82

A Guide to LP, The Larch Prover

Stephen J. Garland and John V. Guttag

December 31, 1991

A Guide to LP, The Larch Prover

Stephen J. Garland and John V. Guttag

December 31, 1991

John V. Guttag and Stephen J. Garland are at the MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139.
E-mail: garland@lcs.mit.edu, guttag@lcs.mit.edu
The authors were supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-89-J-1988, by the National Science Foundation under grant CCR-8910848, and by the Digital Equipment Corporation.

Abstract

This guide provides an introduction to LP (the Larch Prover), Release 2.2. It describes how LP can be used to axiomatize theories in a subset of multisorted first-order logic and to provide assistance in proving theorems. It also contains a tutorial overview of the equational term-rewriting technology that provides, along with induction rules and other user-supplied nonequational rules of inference, part of LP's inference engine.

Contents

1 Introduction

LP is a theorem prover for a subset of multisorted first-order logic. It is designed to work efficiently on large problems and to be used by relatively naive users. It has been used to analyze formal specifications written in Larch [14, 15, 12], to reason about algorithms involving concurrency [10, 30], and to establish the correctness of hardware designs [10, 28].

LP is intended primarily as an interactive proof assistant or proof debugger, not as a fully automatic theorem prover. Its design is based on the assumption that initial attempts to state conjectures correctly, and then to prove them, usually fail. As a result, LP is designed to carry out routine (and possibly lengthy) steps in a proof automatically and to provide useful information about why proofs fail, if and when they do. To ensure that users will not be surprised by its behavior, LP does not employ complicated heuristics for finding proofs automatically. It does make it easy for users to employ standard techniques such as proofs by cases, induction, or contradiction.

Section 2 provides a context for the technical details of LP (Release 2.2) by discussing the style of use that LP is intended to support. Section 3 tells how to get started using LP. Section 4 describes how theories are axiomatized in LP. Sections 5 and 6 describe LP's proof techniques. Section 7 summarizes the features of LP, and Section 8 provides some hints for using LP. Appendix A contains a tutorial on the theory and implementation of equational term-rewriting. Appendix B contains a complete record of a sample proof carried out using LP.

Readers with access to LP will find it helpful to experiment with LP while reading this guide. LP has an extensive online help facility that contains most of the information in Sections 3 through 7.

2 The proof life cycle

Proving is similar to programming: proofs are designed, coded, and debugged. The first step in designing a proof is to formalize the objects being reasoned about. The next is to formalize a conjecture to be proved, for example, a property implied by a specification, or an invariant to be maintained by a concurrent algorithm. The last step in the design is to outline a structure for the proof, including key lemmas and methods of proof.

Formalization is straightforward for Larch Shared Language [15] specifications and has been automated [12]. At present it is less automatic for concurrent algorithms and for circuits, although efforts are underway to automate some of these translations [30, 21]. For large applications, formalization usually involves identifying subtheories that are analogous to data abstractions. Generally, the most difficult design step, and the one requiring the most insight, is determining the structure of the proof.

Designs for proofs translate into sequences of LP commands, much as program designs translate into code in a programming language. Details of this translation are discussed later in this guide.

Once part of a proof has been coded, LP can be used to debug it. Proofs of interesting conjectures hardly ever succeed the first time. Sometimes the conjecture is wrong. Sometimes the formalization is incorrect or incomplete. Sometimes the proof strategy is flawed or not detailed enough. When an attempted proof does fail, a variety of LP facilities (e.g., case analysis) can be used to understand the problem. Because most proof attempts do fail, LP is designed to fail relatively quickly and to provide useful information when it does.

LP is not designed to find difficult proofs automatically. Unlike the Boyer-Moore prover [3, 4], it does not use heuristics to formulate additional conjectures that might be useful in a proof. Unlike LCF [24] and Isabelle [25], it does not encourage users to define their own proof tactics; rather, it provides a set of standard tactics and simple mechanisms for controlling the application of these tactics. Strategic decisions, such as trying induction on a particular variable, must appear as explicit LP commands (either entered by the user or generated by an application-specific front-end to LP). But LP is more than a proof checker, since it does not require proofs to be described in minute detail. Hence it is best described as a proof debugger.

In line with LP's emphasis on debugging proofs, it is generally advisable to use axiomatizations that simplify terms rather than axiomatizations that produce unique (but possibly larger) normal forms. Such axiomatizations are often incomplete, and thereby increase the need for the kinds of auxiliary proof mechanisms described in Sections 5 and 6.

When debugging proofs, users frequently reformulate axioms and conjectures. When verifying a circuit, for example, users may discover that some important property does not follow from the description of the circuit (after all, discovering such things is the whole point of the process). When changing an axiomatization, users should recheck not only the conjecture whose proof uncovered a problem, but also any conjecture proved with the old axiomatization. LP has facilities that support such regression testing.

LP will, upon request, record a session in a script file that can be replayed using the **execute** command. In addition to recording user input, LP indents script files to reveal their proof structure, and it annotates proofs in script files with information that indicates when subgoals are introduced (e.g., for a proof by induction) and when subgoals or theorems are proved. On request, as LP executes an annotated (and possibly edited) proof, it halts execution and prints an error message if the annotations do not match the execution. These checks are useful when changes in an axiomatization cause some step in a proof to succeed with less user guidance than expected or to require more guidance. Without the check, LP

```
set name nat
declare sort Nat
declare variables i, j, k: Nat
declare operators
    0: → Nat
    s: Nat → Nat
    +: Nat, Nat → Nat
    <: Nat, Nat → Bool
    ..
assert Nat generated by 0, s
assert ac +
assert
    i + 0 == i
    i + s(j) == s(i + j)
    not(i < 0)
    0 < s(i)
    s(i) < s(j) == i < j
    ..
set name lemma
prove i < j ⇒ i < (j + k) by induction on j
    <> 2 subgoals for proof by induction on ‘j’
        [ ] basis subgoal
        resume by induction on i
        <> 2 subgoals for proof by induction on ‘i’
            [ ] basis subgoal
            [ ] induction subgoal
        [ ] induction subgoal
    [ ] conjecture
qed
```

Figure 1: Sample LP-annotated script file

might, for example, apply a tactic that the user intended for the basis step of an induction to the proof of the induction step. This checking helps prevent proofs from getting “out of sync” with their author’s conception of how they should proceed.

Figure 1 displays an LP-annotated script for a simple proof (see Section 3.1 for an explanation of the typesetting conventions). The **declare** commands introduce variables and operators (the **..** terminates a multiline command), the **assert** commands axiomatize properties of the operators (e.g., that $+$ is associative and commutative), and the **prove** command initiates a proof by induction. The next two lines contain annotations supplied by LP. The diamond ($<>$) indicates that LP has introduced two subgoals for the proof by induction. The box ([ ]) indicates that the basis case of the induction succeeded without further interaction. The **resume** command starts a subsidiary induction, and the subsequent LP-supplied diamonds and boxes indicate that LP finished the remaining steps in the proof without further interaction. The **qed** command on the last line asks LP to confirm that there are no outstanding conjectures. Details about the commands in this script, and about their execution, are given in the following sections.

3 Getting started

This section describes how to use elementary LP features for entering commands and displaying information.

3.1 Typesetting conventions

This guide has been typeset using LaTeX [20]. Most of the examples of LP input and output in this guide have been preprocessed so that they are typeset using the following conventions.

- LP command names and other keywords are printed in boldface (e.g., **assert**, **display**, **generated by**, **induction**, **set**).
- Identifiers (see Section 4.1) are printed in italics (e.g., *Bool*, *false*, *x*, *x*1), except for numerals (e.g., 0, 1).
- Names of facts and conjectures (see Section 3.4) are printed in italics (e.g., *nat.1*, *lemmaCaseHyp.2*).
- Names of files (see Sections 3.3 and 3.5) are printed in italics.
- The following multicharacter symbols are printed using special non-ASCII characters.

Symbol	Printed as
`->`	$\rightarrow$
`=>`	$\Rightarrow$
`<=>`	$\Leftrightarrow$
`\in`	$\in$
`\union`	$\cup$
`\subseteq`	$\subseteq$
`<=`	$\leq$
`>=`	$\geq$

If you prepare input for LP based on the examples in this guide, you must enter the special symbols in the right column of the table using the ASCII forms in the left column. While `->`, `=>`, and `<=>` are built-in symbols of LP (see Sections 4.1 and 4.8), the infix operator symbols `\in`, `\union`, `\subseteq`, and `<=` must be declared before use (see Section 4.1). The symbol `>=` has a built-in meaning to the **register height** command (see Section 4.9.1), but is not predeclared as an infix operator. Both it and the symbol `<=` must be declared before use (see Section 4.1) and have no built-in semantics.

3.2 Online help

Much of the information in this guide is also available from LP's online help facility.[1] The command **help lp** provides an overview of this facility. In general, users can type **help** followed by a list of topics for which they desire help. The command **help ?** provides a list of all topics for which help is available.

[1] If LP does not seem to be behaving as described in this guide, consult the help facility. You may be using a different release of LP.

In most cases when LP expects input, users can type a question mark to obtain a summary of the legal responses. For example, the command **?** produces a list of all LP commands, and the command **assert ?** provides a summary of the kinds of axioms that can be asserted.

3.3 Entering commands

LP generally prompts users to enter commands interactively from the keyboard. Users can also create files containing sequences of commands and instruct LP to execute these command files; for example, the command **execute *nat*** causes LP to execute the commands in the file *nat.lp*. (By convention, LP command files have names ending with *.lp*, and LP supplies *.lp* as a default suffix when no suffix appears in the **execute** command.) Command files may themselves contain **execute** commands; however, to guard against infinite loops, LP treats execution of a file that is already being executed as an error. Execution of a command file continues until the file is exhausted, until execution is interrupted by the user, until an error occurs, or until a **quit** or **stop** command is executed.

When run under Unix, LP can also be invoked with the names of one or more command files given as optional arguments. For example, the Unix shell command *lp nat* causes LP to start by executing the commands in the file *nat.lp*, and then to prompt the user for input when that file is exhausted.

All LP commands begin with a keyword (e.g., **display** or **critical-pairs**), which can be abbreviated to any unambiguous prefix (e.g., **dis** or **crit**). Some LP commands contain further keywords or phrases, which can also be abbreviated. For example, **set automatic-ordering off** can be shortened to **set auto-ord off**. Commands can be entered in upper, lower, or mixed case.

When an LP command requires more arguments than a user supplies, LP will prompt the user for the missing arguments. Users who need help can type a question mark followed by a carriage return to see what LP expects next; typing a carriage return alone aborts the command. If a missing argument is likely to be lengthy, LP prompts the user to enter it on subsequent lines and to terminate the input with a line consisting of two periods (**..**), which can be preceded by white space. The **declare** and **assert** commands in Figure 1 use this convention.

A comment starting with a % can occur at the end of any line of input or on a line by itself. LP ignores comments.

The following editing capabilities are available for use when typing input to LP.

Editing character	Action
rubout	Delete last character typed
control-U	Delete current line of input
control-R	Redisplay current line
control-L	Clear screen and redisplay current line
control-\	Continue line

Typing a control-G causes LP to interrupt execution of the current command.[2] The **quit** command causes LP to terminate.

[2]On some Unix systems, users must type control-\ instead of control-G. To make control-G work, users can put *stty quit control-G* in their *.login* scripts.

```
Induction rules:
nat.1: Nat generated by 0, s

Operator theories:
nat.2: ac +

Rewrite rules:
lemma.1: (i < j) => (i < (j + k)) -> true
nat.3:   0 + i -> i
nat.4:   s(j) + i -> s(i + j)
nat.5.1: i < 0 -> false
nat.6:   0 < s(i) -> true
nat.7:   s(i) < s(j) -> i < j
```

Figure 2: Output generated by **display** command

3.4 Naming and displaying objects

The **set name** command provides users with control over the names assigned to facts (i.e., to axioms, hypotheses, and theorems) and to conjectures. For example, the command **set name** *nat* in Figure 1 causes LP to assign the names *nat.1*, . . . , *nat.8* to the eight subsequently asserted axioms, and the command **set name** *lemma* causes LP to assign the name *lemma.1* to the conjecture.

