options
7 - weighting options
option family >

One then enters the number corresponding to the family of options one wants to examine and (optionally) modify. The various families are now described. The menus given at the beginning of each section show the default setting of each option.

10.1. Inference Rules

If a "1" is entered, intthp will display the following menu:

Inference Rules:

1. Hyperresolution ......................... y 6. Unit resolution......................... n
2. Ur-resolution ........................... n 7. Unused ................................. n
3. Paramodulation into given clause ... n 8. Forward demodulation ................. n
4. Paramodulation from given clause .. n 9. Back demodulation .................... n
5. Binary resolution ...................... n 10. Factoring .............................. n
11. Unit deletion........................... n
modify >

This is a list of the inference rules which will be used by the n and g commands. A "y" means that a rule will be used; an "n" means it will not be. More than one inference rule may be in effect at once, in which case clauses will be generated from the given clause using inference rules in increasing order of the numbers given above. Here is a brief description of each inference rule. For more details, see the section above on inference rules.

1. Hyperresolution - Multiple positive clauses, called satellites, are clashed against the negative literals in a single clause, called the nucleus. The result must be positive. The given clause may be either the nucleus or a satellite.

2. UR-resolution - Multiple unit clauses, called satellites, are clashed against all but one of the literals in a non-unit clause, called the nucleus. The result must be a unit clause. The given clause may be either the nucleus or a satellite.

3. Paramodulation into given clause - Paramodulation occurs with the given clause as the into clause; that is, the equality clause which is the basis for the substitution is not the given clause. See the subsection on paramodulation in the above section on inference rules.
4. Paramodulation from given clause - The given clause must contain an equality literal. Paramodulation occurs from this clause into other clauses.

5. Binary resolution - The given clause is clashed against one other clause.

6. Unit resolution - One parent of the resolvent must be a unit. It may be either the given clause or another clause.

7. Not used - This value is not currently in use.

8. Forward demodulation - The given clause is demodulated, using all the equalities currently on the demodulator list. One would ordinarily use forward demodulation as an inference rule only when the simplification option in family two is turned off. When it is on forward demodulation of newly generated clauses occurs automatically.

9. Back demodulation - The given clause, which must be a unit equality clause, is used to back demodulate the clauses in the axioms, have been given, and demodulator lists. One would not ordinarily set this option unless automatic back demodulation has been turned off in family two.

10. Factoring - The given clause is factored. There is an option in family two to automatically factor newly generated clauses, but its default value is not to do this.

11. Unit deletion - This is an inference rule in which literals are removed from the given clause if they are instances of unit clauses in the clash lists. There is an option in family two to automatically apply this inference rule to newly generated clauses, but this is not the default.

In inthp the given clause is selected from the set of support. The other clauses participating in the clash must be on the axiom list, the have-been-given list, or the demodulator list. In particular, no two clauses on the set of support will participate in the same clash; one of them must first become the given clause, after which it will be moved to the have-been-given list.

10.2. Options for Processing Generated Clauses

If a "3" is entered in response to the "option family >" prompt, the following menu will be displayed:

Options for processing generated clauses:
1. Simplify new clauses ...................... y 9. Prepare log file........................n
2. Display derivation history ............... y 10. Name of log file ....................logfile
3. Automatically factor new clauses ....n 11. Forward subsumption check ......y
4. Limit on # of simplifications ........20 12. Backward subsumption check ......y
5. Test for unit conflict .................... y 13. Backward demodulation ............y
6. Maximum weight of new clause ......0 14. Print all generated clauses ........n
7. Auto invoke unit deletion ............n 15. Copy output to a script file ........n
8. Echo clauses read from files ........ y 16. Name of script file ...........scriptfile

17. Print weight of generated clauses......n

modify >

This family of options governs what is done with newly generated clauses immediately after they have been generated by one of the inference rules in...
1. Simplify new clauses - Newly generated clauses are processed by demodulation. This includes applying the demodulators in the demodulator list according to their "d" attribute (see below) and evaluation of special built-in functions such as $\$SUM$, etc.