The **display** command enables users to view a set of objects. For example, if the **display** command is executed after the script in Figure 1 terminates, LP produces the output shown in Figure 2. Differences between what the user typed in Figure 1 and what LP displayed in Figure 2 reflect inferences performed by LP, as described in this and the next sections.

Optional arguments to the **display** command can be used to restrict the display to objects of specified types and with specified names. For example, if Figure 2 had been produced by the command **display rewrite-rules *nat*** (or **display r *nat***, for short), it would have contained only the rewrite rules *nat.3* to *nat.7*.

In general, names begin with an identifier that consists of a sequence of letters, digits, and special characters such as underscores. LP is not sensitive to the case of letters in a name. Thus, *Nat.1* and *nat.1* refer to the same item. When displaying a name, LP uses the capitalization it found in the first occurrence of that name.

LP assigns new names of the form *prefix.number* (where *prefix* is "*user*" unless changed by the **set name** command, and where *number* increases each time a new name is required) to axioms introduced by the **assert** command, to critical-pair equations deduced in response to the **critical-pairs** and **complete** commands (see Sections 5.3 and 5.4), and to conjectures introduced by the **prove** command. LP assigns names of the form *prefixCaseHyp.number*, *prefixInductHyp.number*, etc., to case, induction, and other hypotheses it introduces during the proof of a conjecture. It assigns subnames of the form *name.number* to subgoals in a proof of a conjecture named *name* and to consequences deduced by instantiating (see Section 5.5) a fact named *name* or by applying a deduction rule (see Section 4.7).

When used as arguments to commands, names such as *nat.5* encompass the item with that name as well as all items with subnames of that name (e.g., *nat.5.1*). Subnames can be excluded by appending an exclamation mark to a name (e.g., *nat.5!*). Ranges of names can be specified by appending a colon followed by a number or the word *last* to a numbered name (e.g., *nat.3:6* or *nat.5:last*).

Asterisks in name prefixes supplied as arguments to commands serve as patterns that match arbitrary sequences of characters in the prefix. For example, the command **display** ∗***Hyp*** causes LP to display all facts whose name prefix ends with the letters *Hyp*.

Figure 14 in Section 7 provides details concerning these naming conventions.

3.5 Recording sessions

The command **set log** *fileName* causes LP to record all subsequent input and output in a file named *fileName.lplog* (unless *fileName* contains a period, in which case LP does not supply the suffix *.lplog*). Logging is ended by the **unset log** and **quit** commands.

The command **set script** *fileName* causes LP to record all subsequent user input in a file named *fileName.lpscr* (unless *fileName* contains a period, in which case LP does not supply the suffix *.lpscr*). LP annotates such a script file by commenting out illegal commands, by substituting the text of the executed file for an **execute** command, by marking the creation of subgoals and the completion of proofs, and by indenting the script file to reveal its proof structure. Scripting is ended by the **unset script** and **quit** commands.

Script files can be replayed using the **execute** command, and they can be edited before being replayed. Although a script file can be replayed directly using the command **execute *fileName.lpscr***, it is generally advisable to rename the script file to *fileName.lp* and then replay it using the command **execute *fileName*** (lest a **set script** command cause LP to overwrite the command file being executed).

3.6 Settings

The **set** command can be used to control many aspects of LP's behavior. Typing **set** alone causes LP to display a list of its current settings. Typing **set** followed by the name of a setting causes LP to display that setting and to prompt the user for a new value; responding with a carriage return leaves the setting unchanged. Typing **set** followed by the name of a setting and a value changes that setting. This section describes some elementary settings; Section 7.5 summarizes the others.

All settings have default values. The **unset** command can be used to reset a setting to its default value. The **unset all** command returns all settings to their default values.

set directory *string*

The **directory** is the name of the directory in which LP creates script (*.lpscr*), log (*.lplog*), and other output files. By default, **directory** is the name of the working directory from which LP was invoked.

set lp-path *string*

The **lp-path** is a list of names of directories that LP searches when looking for command or other input files. By default, **lp-path** is "`. ~ ~lp/axioms ~lp`". A period (.) in the value of **lp-path** refers to the current **directory**. A tilde (~) refers to the user's home directory. The characters `~lp` refer to the directory in which the help files and examples for LP are installed. The actual location of this directory can be determined by typing the command **version**; it can be changed by invoking LP with the shell command *lp −d directoryName*.

set page-mode { **on** | **off** }

When **page-mode** is **off** (the default setting), LP displays output continuously. When it is **on**, LP displays output a screenful at a time. At the end of each screenful, LP prompts the user with `--More--` to type a character indicating what to do next. The options are as follows:

Response	Action
<space>	display next screenful
<return>	display next line
<digit>	display next <digit> lines
`d`	display next half screenful
`u`	display continuously until next user interaction
`q`	display nothing until next user interaction
`?`	display this menu

set prompt ***string***

The **prompt** is the string that LP uses when prompting users to enter commands. The **set prompt** command allows users to change this prompt. If the new prompt begins or ends with a space, it should be enclosed within `''` marks, as in **set prompt** `'>> '`.

LP replaces the first exclamation mark (!) in the prompt, if any, by the number of the next command. LP numbers commands entered from the terminal by consecutive integers. It numbers commands obtained during execution of a command file by appending a period followed by consecutive integers to the command number for the **execute** command; thus command 5.2.3 is the third command in the file executed in response to the second command in the file executed in response to the fifth command typed by the user.

By default, **prompt** is `'LP!: '` , which causes LP to issue prompts of the form "`LP1: `", "`LP2: `", . . .

set trace-level ***number***

The **trace-level** controls how much information LP prints as it executes commands. At trace level 0, which is the least verbose, LP prints nothing other than user interactions and the final results of commands. At trace level 1, which is the default, LP reports major actions taken in the course of a session. At higher trace levels, it provides more detailed information, as described by the online help facility.

4 Defining theories

The basis for proofs in LP is a logical system consisting of a set of declared operators, the properties of which are axiomatized by equations, rewrite rules, operator theories, induction rules, and deduction rules (all expressed in a subset of multisorted first-order logic). Each kind of axiom has two semantics, a definitional semantics in first-order logic and an operational semantics that is sound with respect to the definitional semantics but not necessarily complete.

Sections 5 and 6 describe how axioms interact with LP's proof techniques. Appendix B illustrates how they are used in a complete proof.

4.1 Declarations and identifiers

Identifiers for sorts, operators, and variables must be declared prior to use. Declarations (such as those in Figures 1 and 3) assign sorts to variables and signatures to operators. The symbol $\rightarrow$ (typed as `->`) separates the declaration for the domain of an operator (which is a list of sorts) from that of its range (which is a single sort). Constants (e.g., 0 and *empty*) are special cases of operators.

LP predefines the sort *Bool*, as well as the built-in operators *true*, *false*, *if*, *not*, $=$, & (and), | (or), $\Rightarrow$ (implies, typed as `=>`), and $\Leftrightarrow$ (if and only if, typed as `<=>`). It can also generate new variables, constants, and operators during the course of a session. The name of an LP-generated variable consists of the first letter of its sort, possibly followed by a number (e.g., n, $n1$ for sort *Nat*). The names of LP-generated constants end with the letter c, possibly followed by a number (e.g., *xc* or *xc1*). Users are not prevented from declaring such identifiers themselves, but may find it confusing to do so (or even unsound, if they mistakenly believe that they are declaring new constants).

Identifiers for sorts, variables, constants, and prefix operators are sequences of letters, digits, and special characters such as underscores (_) and apostrophes ('). Identifiers for infix operators are sequences of characters drawn from an implementation-defined set of infix characters (e.g., "`!#$&*+-./<=>@\^|~`"); the symbols $==$ and $\rightarrow$ (i.e., `->`) cannot be used for infix operators because LP reserves them for other uses. Identifiers for infix operators may also consist of a backslash (\) followed by a prefix identifier (e.g., `\in`, which is printed in this guide as $\in$ by the conventions described in Section 3.1). The command **help operator** provides a precise description of LP's lexicographical conventions.

LP automatically overloads the built-in operators $=$ and *if*, once for each declared sort S, with signatures $=\!:\!S,\, S \rightarrow Bool$ and $if\!:\!Bool,\, S,\, S \rightarrow S$. Users can also overload identifiers. For example, the declarations and axioms in Figure 3 can be used together with those in Figure 1 when reasoning about both natural numbers and sets. LP uses context to distinguish the variable s (of sort *Set*) from the operator s (with signature $Nat \rightarrow Nat$). Users can overload other operators as well. For example, the commands

declare operators
    $\cup$: *Set, Elem* $\rightarrow$ *Set*
    $\cup$: *Elem, Set* $\rightarrow$ *Set*
    ..
assert
    $s \cup e == insert(e, s)$
    $e \cup s == insert(e, s)$
    ..

further overload the operator $\cup$ in Figure 3 to invent shorthands for adding an element to a set. The only

```
set name set
declare sorts Elem, Set
declare variables e, e′: Elem, s, x, y, z: Set
declare operators
      empty: → Set
      insert: Elem, Set → Set
      singleton: Elem → Set
      ∪: Set, Set → Set
      ∈: Elem, Set → Bool
      ⊆: Set, Set → Bool
      ..
assert Set generated by empty, insert
assert Set partitioned by ∈
assert
      singleton(e) == insert(e, empty)
      not(e ∈ empty)
      e ∈ insert(e′, s) == e = e′ | e ∈ s
      e ∈ (x ∪ y) == e ∈ x | e ∈ y
      empty ⊆ s
      insert(e, x) ⊆ y == e ∈ y & x ⊆ y
      ..
```

Figure 3: Sample axiomatization for finite sets

restrictions on overloading identifiers is that users do not overload the built-in identifiers and that they do not declare an identifier both as a variable and as a constant of the same sort. The next section describes how to specify, when necessary, one of several possibly ambiguous overloadings of an operator.

The command **display symbols** causes LP to print a list of all declared identifiers.

4.2 Terms

A term in multisorted first-order logic consists of either a variable or of an operator and a sequence of terms, known as its arguments. The number and sorts of the arguments in a term must agree with the declaration for (some overloading of) the operator. The number of arguments is called the *arity* of the operator. An operator with arity 0 is called a *constant*. Infix operators are written between their arguments (e.g., $i + j$), constants are written with no arguments (e.g, 0 or *empty*), and prefix operators with nonzero arity are followed by a parenthesized list of arguments (e.g, $s(i)$ or $insert(e, x)$).

LP uses a limited amount of precedence when parsing terms, but generally requires users to supply parentheses to specify the associativity of operators in terms with multiple infix operators. User-declared operators bind more tightly than the equality operator, which binds more tightly than the built-in boolean operators. Thus, LP parses the term $(a < b \,\&\, b = c + d) \Rightarrow a < (c + d)$ in the same way that it parses $((a < b) \,\&\, (b = (c + d))) \Rightarrow (a < (c + d))$. Unparenthesized sequences of infix operators at the same precedence level are permitted only in terms such as $t_1 + t_2 + t_3 + t_4$, which consist of a sequence of terms separated by a single operator with signature $S, S \rightarrow S$ for some sort S.[3] Thus, LP allows $p \,\&\, q \,\&\, r$ and

[3]It is considered good practice to write terms such as this only when the operator is associative. When the operator is not

$a + b + c$, but not $p \,\&\, q \mid r$ or $a + b - c$.

In some cases, users must append qualifications to terms to clarify which of several overloadings of an identifier is meant. For example, given the declarations

declare operators a, b: $\rightarrow$ *Nat*, $-$: *Nat*, *Nat* $\rightarrow$ *Nat*
declare operators a, b: $\rightarrow$ *Set*, $-$: *Set*, *Set* $\rightarrow$ *Set*

it is ambiguous whether the term $a - b$ denotes the difference of two numbers or the difference of two sets. To distinguish which of these two interpretations they intend, users must write either $a{:}Nat - b{:}Nat$ or $a{:}Set - b{:}Set$.[4]

4.3 Equations

LP is based on a subset of first-order logic in which equations play a prominent role. Figure 4, for example, contains LP commands that enter the usual first-order axioms for groups. Variables appearing in the axioms are implicitly universally quantified.

declare sort G
declare variables x, y, z: G
declare operators e: $\rightarrow G$, i: $G \rightarrow G$, $*$: $G, G \rightarrow G$
assert
    $(x * y) * z == x * (y * z)$
    $e == i(x) * x$
    $x == e * x$
    ..

Figure 4: LP axiomatization of group theory

LP uses the logical symbol $==$ for equality in an equation. This symbol is implicit in axioms such as $0 < s(i)$ in Figure 1, which are shorthands for equations with right side *true*. LP binds $==$ less tightly than the (overloaded) equality operator $=$, so that, for example, $e \in insert(e', s) == e = e' \mid e \in s$ in Figure 3 can be written without more parentheses. It is parsed as

$$(e \in insert(e', s)) == ((e = e') \mid (e \in s))$$

The connective $==$ can appear only once in an equation, whereas $=$ can appear many times. The definitional semantics makes no distinction between $==$ and $=$.

Equations can also be entered using the connective $\rightarrow$ instead of $==$. This constrains the way in which LP will orient them into rewrite rules (see Section 4.4), but does not alter their definitional semantics.

LP treats as inconsistent the equation $true == false$ and all equations of the form $x == t$ or $not(x = t) == true$, where x is a variable and t is a term not containing x. Thus, LP is designed for reasoning about models in which every sort has at least two elements.[5] Inconsistencies can be used to

associative, the term is parsed from left to right, for example, as $((t_1 + t_2) + t_3) + t_4$.)

[4]Release 2.2 of LP requires both a and b in these terms to be qualified, even though the sort of one determines the sort of the other. For terms such as i/j, where i and j are unambiguous, but $/$ is not, users can write $(i/j){:}Rational$ or $(i/j){:}Nat$ to disambiguate the term. Later releases of LP will do a better job of type inference.

[5]In practice, axioms that constrain a sort to be empty or to have a single element are almost always either intentionally inconsistent or result from mistaken formalizations. Hence LP chooses to treat them as inconsistent to protect users from mistakes, and to make some proofs go faster, rather than to provide a more general (and elaborate) logical framework.

establish subgoals in proofs of implications and in proofs by cases and contradiction. If they arise in other situations, they indicate flaws in the current logical system.

An *equational theory* is a theory (i.e., a set of facts) axiomatized by a set of equations. Equational theories can be characterized syntactically, as follows. The set of terms constructed from a set of variables and operators is called a *free word algebra* or *term algebra*. A set E of equations defines a congruence relation on a term algebra, this relation being the smallest one that contains the equations in E and that is closed under reflexivity, symmetry, transitivity, instantiation of free variables, and substitution of equals for equals. An equation $t_1 == t_2$ is in the *equational theory* of E, or is an *equational consequence* of E, if t_1 is congruent to t_2.

Figure 5 shows a sample informal proof that $i(e) == e$ is an equational consequence of the axioms for groups in Figure 4.

Step	Equation	Justification
1.	$(x * y) * z == x * (y * z)$	Axiom
2.	$e == i(x) * x$	Axiom
3.	$x == e * x$	Axiom
4.	$(i(y) * y) * z == i(y) * (y * z)$	Replace x by $i(y)$ in 1
5.	$e * z == i(y) * (y * z)$	Apply 2 to 4 (with y for x)
6.	$z == i(y) * (y * z)$	Apply 3 to 5 (with z for x)
7.	$z == i(i(z)) * (i(z) * z)$	Replace y by $i(z)$ in 6
8.	$z == i(i(z)) * e$	Apply 2 to 7 (with z for x)
9.	$(i(i(e)) * e) * i(e) ==$ $i(i(e)) * (e * i(e))$	Replace x by $i(i(e))$, y by e, z by $i(e)$ in 1
10.	$e * i(e) == i(i(e)) * (e * i(e))$	Apply 8 to 9 (with e for z)
11.	$i(e) == i(i(e)) * i(e)$	Apply 3 to 10 (with e for x)
12.	$i(e) == e$	Apply 2 to 11 (with $i(e)$ for x)

Figure 5: Sample derivation from group axioms

4.4 Rewrite rules

Some of LP's inference mechanisms work directly with equations. Most, however, require that equations be oriented into rewrite rules. Rewrite rules have the same logical meaning as equations, but behave differently operationally. A *rewrite rule* is an ordered pair $\langle l, r\rangle$ of terms, usually written $l \rightarrow r$, such that l is not a variable and every variable that occurs in r also occurs in l.[6] A *term-rewriting system*, or a *rewriting system* for short, is a set of rewrite rules.

LP orients equations into rewrite rules and uses these rewrite rules to reduce terms to normal forms. For example, LP orients the equations asserted in Figure 1 into the rewrite rules displayed in Figure 2. The additional commands

declare operator 1: $\rightarrow$ *Nat*
assert $1 == s(0)$
prove $1 < 1 + 1$

cause LP to orient the equation into the rewrite rule $1 \rightarrow s(0)$, after which it can prove the conjecture

[6]As explained below, pairs that violate these restrictions have unpleasant operational consequences.

$1 < 1+1$ by reducing it to $true$, as follows.

Term	Derivation
$1 < 1+1$	Conjecture
$s(0) < s(0) + s(0)$	Apply $1 \rightarrow s(0)$ three times
$s(0) < s(0 + s(0))$	Apply *nat.4*
$0 < 0 + s(0)$	Apply *nat.7*
$0 < s(0)$	Apply *nat.3*
true	Apply *nat.6*

Note that rewrite rules *nat.3* and *nat.7* can be applied in either order.

To describe this process more precisely, we define a *substitution* σ to be a mapping from variables to terms such that $\sigma(v)$ is identical to v for all but a finite number of variables. The domain of a substitution is extended to terms in the usual way: $\sigma(f(t_1, \ldots, t_n))$ is defined to be $f(\sigma(t_1), \ldots, \sigma(t_n))$. A substitution σ *matches* a term t_1 to a term t_2 if $\sigma(t_1)$ is identical to t_2.

Each rewriting system R defines a binary relation $\leadsto_R$ (*rewrites* or *reduces directly to*) on the set of all terms. Operationally[7], $t \leadsto_R u$ if there is some rewrite rule $l \rightarrow r$ in R and some substitution σ that matches l to a subterm of t such that u is the result of replacing that subterm by $\sigma(r)$.

The relation $\leadsto_R^*$ (*reduces* or *rewrites to*) is the reflexive transitive closure of $\leadsto_R$. Thus $t \leadsto_R^* u$ if and only if there are terms $t_1, \ldots, t_n$ such that $t = t_1 \leadsto_R \ldots \leadsto_R t_n = u$. The relation $\leadsto_R^+$ is the transitive irreflexive closure of $\leadsto_R$. When R is clear from context, we write $\leadsto$ for $\leadsto_R$, $\leadsto^*$ for $\leadsto_R^*$, and $\leadsto^+$ for $\leadsto_R^+$.

It is usually essential that R be *terminating*, in other words, that there be no infinite sequence $t_1 \leadsto_R t_2 \leadsto_R t_3 \ldots$ of reductions. In general, it is undecidable whether a set of rewrite rules is terminating. However, as discussed below, LP provides several mechanisms that automatically orient many sets of equations into terminating rewriting systems. For example, LP automatically orients the equations for groups in Figure 4 into the rewrite rules

$$(x * y) * z \rightarrow x * (y * z)$$
$$i(x) * x \rightarrow e$$
$$e * x \rightarrow x$$

It automatically reverses the left and right sides of the second and third equations, thus preventing nonterminating reduction sequences such as $e \leadsto i(e) * e \leadsto i(e) * i(e) * e \leadsto \ldots$ and $e \leadsto e * e \leadsto e * e * e \leadsto \ldots$[8] LP's facilities for orienting equations into rewrite rules are discussed in Section 4.9.

A term t is said to be *irreducible* if there is no term u such that $t \leadsto u$. If $t \leadsto^* u$ and u is irreducible, then u is a *terminal* or *normal form* of t. A term can have many different terminal forms. For example, both $e * z$ and $i(y) * (y * z)$ are normal forms of $(i(y) * y) * z$ with respect to the rewrite rules for group theory above.

Unless directed otherwise, LP keeps all rewrite rules and equations in normal form. If a rewrite rule or equation reduces to an *identity*, that is, to one in which the right and left sides have the same normal form, it is discarded.

If a term has only one normal form, that is called the *canonical form* of the term. A terminating rewriting

[7]Another characterization of $\leadsto_R$ is as the smallest binary relation such that $\sigma(l) \leadsto_R \sigma(r)$ for every rewrite rule $l \rightarrow r$ in R and every substitution σ, and such that $f(t_1, \ldots, t_i, \ldots, t_n) \leadsto_R f(t_1, \ldots, u, \ldots, t_n)$ whenever $t_i \leadsto_R u$.

[8]These examples show why a rewrite rule $l \rightarrow r$ must obey the restrictions that l cannot be a variable and that all variables in r must also be in l.

system in which all terms have a canonical form is said to be *convergent* (cf. Appendix A).

If a rewriting system is convergent, its rewriting theory (that is, the equations that can be proved by reducing them to identities) is identical to its equational theory (that is, the equations that follow logically from the rewrite rules considered as equations). Unfortunately, most rewriting systems that arise in practice are not convergent. In these systems, the rewriting theory is a proper subset of the equational theory. For example, the equation $i(e) == e$ is in the equational theory of the rewrite rules for group theory above, as proved in Figure 5, but it is not in the rewriting theory (because it is irreducible and yet is not an identity).

The proof mechanisms discussed in Sections 5 and 6 compensate for the incompleteness that results when a system's rewriting theory does not include all of its equational theory.

4.5 Operator theories

LP provides special mechanisms for handling some equations that cannot be oriented into terminating rewrite rules. The LP command **assert ac** $+$ in Figure 1 says that $+$ is associative and commutative. Logically, this assertion is merely an abbreviation for two equations:

$$\begin{aligned} x + (y + z) \quad &== \quad (x + y) + z \\ x + y \quad &== \quad y + x \end{aligned}$$

Operationally, it causes LP to use *equational term-rewriting* to match and unify terms (see Section 5.3) modulo associativity and commutativity. In equational term-rewriting, a substitution σ *matches* t_1 to t_2 *modulo* a set E of equations if $\sigma(t_1) == t_2$ is in the equational theory of E. For example, if $+$ is **ac**, the rewrite rule $a + b \rightarrow c$ will reduce the term $a + c + b$ to $c + c$.

Equational term-rewriting not only increases the number of theories that LP can reason about, but also reduces the number of axioms required to describe various theories, the number of reductions necessary to derive identities, and the need for certain kinds of user interaction, for example, case analysis. The main drawback is that equational term-rewriting can be much slower than conventional term-rewriting; associative-commutative matching, for example, is NP-hard, whereas conventional matching is linear.

LP recognizes two nonempty operator theories: the associative-commutative theory (**assert ac**) and the commutative theory (**assert commutative**). The commutative theory is important because commutative axioms, such as $x + y == y + x$, cannot be oriented into terminating rewrite rules. The associative-commutative theory is important because an equation describing associativity, such as $(x + y) + z == x + (y + z)$, cannot be oriented into a terminating rule if commutative matching is used for the associative operator.

To facilitate matching terms involving **ac** or **commutative** operators, LP *flattens* the internal representation of terms by arranging the arguments to associative-commutative and commutative operators in a canonical order.[9] The visible impact of this is that, when LP prints terms, the order in which arguments appear may be affected. For example, when $+$ is associative-commutative and $=$ is commutative, LP will recognize $(a + b) + c = d$ and $d = b + (c + a)$ as having the same meaning, and it will print both as $a + b + c = d$. Flattening also explains why the display of *nat.3* and *nat.4* in Figure 2 differs from the original form of those equations in Figure 1.

[9]When an assertion that an operator is **commutative** or **ac** is deleted, terms involving that operator are unflattened, perhaps to different forms than they originally had.