2. Display derivation history - when clauses are displayed, include a list of the clauses from which they were derived. These are labeled as the "ancestors" of the generated clause.

3. Automatically factor new clauses - New clauses are factored.

4. Limit on # of simplifications - Since it is possible for demodulation to loop, this limit is set to abort the demodulation process after the specified number of demodulations has occurred. A message to the effect that this number was exceeded is printed when this occurs. Note that the default is quite low for some uses of demodulation.

5. Test for unit conflict - New unit clauses are tested to see whether they clash with other unit clauses in the clause space. This allows the last step of a proof by contradiction to be obtained without waiting for one of the two contradictory clauses to be chosen as the given clause.

6. Maximum weight of new clause - Each new clause is weighed (see section below on weighting) and if its weight is greater than this number, it is discarded. If the value of this parameter is zero, then no screening on the basis of weight is done.

7. Automatically invoke unit deletion - The unit deletion inference rule is automatically invoked on newly generated clauses to see whether any of the new clause's literals can be removed because they conflict with existing unit clauses. In such a conflict, the literal being removed from the newly generated clause must be an instance (but with opposite sign) of the existing unit clause.

8. Echo clauses read from files - Clauses which are read in using the r or i commands are echoed to the terminal.

9. Prepare log file - In order to use the p command for examining a proof, described below, a log file must be prepared. This option causes it to be prepared automatically. The name of the file is given by the following option. At the time this parameter is set to "y", the file is opened, so if the default log file name is not desired, the log file name must be set before this option is set.

10. Name of log file - This is the name of the file of inference information automatically written during a run if the above option is set to "y".

11. Forward subsumption check - Each new clause is tested to determine whether there is already in the clause space a clause which is more general than it (or identical to it). If so, the new clause is discarded.

12. Backward subsumption check - Each new clause is checked to see whether it is more general than an existing clause (or identical to it). If so, the existing clause is removed from the clause space.

13. Backward demodulation - New clauses are tested to see if they are unit equalities. If they are, they become demodulators and then are applied to all existing clauses in the clause space.
14. Print all generated clauses - Each newly generated clause is printed, even if it is destined to be subsumed or rejected as too heavy.

15. Copy output to a script file - Output which goes to the screen is copied to a file as well. The name of the file is given in the next option. Currently (11/15/82) this option is not implemented.

16. Name of script file - This is the name of the script file written if the above option is "y".

17. Print weight of generated clauses - As each new clause is generated, its weight is printed.

10.3. Paramodulation Options

There are many restrictions which can be placed on the paramodulation inference rule, having to do with whether or not variables can be paramodulated into or from, whether restrictions should be imposed on the substitution which unifies the into and from terms, and which side of the equality the from term can occur on. This family of options allows one to choose most useful combinations of these restrictions.

**Paramodulation Options:**

1. **Instantiation options**
   - 0 - both into and from can be instantiated
   - 1 - into term must be instance of equality argument
   - 2 - equality argument must be instance of into term
   - 3 - noncomplexifying paramodulation (into variables can be instantiated only to constants or variables, unless they occur nowhere else in the into clause)

2. **Into options**
   - 0 - any term is allowed
   - 1 - variables are not allowed
   - 2 - neither variables nor constants are allowed

3. **From options**
   - 0 - either argument of equality, no restriction
   - 1 - only left argument, no restriction
   - 2 - either argument, no variable
   - 3 - only left argument, no variable
   - 4 - either argument, no variable or constant
   - 5 - only left argument, no variable or constant

modify >

Paramodulation always involves the unification of two terms: the from term (the one in the equality clause, which is not affected by the paramodulation) and the into term. One often wishes to place restrictions on the substitution which unifies the from term and the into term. The instantiation options control this. The motivation for noncomplexifying paramodulation is that the result of such a paramodulation cannot be vastly more complex than the original into clause, since at most one occurrence of an into variable can be expanded.
Since unifying with a variable is "too easy," one often (but not always) wants to prevent paramodulations which result from such a unification. The into option controls the types of into terms that can be unified with, and the from option enables one to place similar restrictions on the from term. In addition, the from option controls which argument of the equality clause an be used as the from term.