4.6 Induction rules

LP allows users to axiomatize theories using induction rules, which are logically equivalent to infinite sets of first-order formulas. Induction rules increase the number of theories that can be axiomatized by finite sets of assertions. For example, none of the infinitely many facts $not(i = s(i))$, $not(i = s(s(i)))$, ... is an equational consequence of the equations in Figure 1.[10] But this infinite set of facts does follow when the equations are supplemented by the axiom

assert *Nat* **generated by** 0, *s*

The intuitive content of this axiom is that each element of sort *Nat* is either 0 or $s^n(0)$, where n is a positive integer. This axiom is equivalent to the infinite set of first-order formulas[11]

$$(E[0] \wedge (\forall i{:}Nat)(E[i] \Rightarrow E[s(i)])) \Rightarrow (\forall j{:}Nat)E[j]$$

where E is an arbitrary equation.[12]

As described in Section 6.3, LP uses induction rules to generate subgoals to be proved for the basis and induction steps in proofs by induction. The command

prove $not(i = s(i))$ **by induction**

directs LP to begin a proof of the conjecture $not(i = s(i))$ by induction, in other words, to prove $not(0 = s(0))$ as the basis subgoal, and then to prove $not(s(ic) = s(s(ic)))$ as the induction subgoal using the induction hypothesis $not(ic = s(ic))$, where ic is a new constant introduced by LP to formulate the induction hypothesis and subgoal.

Similary, the assertion *Set* **generated by** *empty*, *insert* in Figure 3 provides an induction rule for the sort *Set* that is equivalent to the infinite set of axioms

$$(E[empty] \wedge (\forall e{:}Elem, s{:}Set)(E[s] \Rightarrow E[insert(e, s)])) \Rightarrow (\forall s{:}Set)E[s]$$

Users can specify multiple induction rules for a single sort. For example, given the declarations in Figure 3, the LP commands

set name ***setInduction2***
assert ***Set generated by empty***, *singleton*, $\cup$

change the current name prefix and then assert that all objects of sort *Set* can be generated by taking unions of singleton and empty sets. Users can choose either induction rule when attempting to prove an equation by induction; for example,

prove $x \subseteq x$ **by induction using** *setInduction2*

As described in Section 6.11, users may use one induction rule to prove another. For example, a user might choose to prove rather than assert the rule *setInduction2*.

[10] For each n, there is a model of the equations that contains the natural numbers plus an additional n elements that form a cycle under s and in which the relation $a < b$ is always false when a and b are among these n elements.

[11] Because LP interprets induction axioms by sets of first-order formulas, these axioms do not rule out the existence of nonstandard models, that is, of models that contain elements not of the form 0 or $s^n(0)$, but with the same first-order properties as these elements. Interpreting induction axioms by single second-order sentences would rule out nonstandard models, but would not necessarily increase the number of theorems that can be proved (because complete systems of inference do not exist for second-order logic).

[12] Since the set of first-order formulas corresponding to an induction axiom in LP involves arbitrary equations E, this set can become larger when new operators are declared.

4.7 Deduction rules

LP uses deduction rules to deduce new equations from existing equations and rewrite rules. For example, the LP command

assert when $i + j == i + k$ **yield** $j == k$

specifies a cancellation law for addition.[13] Logically, this deduction rule has the same meaning as the equation $i + j = i + k \Rightarrow j = k == true$, but there is an important operational difference: LP can apply the deduction rule directly to the equation $f(x) + c == g(x) + c$ to deduce the equation $f(x) == g(x)$. Section 8 contains a discussion of the pragmatic ramifications of the differences between expressing axioms as deduction rules and expressing them as implications.

More powerful deduction rules allow explicit universal quantification of variables in their hypotheses. For example, the LP command

assert when $(forall\ e)\ e \in x == e \in y$ **yield** $x == y$

defines a deduction rule equivalent to the universal-existential formula

$$(\forall x, y{:}Set)\,[((\forall e{:}Elem)(e \in x \Leftrightarrow e \in y)) \Rightarrow x = y]$$

of set extensionality. This deduction rule, which can also be asserted by the LP command **assert** S **partitioned by** $\in$, as was done in Figure 3, enables LP to deduce equations such as $x == x \cup x$ automatically from equations such as $e \in x == e \in (x \cup x)$. (Section 5.5 shows another way to obtain this conclusion using the deduction rule.)

Deduction rules can have multiple hypotheses and/or multiple conclusions. For example, the transitivity of $<$ can be formulated as a deduction rule with two hypotheses:

when $i < j,\ j < k$ **yield** $i < k$

An example of a deduction rule with two conclusions is the &-splitting law:

when $p\ \&\ q$ **yield** p, q

where p and q have been declared as variables of sort $Bool$.

A deduction rule can be applied to an equation or a rewrite rule. An application succeeds if there is a substitution that matches the deduction rule's first hypothesis to the equation or rewrite rule and that maps the variables in the **forall** clause to distinct variables[14] not appearing elsewhere[15] in the matched equation or rewrite rule.

The result of applying a deduction rule with one hypothesis is the set of equations obtained by instantiating each of its conclusions by the substitution(s) that matched its hypothesis. LP substitutes fresh variables for variables that occur in the range of the matching substitution, but not in the hypothesis. For example, applying the deduction rule **when** $P(x)$ **yield** $Q(x, y)$ to $P(f(y))$ produces the result $Q(f(y), y1)$ and not the weaker result $Q(f(y), y)$.

The result of applying a deduction rule with more than one hypothesis is the set of deduction rules obtained by deleting the first hypothesis and instantiating the remainder of the deduction rule by the substitution(s) that matched it. For example, applying the deduction rule **when** $x < y,\ y < z$ **yield** $x < z$

[13] Such cancellation laws generalize those used by Stickel [31] in reasoning about rings.

[14] If the matched variables were not required to be distinct, then, for example, the deduction rule **when** (**forall** x, y) $x * z == y * z$ **yield** $z == 0$ would apply to the equation $w * 1 == w * 1$ and yield the erroneous result $1 == 0$.

[15] The matched variables must not appear elsewhere lest, for example, the deduction rule for set extensionality apply to the equation $e \in insert(e, x) == e \in insert(e, y)$ to yield the erroneous result $insert(e, x) == insert(e, y)$.

to the equation $a < b$ yields the deduction rule **when** $b < z$ **yield** $a < z$.

Deduction rules serve to increase LP's logical power, to improve its performance, and to reduce the need for user interaction. Examples of deduction rules that serve the latter two purposes are the &-splitting law and the cancellation law for addition. The &-splitting law is so useful that it is built into LP to further improve performance.

LP automatically applies deduction rules to equations and rewrite rules whenever they are normalized. The sample proof in Appendix B illustrates the logical power of deduction rules, as well as the benefits of applying them automatically to additional hypotheses introduced in the course of a proof.

Like other facts in LP, deduction rules may be asserted as axioms or proved as theorems (cf. Section 6.10).

4.8 Built-in operators and axioms

LP provides built-in rewrite rules (see Figure 6) to simplify terms involving the Boolean operators *not*, &, |, $\Rightarrow$, and $\Leftrightarrow$, the overloaded equality operators =, and the overloaded conditional operators *if*.

$p \mathbin{\&} \mathit{true} \rightarrow p$	$p \mid \mathit{true} \rightarrow \mathit{true}$
$p \mathbin{\&} \mathit{false} \rightarrow \mathit{false}$	$p \mid \mathit{false} \rightarrow p$
$p \mathbin{\&} p \rightarrow p$	$p \mid p \rightarrow p$
$p \mathbin{\&} \mathit{not}(p) \rightarrow \mathit{false}$	$p \mid \mathit{not}(p) \rightarrow \mathit{true}$
$x = x \rightarrow \mathit{true}$	$p \Leftrightarrow p \rightarrow \mathit{true}$
$p = \mathit{true} \rightarrow p$	$p \Leftrightarrow \mathit{true} \rightarrow p$
$p = \mathit{false} \rightarrow \mathit{not}(p)$	$p \Leftrightarrow \mathit{false} \rightarrow \mathit{not}(p)$
$p = \mathit{not}(p) \rightarrow \mathit{false}$	$p \Leftrightarrow \mathit{not}(p) \rightarrow \mathit{false}$
$\mathit{not}(\mathit{true}) \rightarrow \mathit{false}$	$\mathit{if}(\mathit{true}, p, q) \rightarrow p$
$\mathit{not}(\mathit{false}) \rightarrow \mathit{true}$	$\mathit{if}(\mathit{false}, p, q) \rightarrow q$
$\mathit{not}(\mathit{not}(p)) \rightarrow p$	$\mathit{if}(\mathit{not}(p), x, y) \rightarrow \mathit{if}(p, y, x)$
	$\mathit{if}(p, \mathit{true}, q) \rightarrow p \mid q$
$\mathit{true} \Rightarrow p \rightarrow p$	$\mathit{if}(p, \mathit{false}, q) \rightarrow \mathit{not}(p) \mathbin{\&} q$
$\mathit{false} \Rightarrow p \rightarrow \mathit{true}$	$\mathit{if}(p, q, \mathit{true}) \rightarrow p \Rightarrow q$
$p \Rightarrow \mathit{true} \rightarrow \mathit{true}$	$\mathit{if}(p, q, \mathit{false}) \rightarrow p \mathbin{\&} q$
$p \Rightarrow \mathit{false} \rightarrow \mathit{not}(p)$	$\mathit{if}(p, x, x) \rightarrow x$
$p \Rightarrow p \rightarrow \mathit{true}$	
$p \Rightarrow \mathit{not}(p) \rightarrow \mathit{not}(p)$	
$\mathit{not}(p) \Rightarrow p \rightarrow p$	
Notes: p, q, and r are variables of sort *Bool*. x and y are variables of an arbitrary sort.	

Figure 6: Rewrite rules built into LP

These rewrite rules are sufficient to prove many, but not all, identities involving these operators. Unfortunately, the sets of rewrite rules for propositional logic that are known to be *complete* (i.e., to be convergent and to yield all propositional identities) require exponential time and space [18, 32]. Furthermore, they can expand, rather than simplify, conjectures that do not reduce to identities. These are serious drawbacks, because when we are debugging specifications we often attempt to prove conjectures that are not true. So a complete set of rewrite rules for propositional logic is not built into LP. Instead, LP provides proof mechanisms that can be used to overcome incompleteness in a rewriting system, and it allows users to add any of the complete sets (which subsume the built-in rewrite rules) when they wish to use them.

LP also provides a built-in metarule for rewriting terms containing the conditional operator *if*. This metarule has the form

$$if(t_1, t_2[t_1], t_3[t_1]) \rightarrow if(t_1, t_2[true], t_3[false])$$

and can be applied when t_1 occurs as a subterm of t_2 or t_3. For example, LP uses this metarule to reduce the term $if(p,\ p \,\&\, q,\ p \mid r)$ to $if(p, q, r)$.

LP treats &, |, and $\Leftrightarrow$ as **ac** operators, and it treats =, in all overloadings, as a **commutative** operator. Finally, LP provides the built-in deduction rules shown in Figure 7.[16]

lp_not_is_true:	**when** $not(p)$	**yield** $p == false$
lp_not_is_false:	**when** $not(p) == false$	**yield** p
lp_and_is_true:	**when** $p \,\&\, q$	**yield** p, q
lp_or_is_false:	**when** $p \mid q == false$	**yield** $p == false, q == false$
lp_iff_is_true:	**when** $p \Leftrightarrow q$	**yield** $p == q$
lp_equals_is_true:	**when** $x = y$	**yield** $x == y$

Figure 7: Deduction rules built into LP

4.9 Orienting equations into rewrite rules

Ordinarily, LP automatically orients equations into rewrite rules without users having to enter explicit ordering commands. However, the **set automatic-ordering off** command causes LP to refrain from orienting equations until it receives an explicit **order** command.

LP provides three types of ordering mechanisms for orienting equations into rewrite rules. The command **set ordering** *method* can be used to select any of these mechanisms.

- Two registered orderings (the **dsmpos** and **noeq-dsmpos** orderings), based on LP-suggested partial orderings of operators [6, 8], that guarantee termination of sets of rewrite rules when no commutative or associative-commutative operators are present.
- A **polynomial** ordering, based on user-supplied polynomial interpretations of operators [1], that guarantees termination even when commutative or associative-commutative operators are present. Unfortunately, this powerful mechanism is difficult to use.
- Three "brute-force" ordering procedures, which give users complete control over whether equations are oriented from left to right or from right to left. These provide no guarantees about termination.

[16] *lp_and_is_true* and *lp_or_is_false* could both be written with single conclusions, because & and | are commutative.

Most users rely on LP's registered orderings to order all equations; **noeq-dsmpos** is the default ordering. In striking contrast to the brute-force methods, they hardly ever cause difficulties by producing a nonterminating set of rewrite rules.

4.9.1 Registered orderings

LP's registered orderings use information in a *registry* to orient equations. When no **ac** or **commutative** operators are involved, these orderings guarantee that the resulting rewrite rules terminate. There are two kinds of information in a registry: height information and status information.

Height information relates pairs of operators. If an operator f has greater height than another operator g, LP will attempt to orient equations containing f and g into rewrite rules that replace an occurrence of f by one or more occurrences of g. For example, $g(g(x)) == f(x)$ will be oriented into the rewrite rule $f(x) \to g(g(x))$.

Status information assigns relative weights to the arguments of operators with arity greater than one. If an operator h has **left-to-right** (**right-to-left**) status, more weight is assigned to h's leftmost (rightmost) arguments. For example, if h has left-to-right status, $h(f(x), x) == h(x, f(x))$ will be oriented into the rule $h(f(x), x) \to h(x, f(x))$, whereas if h has right-to-left status it will be oriented into $h(x, f(x)) \to h(f(x), x)$. If an operator has **multiset** status, its arguments are given equal weight. If h has multiset status, the equation $h(f(x), x) == h(x, f(x))$ cannot be oriented. LP automatically assigns multiset status to **ac** and **commutative** operators.

Figure 8 shows how the **register** command can be used to place information in the registry and how that information constrains the way in which equations are oriented. As discussed in Section 8, the **register** command can also be used to enhance performance.

Command	Effect on ordering
register height $f > g$	rewrite f to g
register height $f = g$	give f and g equal height
register height $f \geq g$	rule out $g > f$
register bottom f	rewrite any non-bottom operator to f
register top f	rewrite f to any non-top operator
register status right-to-left f	assign more weight to f's right arguments
register status left-to-right f	assign more weight to f's left arguments
register status multiset f	assign equal weight to all arguments of f

Figure 8: LP commands for supplying ordering constraints

Information about the relative height of operators can be combined in a single command such as

register height $\Rightarrow\ >\ (\&, |)\ >\ true\ =\ false$

which suggests that $\Rightarrow$ be rewritten to either & or |, and that each of these be rewritten to *true* or *false*, which have the same height. The partial ordering on operators is transitively closed (so that $\Rightarrow\ >\ true$ is a consequence of this command). LP rejects **register** commands that do not represent a consistent addition to the registry, for example, commands that imply both $f > g$ and $g > f$.

When the current registry does not contain enough information to orient an equation, LP will generate minimal sets of extensions to the registry, called *suggestions*, that would permit the equation to be oriented. It will not generate suggestions that would cause an equation entered by the user with $\to$ instead of $==$

to be oriented from right to left. Furthermore, the **noeq-dsmpos** ordering does not generate suggestions assigning equal heights to two operators; as a result, it is faster, but less powerful than **dsmpos**.

Ordinarily, LP adds suggestions automatically to the registry when needed. These actions can be overridden by the command **set automatic-registry off**, which directs LP to ask the user to choose a suggestion to be added to the registry. For example, when asked to orient $f(a,b) == f(b,a)$ with an empty registry, LP presents the user with the following suggestions for adding height and status information to the registry. Section 4.9.4 discusses how to respond to such suggestions.

```
        Direction   Suggestions
        ---------   -----------
   1.      ->       a > b     f(L)
   2.      ->       b > a     f(R)
   3.      <-       b > a     f(L)
   4.      <-       a > b     f(R)
```

Had the equation been entered as $f(a,b) \rightarrow f(b,a)$, LP would have presented only the first two suggestions.

In addition to registering height and status information, a user may register operators as **top** or **bottom** operators. This does not immediately extend the height relation. When LP attempts to orient an equation that cannot be oriented with the current registry, but can be oriented by adding height relations that make non-top operators less than top operators, or non-bottom operators greater than bottom operators, LP will automatically extend the registry by adding such height relations, even if **automatic-registry** is off. Furthermore, registering an operator as bottom prevents LP from automatically extending the registry by making that operator greater than a non-bottom operator, and registering an operator as top prevents LP from automatically making that operator less than a non-top operator. However, unless it contradicts the current height relation, users may explicitly introduce relations in which bottom operators are greater than non-bottom ones and top operators are less than non-top ones.

The **unregister** command allows users to delete the entire registry, or to remove operators from the bottom or top of the registry, but not to remove height or status information in the registry (see Section 7.4).

4.9.2 Polynomial orderings

The polynomial ordering requires considerable user input. It is generally used only to experiment with termination proofs of small sets of rewrite rules, not to orient large sets of equations into rewrite rules.

The **polynomial** ordering is based on user-supplied interpretations of operators by sequences of polynomials [1]. The ordering extends these interpretations to terms by interpreting a variable by a sequence of identity polynomials and a compound term by the interpretation of its root operator applied to the interpretations of its arguments. One term is less than another in the polynomial ordering if its interpretation is lexicographically less than that of the second term (one polynomial is less than another if its value is less than that of the other for all sufficiently large values of its variables).

The command **set ordering polynomial *length*** sets the current ordering to a polynomial ordering based on sequences containing *length* polynomials; if no *length* is specified, it is assumed to be 1.

The command **register polynomial** f *polynomials* assigns the sequence of *polynomials* as the polynomial interpretation of f. The polynomials are entered like standard LP terms separated by spaces, using the binary operators $+$, $*$, ˆ (for exponentiation), variables, and positive integer coefficients. LP understands operator precedence for terms representing polynomials, so these terms need not be fully parenthesized.

If the sequence of polynomials associated with an operator is longer than the length of the current polynomial ordering, the extra polynomials are ignored. If it is shorter, it is extended by replicating its last element.

The commands in Figure 9, if issued before asserting the axioms in Figure 1, cause LP to use the polynomial ordering to prove that set of rewrite rules shown in Figure 2 terminates. For example, they cause LP to orient $s(i) + j == s(i + j)$ from left to right, because the polynomial interpretation $(i + 2) * j$ of the left side dominates the interpretation $i * j + 2$ of the right side when j is sufficiently large, for example, when $j > 1$. The **noeq-dsmpos** ordering produces the same set of rewrite rules as this polynomial ordering, but does not guarantee that they terminate, because $+$ is **ac**.

set ordering polynomial		
register polynomial	0	2
register polynomial	s	$x + 2$
register polynomial	$+$	$x * y$
register polynomial	$<$	$x * y$

Figure 9: Sample polynomial interpretation

4.9.3 Brute-force orderings

These orderings provide no guarantee about termination.

The **manual** ordering causes LP to ask the user how to orient each equation. The user is allowed to choose either orientation, provided it results in a valid rewrite rule, that is, provided that the left side of the resulting rewrite rule does not consist of a variable and that the right side does not introduce a variable not present in the left side.

The **left-to-right** ordering causes LP to orient equations into rewrite rules from left to right, provided the results are valid rewrite rules. The **either-way** ordering behaves like the **left-to-right** ordering, except that it orients an equation into a rewrite rule from right to left if that is possible and left to right is not.

4.9.4 Interacting with the ordering procedures

When **automatic-ordering** is **off**, users must issue explicit **order** commands to cause LP to orient equations into rewrite rules. When **automatic-registry** is **off**, LP will prompt users to confirm any extensions to the registry when a registered ordering is in use, or to select an action for an equation LP is unable to orient. When presented with a prompt like

```
The following sets of suggestions will allow the equation to be
ordered:

    Direction   Suggestions
    ---------   -----------
1.     ->       a > b
2.     <-       b > a

What do you want to do with the equation?
```

users can type **?** to see a menu such as

```
 Enter one of the following, or type <ret> to exit.
   accept[1..2]   kill           postpone
   divide         left-to-right  right-to-left
   interrupt      ordering       suggestions
```

of possible responses, which have the following effects.

- *accept* (or a number in the indicated range): confirms the first (or the selected) extension to the registry. If this action is missing from the menu, no extension to the registry will orient the equation.
- *divide:* causes LP to add two new equations that imply the original equation. This action is useful when an equation such as $x/x == y/y$ cannot be oriented because each side contains a variable not in the other side. If the user elects to divide this equation, LP will ask the user to supply a name for a new operator, for example, e; it will then declare the operator and assert two equations, $x/x == e$ and $y/y == e$, both of which can be oriented (by making / higher than e) and which normalize the original equation to an identity.
- *interrupt:* interrupts the ordering process and returns LP to command level.
- *kill:* deletes the problematic equation from the system. This should be used with caution, since it may change the theory associated with the current logical system.
- *left-to-right:* orients the equation from left-to-right without extending the registry. Doing this removes any guarantee of termination.
- *ordering:* displays the current registry (as does **display ordering** at the command level) and prompts the user for another response.
- *postpone:* defers the attempt to orient this equation. Whenever another equation is successfully oriented, all postponed equations are re-examined, since they may have been normalized into something more tractable.
- *right-to-left:* orients the equation from right-to-left without extending the registry. Doing this removes any guarantee of termination.
- *suggestions:* redisplays the LP-generated suggestions for extending the registry and prompts the user for another response.

5 Forward inference in LP

LP provides mechanisms for proving theorems using both forward and backward inference. Forward inferences (discussed in this section) produce consequences from a logical system. Backward inferences (discussed in the next) produce a set of subgoals whose proof will suffice to establish a conjecture. Appendix B illustrates the use of both kinds of inference in a sample proof.

LP provides four methods of forward inference, each of which uses the axioms in LP's logical system to deduce new facts.

5.1 Normalization

Whenever a new rewrite rule is added to its logical system, LP automatically renormalizes all equations, rewrite rules, and deduction rules.[17] If an equation or rewrite rule normalizes to an identity, it is discarded. If all hypotheses of a deduction rule normalize to identities, the deduction rule is replaced by the equations in its conclusions. If all conclusions of a deduction rule normalize to identities, the deduction rule is discarded.

5.2 Application of deduction rules

Whenever a new deduction rule is added to its logical system, LP automatically applies that deduction rule to all equations and rewrite rules in its system. Likewise, whenever an equation or rewrite rule in its system is normalized, LP automatically applies all deduction rules in its system to the new normal form.[18] As described in Section 4.7, these actions can produce equations (in the case of single-hypothesis deduction rules) or deduction rules (in the case of multiple-hypothesis deduction rules). To increase the likelihood that the hypothesis of a deduction rule will match an equation (or a rewrite rule), LP normalizes both the deduction rule and the equation (or the rewrite rule) before attempting to apply the deduction rule.

5.3 Critical-pair equations

A common problem arises when a set E of equations is oriented into a rewriting system R, namely, $\rightsquigarrow_R$ is not convergent, and hence reduction to normal form does not provide a decision procedure for the equational theory of E. Consider, for example, the rewrite rules

group.1: $(x * y) * z \rightarrow x * (y * z)$
group.2: $i(x) * x \rightarrow e$
group.3: $e * x \rightarrow x$

produced by orienting the axioms for groups given in Figure 4. These rewrite rules can be used to reduce the term $(i(y) * y) * z$ to a terminal form in either of two ways. Applying rule *group.1* produces the terminal form $i(y) * (y * z)$. Applying rule *group.2* produces $e * z$, which rule *group.3* reduces to the terminal form z. These two terminal forms, $i(y) * (y * z)$ and z, are equivalent under the equational theory of the group axioms, but the rewrite rules *group.1:3* do not reduce them to a common terminal

[17] Ways of preventing automatic renormalization are discussed in Section 5.6.
[18] Ways of preventing deduction rules from being applied automatically are discussed in Section 5.6.

form. Likewise, as shown in Figure 5, $i(e) == e$ is an equational consequence of these three axioms; yet $i(e)$ and e are distinct terminal forms.

Nonconvergent rewriting systems can cause LP to exhibit even stranger behaviors. For example, LP may fail to reduce two terms u and v to the same normal form even though $u \rightsquigarrow v$. Worse yet, the behavior of LP may be nonmonotonic; in other words, it may reduce u and v to the same normal form using the rewriting system R but not using the system $R \cup \{l \to r\}$.

The **critical-pairs** command provides a method of extending the rewriting theory of a system to more nearly approximate its equational theory. The command

crititical-pairs ***group.1 with group.2***

causes LP to compute (in a manner more fully described below) the *critical-pair equation* $i(y)*(y*z) == e*z$ (whose left and right sides are the results of reducing $(i(y)*y)*z$ by rules *group.1* and *group.2*, respectively), which is then reduced by rule *group.3* and oriented to give a new rewrite rule, *group.4*: $i(y)*(y*z) \to z$.