10.4. Unused Option Family

Family four involves the options which control a feature which is still under development.

10.5. Miscellaneous Options

Several of the options do not logically belong together with any others, so they are lumped together in the family called "miscellaneous options."

Miscellaneous Options:

1. Qualification options.................0 5. Print given clause ....................y
2. Locking options......................0 6. Print weight of given clause .......n
3. Stop after # of null clauses ..........1 7. Level of unification properties ...2
4. How to pick given clause .............3 8. Use consecutive cl numbering ......y

1 - pick first clause on set of support
2 - pick last clause on set of support
3 - pick lightest clause on set of support

modify >

The meanings of these parameters are as follows:

1. Qualification options - These are not currently used in intthp, although they are implemented in LMA. See [20] for the theoretical foundation, [10] for motivation and examples, and [9] for the details of the LMA implementation.

2. Locking options - These are not currently used in intthp, although like the qualification options they are implemented in LMA and the details of what the various values mean can be found in [9]. The motivation is discussed in [10].

3. Stop after # of given clauses - Control will be returned to the highest level after this many null clauses have been generated. Usually the default of one is appropriate, but higher numbers are used when multiple proofs are desired.

4. How to pick given clause - Currently there are three different ways to choose the next given clause from the set of support. Always picking the first clause leads to a breadth-first search for a proof. Always picking the last clause (most recently added) leads to a depth-first search. The third alternative is to measure all the clauses on the set of support and try to pick the "best." The measurement mechanism is called weighting and is described below.

5. Print given clause - When the given clause is selected, print "given clause is: " followed by the selected clause. This occurs even if no clauses are generated.
6. Print weight of given clause - When the given clause is selected and printed, print its weight. In order for this to have any effect, the "print given clause" option must be on as well.

7. Level of unification properties - Search for relevant terms and literals is accomplished through an indexing scheme which maintains syntactic properties of terms and literals. The level of detail maintained by this index is variable, although it should not be varied during the course of a run. In the presence of many terms which are identical at the highest few levels of nesting (such as \( P(h(g(a))) \) and \( P(h(g(b))) \)), increasing this level will decrease the number of unifications which fail. The cost will be in memory utilization and in time spent in index management. Warning: a value of zero will result in no derived clauses, since the inference rules rely on the index to find the clauses they will interact with.

8. Use consecutive clause numbering - Clauses may be identified either by their LMA-maintained object identifiers, or by intthp-maintained clause numbers. The LMA identifiers are assigned to all clauses, literals, and terms, and so the ones assigned to clauses are not consecutive. The intthp clause numbers are assigned consecutively to clauses as they are integrated into the clause space.

10.6. I/O Language Options

It is relatively straightforward to install translators so that clauses are written in a form appropriate for a particular user or application. Currently three of the languages are only supported on UNIX, since their translators were constructed using the LEX/YACC tools for writing translators in C. In order to support them on a different system, a C compiler would be required which generates code callable from Pascal.