The critical-pair computation involves unification, which generalizes matching. Recall that a substitution σ matches a term t_1 to a term t_2 if $\sigma(t_1)$ is identical to t_2. It *unifies* t_1 and t_2 (or is a *unifier* of t_1 and t_2) if $\sigma(t_1)$ is identical to $\sigma(t_2)$. If E is a set of equations, σ unifies t_1 and t_2 *modulo* E (or is an E*-unifier* of t_1 and t_2) if $\sigma(t_1) == \sigma(t_2)$ is in the equational theory of E.

There may be no substitutions, or many substitutions, that unify a pair of terms. For example, the terms $x*y$ and $i(x)$ cannot be unified, and the terms $x*y$ and $i(w)*w$ are unified by the substitution

$$\sigma = \{i(w)\ for\ x,\ w\ for\ y\}$$

and also by the substitution

$$\sigma' = \{i(e)\ for\ x,\ e\ for\ y,\ e\ for\ w\}$$

which is an instance of σ, since $\sigma' = \{e\ for\ w\} \circ \sigma$. Here the *composition* $\sigma \circ \tau$ of two substitutions σ and τ is the substitution defined by $(\sigma \circ \tau)(t) = \sigma(\tau(t))$.

For ordinary unification (i.e., unification modulo an empty set of equations), if two terms can be unified, they always have a unique (up to variable renaming) *most general unifier*. That is, any unifiable terms s and t have a unifier σ such that, for each unifier θ of s and t, there exists a substitution τ such that $\theta = \tau \circ \sigma$. For many equational theories, there is not always a most general E-unifier. For the commutative and associative-commutative theories, there are instead finite sets of *minimal unifiers*, that is, unifiers that are not substitution instances (except for variable renaming) of other unifiers.

LP uses unification to compute critical-pair equations, as follows. Let $l_1 \to r_1$ and $l_2 \to r_2$ be rewrite rules such that l_2 can be unified with a nonvariable subterm t_1 of l_1.[19] When such a substitution exists, we say that l_1 and l_2 *overlap* at t_1. Let σ be the most general unifier (or one of the minimal unifiers, in the case of E-unification) of l_2 and t_1. The *critical-pair equation* associated with this overlap is $\sigma(l_1[t_1 \leftarrow r_2]) == \sigma(r_1)$. (The notation $t[t_1 \leftarrow s]$ stands for t with the subterm t_1 replaced by s.) One way to think of this critical-pair equation is as the result of reducing $\sigma(l_1)$ by each of the two rewrite rules.

Each critical-pair equation captures a way in which a pair of rewrite rules (or two different applications of the same rewrite rule) might be used to reduce a single term in two different ways. For example, the substitution $\{i(y)$ for x, y for $w\}$ unifies $i(w)*w$ with a nonvariable subterm of $(x*y)*z$, so that $e*z == i(y)*(y*z)$ is a critical-pair equation between $(x*y)*z \to x*(y*z)$ and $i(w)*w \to e$.

[19]For simplicity, we assume that the rewrite rules have no variables in common. If they do, the variables are renamed.

When associative-commutative operators are present, it is necessary to generalize the critical-pair computation to capture further ways in which a single term can be reduced. See Appendix A.4 for a discussion of how this is done.

The command

critical-pairs *names1 with names2*

causes LP to compute all critical-pair equations between the rewrite rules named by *names1* and those named by *names2*. Critical-pair equations that reduce to identities are discarded; the others are added to LP's logical system and processed as if they had been asserted by the user. For example, starting from the rewrite rules for group theory, the command **critical-pairs** $*$ **with** $*$ causes LP to deduce the critical-pair equation $e * z == i(y) * (y * z)$, which reduces to $z == i(y) * (y * z)$. These equations appear on lines 5 and 6 in Figure 5. Repeating this command causes LP to deduce the critical-pair equation on line 8 in Figure 5, and repeating it a third time causes LP to deduce to the critical-pair equation on line 10, which reduces to the equation $i(e) == e$. Thus, three applications of the **critical-pairs** command suffice to enable LP to deduce the equation $i(e) == e$ from the axioms for group theory.

5.4 Completion

The **complete** command causes LP to compute critical-pair equations, and to orient them into rewrite rules, until there are no nontrivial critical-pair equations between any pair of rewrite rules in the system.[20] If the computation finishes with an empty set of equations and a terminating set of rewrite rules, then that set of rewrite rules provides a decision procedure (using reduction to normal form) for its equational theory. For example, the completion procedure produces the rewrite rules shown in Figure 10 from the axioms for groups given in Figure 4; these rewrite rules are sufficient to reduce any equation that is true about all groups to an identity.

$(x * y) * z \to x * (y * z)$	$i(e) \to e$
$i(x) * x \to e$	$i(i(z)) \to z$
$e * x \to x$	$z * i(z) \to e$
$i(y) * (y * z) \to z$	$x * (i(x) * z) \to z$
$z * e \to z$	$i(y * z) \to i(z) * i(y)$

Figure 10: A complete set of rewrite rules for group theory

Appendix A provides more details concerning the computation of critical-pair equations and the completion procedure.

When using LP, we rarely complete our rewriting systems, because a complete set of rewrite rules with the same equational theory may not exist, may be too expensive to obtain, may be too expensive to use, or may lead to canonical forms that are hard to read. However, we often make selective use of critical-pair equations to derive useful consequences. We also use the completion procedure to look for inconsistencies, and we interrupt it if none are found after a few iterations. During proofs, both the **critical-pairs** and the **complete** commands stop computing critical-pair equations when they produce a

[20]Release 2.2 of LP does not compute critical-pair equations between the built-in rewrite rules and the other rewrite rules in the system. As a result, the completion procedure may not discover some "obvious" consequences of facts that contain built-in operators. Users can overcome this deficiency by explicitly asserting or proving an appropriate set of immune (see Section 5.6) rewrite rules for the booleans, but even then the completion procedure may not discover some "obvious" consequences because the built-in rewrite rules do not axiomatize all properties (e.g., distributivity) of the boolean operators.

consequence that results in normalizing the current conjecture to an identity. This makes these commands convenient for finishing up proofs.

5.5 Instantiation

Explicit instantiation of variables in equations, rewrite rules, and deduction rules is the final method of forward inference in LP. The command

instantiate *variable by term, . . . , variable by term in names*

causes LP to substitute (simultaneously) the specified terms for variables in the named equations, rewrite rules, and deduction rules.

A common use of the instantiate command is in connection with deduction rules. For example, given a logical system that contains the deduction rule

when (**forall** e) $e \in x == e \in y$ **yield** $x == y$

and the rewrite rule $e \in (x \cup y) \rightarrow e \in x \mid e \in y$, instantiating y by $x \cup x$ in the deduction rule produces the conclusion $x == x \cup x$.

Sometimes it is helpful to instantiate an equation to obtain an instance that is orientable even though the original equation is not. For example, the equation $insert(e, insert(e', x)) == insert(e', insert(e, x))$ cannot be oriented into a terminating rewrite rule, but specific instances, such as the equation $insert(0, insert(s(0), x)) == insert(s(0), insert(0, x))$, can be oriented.

Instantiation can sometimes be used as an alternative to computing critical-pair equations For example, given the rewrite rules

group.1: $(x * y) * z \rightarrow x * (y * z)$
group.2: $i(x) * x \rightarrow e$
group.3: $e * x \rightarrow x$

for groups, the command

instantiate x by $i(y)$ **in *group.1***

causes LP to generate the equation $(i(y) * y) * z == i(y) * (y * z)$, which LP reduces (using rules *group.2:3*) and orients into the rewrite rule *group.1.1*, $i(y) * (y * z) \rightarrow z$. Whenever a rewrite rule is instantiated, the initial form of the instantiation can be reduced to an identity by a single application of the original rewrite rule. To avoid this trivial reduction, LP attempts to reduce the instantiation first using the other rewrite rules in the system. Often such a reduction produces an equation that can no longer be reduced by the original rewrite rule, and which LP therefore retains. At other times, the instantiation may be rewritten to an identity by the original rewrite rule even after it has been reduced by some other rule, or because no other rewrite rule could be applied. If it is useful to retain instantiations such as these, LP's *ancestor immunity* facility (see Section 5.6) provides a means of doing so.

An advantage of the **critical-pairs** command is that, in effect, it finds potentially useful instantiations automatically: the command

crititical-pairs $*$ **with** $*$

produces the same rewrite rule as the instantiation (although with the name *group.4*). Furthermore, the **critical-pairs** command can produce several potentially useful consequences from a pair of equations, whereas the **instantiate** command produces but a single consequence.

5.6 Activity and immunity

Ordinarily, all rewrite rules and deduction rules are "active" in that LP will apply them automatically to normalize other facts and to deduce consequences from these facts. Likewise, all facts are ordinarily subject to normalization and deduction. LP provides features, however, for restraining such automatic inferences.

Users can "deactivate" rewrite rules and deduction rules to prevent them from being applied automatically. The command **make passive *names*** deactivates the facts named by *names*. When a rule is known to be inapplicable or expensive to apply, it can be deactivated to prevent LP from spending time trying to apply it. The **display** command indicates passive facts by printing the letter P in parentheses following their names. The command **make active *names*** reactivates the facts named by *names*.

Users can also "immunize" equations, rewrite rules, and deduction rules to protect them from automatic normalization or deduction, either to enhance the performance of LP or to aid in proofs. The command **make immune *names*** immunizes the facts named by *names*. When facts are known to be irreducible, they can be immunized to prevent LP from wasting time trying to reduce them. For example, when reasoning about a large set of axioms that contains axioms for the integers, one might want to immunize all axioms that deal only with the integers. Immunization can also preserve facts for later use (e.g., instantiation) when they might otherwise reduce to identities and disappear. Immunity does not prevent facts from being flattened or equations from being oriented. The **display** command indicates immune facts by printing the letter I in parentheses following their names. The command **make nonimmune *names*** deimmunizes the facts named by *names*.

In addition to the **make** command, which enables users to change the activity or immunity of existing facts, LP provides two settings that determine the activity and immunity of newly entered or generated facts. These settings can be changed by the **set activity** and **set immunity** commands. By default, **activity** is **on** and **immunity** is **off**.

Facts can also be given an intermediate degree of immunity. The commands **set immunity ancestor** and **make ancestor-immune *names*** prevent facts from being reduced or subjected to deduction by rules that are ancestors of the fact. One fact is an *ancestor* of another if its name is a prefix of the other's. For example, a rewrite or deduction rule named a.1 is an ancestor of a rewrite rule named a.1.2 obtained from it by instantiation, and LP will not apply it automatically to a.1.2 if a.1.2 is ancestor-immune. Thus, ancestor immunity provides a way to preserve instantiations of rewrite rules. The command **make ancestor-immune *names*** causes any named facts that were fully immune to become only ancestor-immune. The **display** command indicates ancestor-immune facts by printing the letter i in parentheses following their names.

LP maintains the following invariants. All terms in nonimmune equations, rewrite rules, and deduction rules are normalized with respect to all active rewrite rules. All terms in ancestor-immune facts are normalized with respect to all active rewrite rules other than their ancestors. All active deduction rules have been applied to all nonimmune equations and rewrite rules and to all ancestor-immune equations and rewrite rules that do not have that deduction rule as an ancestor.

LP orients inactive and/or immune equations into inactive and/or immune rewrite rules, which remain that way until explicitly activated or deimmunized.

Rewrite rules (passive or not) can be applied explicitly to selected facts (immune or not) by the commands **normalize *factNames with ruleNames*** and **rewrite *factNames with ruleNames***. The first normalizes each of the named facts using the named rewrite rules; the second rewrites if possible some subterm in each of the named facts using one of the named rewrite rules, if possible. (Rewrite rules named by both *factNames* and *ruleNames* are not normalized or rewritten, because they would reduce themselves to identities.) If **with *ruleNames*** is omitted, all rewrite rules in the system are used. These commands can be used to control when definitions are expanded, or when nonsimplifying rewrite rules (such as distributivity) are applied. The commands **normalize conjecture with *ruleNames*** and **rewrite conjecture with *ruleNames*** apply the named rules to the current conjecture.

Similarly, deduction rules (passive or not) can be applied explicitly to selected equations and rewrite rules (immune or not). The command **apply *ruleNames to factNames*** applies each of the named deduction rules once to each of the named equations and rewrite rules.

When executing the **rewrite**, **normalize**, and **apply** commands, LP begins by making lists of the existing facts named by *ruleNames* and *factNames*. New facts produced during execution of these commands are not added to these lists, even if their names happen to fall within *ruleNames* or *factNames*. However, re-execution of the commands will take these new facts into account.

6 Backward inference in LP

The **prove** command causes LP to initiate a proof of a conjecture by backward inference. The arguments to this command consist of the conjecture and, optionally, the proof method to be used. If no method is supplied, LP uses one from the current list of automatic proof methods (see Section 6.9).

LP maintains a stack of conjectures and subgoals whose proofs are not yet complete. It responds to the **prove** command by pushing the new conjecture on this stack and, depending on the proof method being used, by generating a set of subgoals to be proved and pushing them on the stack as well. These subgoals are lemmas that together are sufficient to imply the conjecture. LP may also generate additional hypotheses that can be used to prove particular subgoals, for example, an induction hypothesis in the induction step of a proof by induction.

The **display proof-status** command (or **dis p** for short) displays the status of all pending proofs. It shows the inference methods and additional hypotheses being used in these proofs, as well as the progress that has been made.

The conjecture on top of LP's proof stack is known as the *current conjecture*. LP tries to make as much progress as it can on the current conjecture before requesting more input from the user. Whenever a proof gets stuck, LP prompts the user to supply additional commands. The user can use the **prove** command to initiate a proof of a lemma that might be useful in the suspended proof, cancel the proof attempt with the **cancel** command, or resume the proof of the current conjecture with the **resume** command. When the user cancels the proof of a subgoal for a conjecture, LP also cancels the proofs of any other subgoals introduced at the same time, and it resets the proof method for the parent conjecture. The **cancel all** command cancels all proof attempts. Like the **prove** command, the **resume** command takes a proof method as an optional argument.

Whenever a proof terminates or is canceled, LP pops the stack of conjectures and restores its logical system to its state before work on the conjecture began (thereby discarding any lemmas proved while working on the conjecture). If the conjecture was proved, LP adds it to its logical system and resumes work on the conjecture now on top of the stack.[21] As soon as it can establish the current conjecture, LP terminates any forward inference mechanisms (such as internormalization of the rewriting system or the computation of critical-pair equations) that may be in progress.

LP prints a line beginning with <> *number* whenever it creates *number* subgoals in a proof, and it prints a line beginning with [ ] whenever it finishes the proof of a conjecture or subgoal. After a successful proof, the number of [ ]'s equals the number of **prove** commands plus the number of subgoals that were created by LP. LP also annotates script files with [ ]'s and <>'s. If a command file is executed with **box-checking** set **on** (see Section 7.3.2), LP will check that proofs are proceeding in accordance with these annotations. Whenever LP generates <> *number* or [ ], it checks that the next nonblank line in the file being executed begins with a confirming <> or [ ]. If it does not, LP prints an error message and halts execution of the file; LP also treats the occurrence of an unexpected <> or [ ] as an error.

The **qed** command can also be used to confirm that a proof is proceeding as expected. LP treats the occurrence of **qed** as an error if any conjectures still remain to be proved.

Conjectures are assigned the default activity and immunity when they are created by the **prove** command; their activity and immunity can be changed by the **make** command. Immune conjectures are not immune during an attempt to prove them, but are added to the system as immune facts in their original form when

[21] More flexible systems of proof management in future releases of LP will enable users to work on any unproved conjecture, not just the one on top of the stack, and to prove lemmas in any context.

proved. Immunizing a conjecture provides a way to prevent it from disappearing once proved (because it normalizes to an identity). Inactive conjectures remain that way when proved.

There are six methods of backward inference for proving equations in LP. In addition, LP provides automatic methods of backward inference for proving deduction rules and induction rules.

6.1 Proofs by normalization

LP uses active rewrite rules to normalize conjectures. If a conjecture normalizes to an identity, it is a theorem. Otherwise the normalized conjecture becomes the current subgoal to be proved. For example, LP succeeds in proving the conjecture $singleton(e) \subseteq insert(e, s)$ by using the axioms in Figure 3 to reduce it to an identity. But the conjecture $x \subseteq x$ is irreducible, and so becomes the current subgoal to be proved by some other proof method.

Passive rewrite rules can be applied explicitly to a conjecture by the commands **normalize conjecture with *names*** and **rewrite conjecture with *names***. The first normalizes the current conjecture using the named rewrite rules. The second rewrites the conjecture at most once; if the conjecture contains a subterm that can be rewritten using one of the named rewrite rules, one such subterm is rewritten. If **with *names*** is omitted, all rewrite rules in the system are used. These commands can be used to control when definitions are expanded, or when nonsimplifying rewrite rules (such as distributivity) are applied.

6.2 Proofs by cases

Conjectures can often be simplified by dividing a proof into cases. When a conjecture reduces to an identity in all cases, it is a theorem. Case analysis has two primary uses. If a conjecture is a theorem, a proof by cases may circumvent a lack of completeness in the rewrite rules. If a conjecture is not a theorem, an attempted proof by cases may simplify the conjecture and make it easier to understand why the proof is not succeeding.

The command **prove** e **by cases** $t_1, \ldots, t_n$, where $t_1, \ldots, t_n$ are boolean terms, directs LP to prove an equation e by division into cases $t_1, \ldots, t_n$ (or into two cases, t_1 and $not(t_1)$, if $n = 1$). LP's actions in response to this command are simplest when the terms t_i contain no variables. For example, given the axioms

order.1: $not(x < x)$
order.2: $(x < y \,\&\, y < z) \Rightarrow x < z$
order.3: $0 < x \mid 0 = x$
order.4: $x < f(x)$

and the command

prove $0 < f(c)$ **by case** $c = 0$

(where c is a constant), LP responds first by adding $c = 0$ as a case hypothesis and proving the subgoal $0 < f(c)$ (by normalizing it to an identity using the case hypothesis and *order.4*). LP then adds $not(c = 0)$ as a case hypothesis, and attempts to establish the same subgoal. The user can finish the proof by entering the **complete** command, which causes LP to generate a sequence of critical-pair equations: $0 < c$ between the case hypothesis (which LP has oriented into the rewrite rule $c = 0 \rightarrow false$) and *order.3*, $c < z \Rightarrow 0 < z$ between the new equation and *order.2*, and finally $true \Rightarrow 0 < f(c)$ between the newest equation and *order.4*. The command

prove $0 < f(c)$ **by case** $f(c) = 0$

can also be used to prove the same result. The **complete** command can be used to show that the case $f(c) = 0$ is impossible (by establishing $c < 0$, which leads to a contradiction). In the case $not(f(c) = 0)$, the **complete** command, or the command **critical-pairs** $*$*CaseHyp* **with** $*$, causes LP to establish the subgoal immediately using *order.3*.

When one of the terms t_i contains variables, LP must proceed more cautiously, because introducing case hypotheses that contain variables can lead to unsound reasoning. For example, if LP responded to the command

prove $f(x) == 0$ **by case** $x = 0$

by generating the case hypotheses $x = 0$ and $not(x = 0)$, then it would erroneously establish the conjecture by showing both cases to be impossible. As discussed in Section 4.3, both of these equations are inconsistent. The problem is that the variable x in the case hypothesis $x = 0$ is not meant to be quantified universally, but to represent the same element as the variable x in the conjecture $f(x) == 0$. To achieve this effect, and thereby preserve soundness, LP generates a new constant for each variable that occurs in the case hypotheses. It then substitutes these constants for the corresponding variables in both the case hypotheses and in the conjecture. For example, LP responds to the command

prove $0 < f(x)$ **by case** $x = 0$

by replacing x by a new constant xc and attempting to prove the subgoal $0 < f(xc)$ from each of the case hypotheses $xc = 0$ and $not(xc = 0)$. The proofs of the subgoals proceed exactly as above, with xc replacing c. Once the suboals have been established for the arbitrarily chosen (but fixed) value represented by xc (i.e., without using any properties of xc other than those expressed by the case hypotheses), LP soundly concludes that the conjecture itself holds for arbitrary values of x. Note that the conjecture $0 < f(x)$ is stronger than the conjecture $0 < f(c)$, because the former asserts that 0 is less than $f(x)$ for any value of the variable x, whereas the latter asserts only that 0 is less than $f(c)$ for the single value of the constant c.

In summary, LP responds as follows to the command **prove** e **by cases** $t_1, \ldots, t_n$. When $n > 1$, the first subgoal is to prove $t_1 \mid \ldots \mid t_n$. When $n = 1$, LP generates a default second case of $not(t_1)$, but does not generate this first subgoal. Then, for each case t_i, LP generates a subgoal e_i' and a hypothesis t_i'. These are formed from t_i and e by substituting new constants for the variables that occur in t_i. Section 4.1 describes the conventions used to name these constants. If an inconsistency results from adding a case hypothesis t_i', that case is impossible, and e_i' is vacuously true. Otherwise, the subgoal e_i' must be shown to follow from the axioms supplemented by the case hypothesis. LP assigns names of the form *prefixCaseHyp.number* to the hypotheses in proofs by cases.

Proofs by cases are often used to simplify implications, terms involving *if*, and terms involving repeated boolean subexpressions. For example, given the axiom $if(divides(x, 2), even(x), odd(x))$, the command

$prove\ even(x) \mid odd(x)\ by\ case\ divides(x, 2)$

first adds $divides(xc, 2) \to true$ (where xc is a new constant) as a case hypothesis to be used in proving the subgoal $even(xc) \mid odd(xc)$. The command **critical-pairs** *∗CaseHyp* **with** *user* can then be used to cause LP to compute the critical-pair equation $if(true, even(xc), odd(xc)) == true$ between the case hypothesis and the axiom. LP normalizes this equation (using the built-in rewrite rules for if) and orients it into the rewrite rule $even(xc) \to true$, which LP then uses to normalize the subgoal in the first case to $true$. LP next adds $divides(xc, 2) \to false$ as a case hypothesis and attempts to prove the same subgoal. This subgoal can also be established by computing critical pairs, thereby completing the proof of the conjecture.

When a case hypothesis contains new constants, it is often useful to compute critical-pair equations between the hypothesis and other rewrite rules.

6.3 Proofs by induction

Proofs by induction are based on the induction rules described in Section 4.6.

The command **prove** e **by induction on** x **using** IR directs LP to prove the equation e by induction on the variable x using the induction rule named IR. The names of the variable and/or the induction rule can be omitted if they can be inferred (e.g., because induction is possible on only one variable in e and there is only one induction rule for the sort of that variable).

LP generates subgoals for the basis and induction steps in a proof by induction, as follows. The basis subgoals involve proving the equations that result from substituting the basis generators of IR for x in e. (Basis generators are those with no arguments of the sort of x; fresh variables are used for arguments of other sorts, as in $singleton(e)$.) LP introduces additional hypotheses for the induction subgoals by substituting one or more new constants for x in e. (As discussed in Section 3.4, these constants have names like *xc*, *xc1*, ...) Each induction subgoal involves proving an equation that results from substituting a nonbasis generator of IR (applied to these constants) for x in e (e.g., $insert(e, xc)$ or $xc \cup xc1$). As in proofs by cases (see Section 6.2), LP substitutes new constants for variables when it generates hypotheses to be used in proving a subgoal, and it assigns names of the form *prefixInductHyp.number* to induction hypotheses.