**I/O Language Options:**

<table>
<thead>
<tr>
<th>Input language</th>
<th>2</th>
<th>2. Output language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Portable format:</td>
<td>( (P(f(a))x) );</td>
<td>( P(f(a,x)) &amp; Q \text{ then } R(a) \mid T; )</td>
</tr>
<tr>
<td>2. Ifthen format:</td>
<td>( \text{ if } P(f(a,x)) &amp; Q \text{ then } R(a) \mid T; )</td>
<td>( \text{ if } x + a &gt; 5 \text{ then } Q(x); )</td>
</tr>
<tr>
<td>3. Infix format:</td>
<td>( \text{ if } x + a &gt; 5 \text{ then } Q(x); )</td>
<td>( \text{ if } x + a &gt; 5 \text{ then } Q(x); )</td>
</tr>
<tr>
<td>4. AURA format:</td>
<td>( \text{ if } x + a &gt; 5 \text{ then } Q(x); )</td>
<td>( \text{ if } x + a &gt; 5 \text{ then } Q(x); )</td>
</tr>
<tr>
<td>5. Circuit design:</td>
<td>( \text{ if } x + a &gt; 5 \text{ then } Q(x); )</td>
<td>( \text{ if } x + a &gt; 5 \text{ then } Q(x); )</td>
</tr>
</tbody>
</table>

**modify >**

Input and output languages are independently selectable. The above displayed examples are present as a reminder of what each language looks like.

1. Input language - This will be used for both terminal input (the `a` command) and file input (the `r` and `i` commands).

2. Output language - This will be used whenever clauses are written: both to the terminal during the course of a run and with the `s` command when clauses are written to a file.

The language options are:

1. Portable format - This is the "raw" format used by LMA itself[?]. It is the language which the other languages are translated into and out of.
2. If-then format - This is the format described in the "Getting Started" section of this paper. Literals are written in the usual predicate/function notation and the connectives "if...then...", "\&", and "\|"] are supported in a limited way. ( "\|"] only in conclusions, "\&" only in hypotheses of if...then... clauses.)

3. Infix format - This is an experimental LEX/YACC-based translator which includes infix representation of mathematical expressions and a very limited ability to translate English sentences.

4. AURA format - This is the standard input format for the AURA theorem prover[24]. Sets of clauses can be converted from one format to the other by choosing different input and output languages.

5. Circuit design - This is another experimental language, useful for expressing as clauses the tables use in that particular application.

10.7. Weighting Options

Family seven consists of the options which control the weighting mechanism. This is complicated enough that we defer discussion of how this submenu of options is managed until after an extended discussion of weighting in a section below.

11. The Rest of the Commands

In this section we present the rest of the highest level commands provided by intthp. The first three have to do with interactive management of the clause space; the others are special-purpose commands.

m (move) moves a clause from one list to another. First the system will prompt for the clause number, then for the list the clause is to be moved to. The clause will be added at the beginning of the target list. The clause will be removed from whatever lists it is in. Moving a clause to a list it is already on has the effect of moving it to the front of the list. (This is sometimes useful in causing a specific clause on the set of support to be used next.)

e (edit) displays certain tags which have been attached to a clause and allows the user to modify them. (These "tags" are the user variables and attributes described in [8].) Currently the only one managed by intthp is the demodulation attribute, which determine in which direction a demodulator acts: right-to-left, left-to-right, or either way. The user will be prompted for the clause identifier, after which the clause and its tags are displayed, followed by an "attribute to modify?" prompt. The prompts for each attribute are different. The "d" attribute prompts for direction: "l", "r", or "e". One common use of the e command is merely to list the clause with a given clause number.

t (report) supplies certain useful statistics gathered during a run, partly but not entirely for debugging purposes. The statistics are divided into levels, corresponding to the layers of LMA. Level 3 contains information on numbers of clauses generated and kept, with separate counts for each of the various reasons why they might be generated but not kept (forward subsumed, too heavy, etc.). Level 2 contains memory utilization statistics for some of the layer 2 data structures. Level 1 statistics include similar information, plus unification statistics. In response to entering a "t", the user is prompted for a level number. If a null line is entered, control returns to the highest level.
p (proof) displays the proof of a clause. Actually, the p command initiates an interactive process which allows one to examine a proof in detail. Note that an attempt to invoke the p command without having logged the inferences just made will have unpredictable results, since the p command begins execution by reading the file named as the current log file, specified in option family two.

f (fast fact finder) is somewhat like the g or n commands in that it initiates an inference process. It assumes that the set of support is currently empty, and prompts (with "> ") for a clause to be put in it. When a clause is entered (ending with a ;") it is as if the g command had been invoked with an infinite number of given clauses to be used. The basic loop is executed until the set of support is exhausted. The main difference is that no output is produced except that generated by the SOUT and SOUTIN functions (see [8]). It can thus be used to mimic certain types of consultation systems. To return control to the highest level, enter a null clause (";") only.

w (weigh) invokes an interactive process which allows one to examine the effects of the weighting parameter settings. It prompts for a clause and a weighting parameter set (see below) and displays the weight of that clause according to that weighting parameter set. It then prompts for literals, terms and subterms of that clause. At each level a null line will cause the level being examined to go up by one, so that a sequence of null lines returns control up to where a new clause can be weighed, then to where a new weighting parameter set can be loaded, and finally to the highest level.

z (zoo) is a special-purpose command for demonstration purposes. Who knows what will happen?

This completes the overview of the commands.

12. Weighting

Weighting[12] is a mechanism for assigning a number to a clause, literal, or term. This number can then be used for such things as determining whether to keep a newly derived clause, picking the next given clause, or deciding whether a newly derived equality should become a demodulator.

12.1. Weighting Parameter Sets

A collection of options called a weighting parameter set determines the weight of each clause, literal, and term. Because weights are used for varying purposes, three sets of weighting parameters are maintained. Currently (11/15/82) the first is used to determine whether or not to keep a clause derived by the g command, the second is used to determine the choice of next given clause, and the third is unused. One potential use is in determining whether a newly derived equality unit should become a demodulator.

Each weighting parameter set consists of sixteen real numbers and a list of patterns. The numbers are structured into three families of five plus one other number. These numbers describe how weights of clauses are built up from the weights of their component literals, weights of literals from the weights of their component predicates and arguments, and the weights of terms from the weights of their function symbols and subterms. The patterns describe how this weighting algorithm is to be bypassed to give special weights to certain classes of clauses, literals, or terms. We will discuss the algorithmic mechanism first and patterns later.
12.2. Weighting Without Patterns

Let us assume that the pattern list for the weighting parameter set we are interested in is empty (this is the default). Then the weight of a clause is calculated from the sixteen numbers in the weighting parameter set in the following way.

Constants and Variables

The weight of a constant is 1. The weight of a variable is the number entered and displayed as "variable weight." The default variable weight is 1.

Complex Terms

For each of clauses, literals, and terms, there is a set of five numbers which controls the way in which their weights are calculated from the weights of their components. The names of these numbers are #ARG, MAXARGWT, SUMARGWT, SYMCT, and BASE. They have slightly different meanings for clauses, literals, and terms. We begin with terms. Simple terms (constants and variables) were covered above. The weight of a complex term (one containing subterms) is calculated as follows:

weight of term = \[\text{BASE} + \text{SYMCT} \times (\text{number of symbols in term}) + \text{#ARG} \times (\text{number of immediate subterms of term}) + \text{MAXARGWT} \times (\text{weight of heaviest immediate subterm}) + \text{SUMARGWT} \times (\text{sum of weights of all immediate subterms})\]

Note that BASE does not apply to simple terms.

For the purposes of weighting, the major function symbol of a term is considered one of its subterms. The number of symbols is the total number of names of constants, variables, and function symbols appearing in the term. Thus the term

\[g(a.f(x1.maxlock))\]

is considered to have three subterms and to contain five symbols.

Suppose, for example, that the variable weight is set to 1 and that the weighting coefficients for terms are as follows, which is the default setting:

\[#\text{ARG}=0 \quad \text{MAXARGWT}=0 \quad \text{SUMARGWT}=1 \quad \text{SYMCT}=0 \quad \text{BASE}=0\]

Then the weights for some sample terms are:

<table>
<thead>
<tr>
<th>Term</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>f(a)</td>
<td>2</td>
</tr>
<tr>
<td>f(a,b)</td>
<td>3</td>
</tr>
<tr>
<td>f(a.g(a,b))</td>
<td>5</td>
</tr>
</tbody>
</table>

On the other hand, if the term weighting coefficients are

\[#\text{ARG}=1 \quad \text{MAXARGWT}=0 \quad \text{SUMARGWT}=0 \quad \text{SYMCT}=0 \quad \text{BASE}=100\]

then the weights of these same terms are:

<table>
<thead>
<tr>
<th>Term</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>f(a)</td>
<td>102</td>
</tr>
<tr>
<td>f(a,b)</td>
<td>103</td>
</tr>
<tr>
<td>f(a.g(a,b))</td>
<td>103</td>
</tr>
</tbody>
</table>
Literals

There are separate values of #ARG, MAXARGWT, SUMARGWT, SYMCT, AND BASE for literals. With these values, the weight of a literal is calculated as follows:

\[
\text{weight of literal} = \text{BASE} + \\
\text{SYMCT} \times (\text{number of symbols in literal}) + \\
\text{#ARG} \times (\text{number of arguments of literal}) + \\
\text{MAXARGWT} \times (\text{weight of heaviest argument}) + \\
\text{SUMARGWT} \times (\text{sum of weights of all arguments})
\]

For the purposes of weighting, the predicate symbol of a literal is counted as one of its arguments. Negative literals can have their weights adjusted, but this is done with patterns, discussed below. The negation symbol is not included in the symbol count.

Suppose that the weighting coefficients for terms are set to the defaults described above and that the weighting coefficients for literals are as follows, which is the default setting:

#ARG=0 MAXARGWT=1 SUMARGWT=0 SYMCT=0 BASE=0

Then the weights from some sample literals are:

- \( P \) 1
- \( P \) 1
- \( P(a,b) \) 1
- \( P(f(a)) \) 2
- \( P(f(a,b),a) \) 3

If instead the literal weighting coefficients are:

#ARG=1 MAXARGWT=5 SUMARGWT=0 SYMCT=0 BASE=0

then the weights of these same literals are:

- \( P \) 1
- \( P \) 7
- \( P(a,b) \) 8
- \( P(f(a)) \) 12
- \( P(f(a,b),a) \) 18

Clauses

There is a third set of #ARG, etc., for clauses. Using these values, the weight of a clause is calculated as follows:

\[
\text{weight of clause} = \text{BASE} + \\
\text{SYMCT} \times (\text{number of symbols in clause}) + \\
\text{#ARG} \times (\text{number of literals of clause}) + \\
\text{MAXARGWT} \times (\text{weight of heaviest literal}) + \\
\text{SUMARGWT} \times (\text{sum of weights of all literals})
\]

For weighting purposes, the number of symbols in the clause includes the implicit OR symbols between the literals, and any negation symbols in front of negative literals. Thus the clause

\( P \mid Q \)

is considered to contain three symbols, and
if \( P \) then \( Q \)
is considered to have four symbols, since it translates into \(-P \mid Q\).

Now suppose that the weighting coefficients for terms and literals have their default settings described above, and that variable weight has its default value of 1. Suppose further that the clause weighting coefficients are

\[
\#\text{ARG}=1 \quad \text{MAXARGWT}=0 \quad \text{SUMARGWT}=1 \quad \text{SYMCT}=0 \quad \text{BASE}=-1,
\]

which is the default. Then the weights of some sample clauses are:

\[
\begin{align*}
P; & \quad 1 \\
R \mid P \mid Q; & \quad 3 \\
P \mid Q \mid R; & \quad 5 \\
-P; & \quad 1 \\
\text{if } P \text{ then } Q; & \quad 3 \\
P(f(a)) \mid Q(x); & \quad 4
\end{align*}
\]

12.3. Weighting With Patterns

Weighting patterns are a mechanism for overriding the above weighting algorithm to assign particular weights to specific terms, literals, and clauses, as well as to terms, literals, or clauses which are characterized by their matching a particular pattern. Some simple patterns and their meanings are:

- \( a:+10 \) the term \( a \) has weight 10
- \( \text{NOT}:+6 \) negative literals should have 6 added to their weight
- \( f(2):+3 \) the weight of any term of the form \( f(<\text{term}> \) should be 3 plus twice the weight of \( <\text{term}> \).

There is a list of patterns in each weighting parameter set. If a given term, literal, or clause matches more than one pattern in the list, then the first one it matches has priority. For example, if the term \( f(a,b) \) is weighed according to the pattern list:

\[
f(a,2):+5 \quad f(a,b):+15;
\]

then it is given a weight of seven (assuming that \( f \), \( a \), and \( b \) have their default weights of 1).

The exact format of a weighting pattern is

\[
<\text{basic-pattern}>:<\text{increment}>
\]

where \(<\text{increment}>\) is a signed floating-point number, and \(<\text{basic-pattern}>\) can be any one of the following:

1. A constant. This matches only an occurrence of the constant.
2. \(<\text{x<int}>\) where \(<\text{int}>\) is a positive integer (e.g., \( x4 \)). This matches only a variable with the given number.
3. \(*<\text{x<int}>\) where \(<\text{int}>\) is a positive integer. This matches any variable, except that multiple occurrences of \(*<\text{int}>\) in the same pattern must match the same variable. For example, the pattern \( f(*x1,*x1):+2 \) would match the term \( f(x2,x2) \), but not the term \( f(x1,x2) \).
4. \(*<\text{t<int}>\) where \(<\text{int}>\) is a positive integer. This matches any term, except that multiple occurrences of \(*<\text{t<int}>\) in the same pattern must match the same term.
5. \(<\text{multiplier}>\), which is a real number. This matches any term. The effect of a match is to multiply the weight of the subterm by the multiplier. The result is added into the weight of the current term.

6. \(<\text{name}>\langle<\text{arg}-1>,<\text{arg}-2>,\ldots,<\text{arg}-n>\rangle\) where \(<\text{arg}-i>\) is a \(<\text{basic-pattern}>\). This matches a complex term in which \(<\text{name}>\) is the predicate/function symbol, and \(<\text{arg}-i>\) matches the \(i\)th subterm (for all \(i\) from 1 to \(n\)).

The weight of the term matched by the pattern is computed by adding the \(<\text{increment}>\) to the weights generated from having \(<\text{multiplier}>\)s in the pattern. Thus, if \(f(a,g(1.5,-.5)):+2.5\) matches a term, the final weight is 2.5 (the increment) plus 1.5 times the weight of the first argument of \(g\) plus -0.5 times the weight of the second argument of \(g\).

### 12.4. Use of Weighting Options

When option family seven is requested, the user is prompted for a weighting parameter set number. Recall that parameter set one is used for determining whether to keep a new clause and parameter set two for picking the next given clause. Then the following is displayed.

```plaintext
for clauses: #ARG=1 MAXARGWT=0 SUMARGWT=1 SYMCT=0 BASE=-1;
for literals: #ARG=0 MAXARGWT=1 SUMARGWT=0 SYMCT= BASE=0;
for terms: #ARG=0 MAXARGWT=0 SUMARGWT=1 SYMCT= BASE=0;
variable weight = 1
patterns: ;
modify? >
```

If the user responds with an "n" to the "modify >" prompt, he will be prompted for another weighting parameter set. A null line returns control to the highest level. If he responds "y" he will be prompted to enter alternatives to the defaults for the various weighting parameters. To leave one of them unchanged, enter a null line in response to the prompt. New patterns will be added to the set of existing patterns. Currently there is no convenient way to delete an existing pattern; this must be done by editing the saved option file.

As an aid to debugging weighting options and patterns, the \(w\) command can be used to examine the weights of clauses, literals, and terms.

### References