Figures 11 and 12 show the output produced by LP as it initiates proofs by induction using the axiomatizations in Figures 1 and 3. As in a proof by cases, it is often useful to compute critical-pair equations between induction hypotheses and other rewrite rules.

```
LP101: prove i < j ⇒ i < (j + k) by induction on j
Conjecture lemma.1: Subgoals for proof by induction on ‘j’
Basis subgoal:
    lemma.1.1: (i < 0) ⇒ (i < (0 + k)) == true
Induction constant: jc
Induction hypothesis:
    lemmaInductHyp.1: (i < jc) ⇒ (i < (jc + k)) == true
Induction subgoal:
    lemma.1.2: (i < s(jc)) ⇒ (i < (s(jc) + k)) == true
```

Figure 11: Subgoals for a proof by induction over the sort *Nat*

```
LP101: set name setInduction2
LP102: assert Set generated by empty, singleton, ∪
LP103: set name lemma
LP104: prove x ⊆ x by induction using setInduction2
Conjecture lemma.1: Subgoals for proof by induction on ‘x’
Basis subgoals:
    lemma.1.1: empty ⊆ empty == true
    lemma.1.2: singleton(e) ⊆ singleton(e) == true
Induction constants: xc, xc1
Induction hypotheses:
    lemmaInductHyp.3: xc ⊆ xc == true
    lemmaInductHyp.4: xc1 ⊆ xc1 == true
Induction subgoal:
    lemma.1.3: (xc1 ∪ xc) ⊆ (xc1 ∪ xc) == true
```

Figure 12: Subgoals for a proof by induction over the sort *Set*

LP also allows multilevel induction. Such inductions are useful, for example, when proving facts about the Fibonacci numbers. The command

prove e **by induction on** x **depth** n **using** IR

directs LP to prove e by n-level induction using the induction rule IR. If IR is the induction rule for *Nat* in Figure 1, then LP would attempt to prove an equation e by 2-level induction by proving the subgoals $e(0)$ and $e(s(0))$ in the basis step of the induction, and then by proving the subgoal $e(s(s(c)))$ in the induction step using $e(c)$ and $e(s(c))$ as induction hypotheses.

6.4 Proofs by contradiction

Proofs by contradiction provide an indirect method of proof. If an inconsistency follows from adding the negation of a conjecture to a logical system, then the conjecture is a theorem of that system.

When LP attempts to prove an equation $t_1 == t_2$ by contradiction, it first generates a hypothesis $not(t_1' = t_2')$ by substituting new constants for the variables in $not(t_1 = t_2)$. (This hypothesis is logically equivalent to the negation of the conjecture because introducing the new constants is equivalent to replacing the universally quantified variables in the conjecture by existentially quantified ones. See Section 6.2.) LP assigns a name of the form *prefixContraHyp.number* to this hypothesis. It then generates the single subgoal $true == false$.

Figure 13 shows the output produced by LP as it initiates a proof by contradiction using the axiomatization in Figure 1. The proof can be finished by computing critical pairs between *lemmaContraHyp* and other rules. One way to do this is for the user to type **complete**.

```
LP101: prove not(0 = s(i)) by contradiction
Conjecture lemma.1: Subgoal for proof by contradiction
New constant: ic
Hypothesis:
    lemmaContraHyp.1: not(0 = s(ic)) == false
Subgoal:
    lemma.2.1: true == false
```

Figure 13: Subgoals for proof by contradiction

6.5 Proofs of implications

Proofs of implications can be carried out using a simplified proof by cases. The command **prove** $t_1 \Rightarrow t_2$ **by** $\Rightarrow$ directs LP to prove the subgoal t_2' using the hypothesis $t_1' == true$, where t_1' and t_2' are obtained as in a proof by cases. (This suffices because the implication is vacuously true when t_1' is false.) LP assigns a name of the form *prefixImpliesHyp.number* to the hypothesis in such a proof.

For example, given the axioms $a \Rightarrow b \rightarrow true$ and $b \Rightarrow c \rightarrow true$, the command **prove** $a \Rightarrow c$ **by** $\Rightarrow$ uses the hypothesis $a \rightarrow true$ to normalize the axioms and to reduce the conjecture to an identity.

The command **resume by** $\Rightarrow$ directs LP to resume the proof of the current conjecture using the **by** $\Rightarrow$ proof method; this command is applicable only when the current conjecture has been reduced to an implication.

Users should beware of using the **by** $\Rightarrow$ proof method prematurely. When using this method in a proof of $t_1 \Rightarrow t_2$, LP replaces all variables in t_2 that also occur in t_1 by fresh constants, thereby making it impossible for the user to continue the proof by induction on those variables.[22]

6.6 Proofs of conditionals

Proofs of equations involving the conditional operator *if* can also be carried out using a simplified proof by cases. The command **prove** $if(t_1, t_2, t_3) == t_4$ **by if-method** directs LP to prove an equation by division into two cases, t_1 and $not(t_1)$. As in a proof by cases, LP substitutes new constants for the variables of t_1 in all terms t_i to obtain terms t_i'. In the first case, LP assumes $t_1' \rightarrow true$ as an additional hypothesis and attempts to prove the subgoal $t_2' == t_4'$. In the second case, it assumes $t_1' \rightarrow false$ as an additional hypothesis and attempts to prove $t_3' == t_4'$. LP assigns names of the form *prefixIfHyp.number* to the hypotheses in such a proof.

The command **resume by if-method** directs LP to resume the proof of the current conjecture using the **if-method**; this command is applicable only when the current conjecture has been reduced to an equation of the form $if(t_1, t_2, t_3) == t_4$, where t_4 does not begin with *if*.

6.7 Proofs of conjunctions

Proofs of conjunctions can be slow because & is associative and commutative, and **ac**-matching is inherently slow. The command **prove** t_1 & ... & t_n **by** & provides a way to reduce this expense by directing LP to prove each of the conjuncts $t_1, \ldots, t_n$ as a separate subgoal.

The command **resume by** & directs LP to resume the proof of the current conjecture using the **by** & proof method; this command is applicable only when the current conjecture has been reduced to a conjunction.

Users should beware that employing this method too early in a proof can result in an increased need for user interaction and even in increased computation later in the proof, for example, when the same lemma is needed to prove more than one conjunct.

6.8 Proofs by explicit commands

The special proof-method **explicit-commands** directs LP not to apply any method of backward inference automatically to a conjecture, but to wait for an explicit method to be given with a subsequent **resume** command. This is used most frequently to prevent LP from attempting to normalize a conjecture when it would be time-consuming and unfruitful to do so. For example, if a conjecture includes many conjuncts, it may be appropriate to first compute some critical pairs, then apply the & method, and finally normalize the individual subgoals.

6.9 Default proof methods

LP allows users to determine which methods of backward inference for proving equations are applied automatically and in what order. The LP command

[22] Future versions of LP will provide mechanisms that circumvent this problem.

set proof-methods m_1, ..., m_n

directs LP to use the first of the methods m_1, ..., m_n that applies to the current conjecture. LP does this repeatedly (as it introduces new subgoals) until none of the methods in the list applies to the current conjecture. The command

set proof-methods explicit-commands

prevents LP from applying any method automatically. The default proof method list is **normalization** alone.

6.10 Proofs of deduction rules

LP permits users to prove deduction rules as well as to assert them. For example, the command

prove when $(forall\ z)\ z \subseteq x == z \subseteq y$ **yield** $x == y$

directs LP to initiate a proof of a deduction rule about set inclusion. In response to this command, LP introduces new constants xc and yc of sort *Set*, adds $z \subseteq xc == z \subseteq yc$ as a hypothesis to its logical system (with a name of the form *prefixWhenHyp.number*) and attempts to prove $xc == yc$ as a subgoal. User guidance is required to finish this proof, for example, by entering the **complete** command, which causes LP to draw the necessary inferences from the additional hypothesis.

In general, LP responds to the command

prove when $(forall\ x_1, \ldots, x_m)\ h_1, \ldots, h_n$ **yield** $c_1, \ldots, c_k$

by replacing all variables other than $x_1, \ldots, x_m$ in the hypotheses $h_1, \ldots, h_n$ of the deduction rule by new constants (see Sections 4.1 and 6.2) to form hypotheses $h'_1, \ldots, h'_n$ and subgoals $c'_1, \ldots, c'_k$ for the proof of the deduction rule. When there is no **forall** clause in the deduction rule, LP replaces all variables in the hypotheses by new constants.

6.11 Proofs of induction rules

LP also permits users to prove induction rules. For example, the command

prove *Set generated by empty*, *singleton*, ∪

directs LP to initiate a proof of the induction rule *setInduction2* displayed in Section 4.6. In response to this command, LP introduces a new operator *isGenerated* with signature $Set \rightarrow Bool$, adds the hypotheses[23]

$isGenerated(empty)$
$isGenerated(singleton(e))$
$(isGenerated(s1)\ \&\ isGenerated(s)) \Rightarrow isGenerated(s1 \cup s)$

to its logical system (with names of the form *prefixGenHyp.number*), and attempts to prove the subgoal $isGenerated(s)$. User guidance is required to finish this proof, for example, by first proving the lemma $insert(e, s) == s \cup singleton(e)$ (by instantiating the deduction rule corresponding to the assertion *Set* ***partitioned by*** ∈ and then issuing the commands **resume by induction** and **complete**).

[23]The hypotheses introduced by this proof method provide an adequate definitional semantics for $isGenerated$, but a rather weak operational semantics. Future versions of LP may introduce additional hypotheses with stronger operational semantics.

7 Features of LP

This section presents a summary of the commands available in LP, together with information about commands not described earlier in this guide. Details concerning all commands are available from LP's online help facility. The following notation is used to describe the command syntax.

Notation	Meaning
e	a keyword
{*e*}	*e* as a syntactic unit
$e \mid e'$	either *e* or e'
[*e*]	an optional *e*
e^*	zero or more *e*'s
$e^*,$	zero or more *e*'s, separated by commas
$e^*_{[,]}$	zero or more *e*'s, optionally separated by commas
e^+	one or more *e*'s
$e^+,$	one or more *e*'s, separated by commas
$e^+_{[,]}$	one or more *e*'s, optionally separated by commas
'*c*'	the character(s) *c*

Many commands take named sets of objects as arguments. Figure 14 describes the syntax for *names* in LP. The keywords for *statement-types* can be abbreviated using unambiguous prefixes (e.g., **deduction-rules** can be abbreviated to **d-r** or to **deduct**). As described in Section 3.4, asterisks in *name-patterns* match

names	::=	*statement-type*$^*_{[,]}$ *name-pattern*$^*_{[,]}$
statement-type	::=	**deduction-rules** | **equations** | **induction-rules** | **operator-theories** | **rewrite-rules**
name-pattern	::=	*name-character*$^+$ [*extension*] ['`!`']
name-character	::=	*identifier-character* | '`*`'
identifier-character	::=	*letter* | *digit* | '_' | '′'
extension	::=	'`.`' *number* ['`.`' *number*]* [*extension-range*]
extension-range	::=	'`:`' { *number* | **last** }
number	::=	*digit*$^+$

Figure 14: Syntax for *names* of sets of objects

any sequence of *identifier-characters*, and, when a *name-pattern* does not end with an exclamation mark, any extension of a matched name is also matched. With these conventions,

display d r ***arith.1:2*** $*$***Hyp***

causes LP to display all deduction and rewrite rules with names *arith.1* or *arith.2*, with subnames of these names, or with name prefixes that end in Hyp.

7.1 Commands for user interaction

The commands in the following table are described in Sections 3.2, 3.3, and 3.4, as well as here in Section 7.1.

User Interaction	
clear	Discard all information other than settings
delete *names*	Delete named facts
display [*display-info*] [*names*]	Display information about named objects
execute ***filename***	Execute commands from *filename.lp*
execute-silently ***filename***	Same as **execute**, but suppressing all output
forget [**pairs**]	Discard information to save space
freeze ***filename***	Save state of LP in *filename.lpfrz*
help ***topic***$^{*}_{[,\ ]}$	Print help about topics
history [*number* |**all**]	Print recent command history
quit	Exit from LP
pop-settings	Restore values of LP settings
push-settings	Remember values of LP settings
set	Print current values of all LP settings
set ***setting***	Print current value of setting, allow change
set ***setting-and-value***	Change value of setting
show normal-form ***term***	Display reduction of term to normal form
show unifiers ***term, term***	Display unifiers of two terms
statistics [*stat-options*]	Display statistics on runtime, storage, rule usage
stop	Stop execution of command files
thaw ***filename***	Restore frozen state from *filename.lpfrz*
unset {*setting* | **all**}	Reset setting to its default value
version	Display information about current version of LP
write ***filename*** [*names*]	Write declarations, registry, named facts to *filename.lp*
% ***comment***	Record *comment* in log and/or script file

7.1.1 Saving and restoring state

The command **freeze** *filename* causes LP to create a file named *filename.lpfrz* and to save its current state (except for the statistics and file-system dependent settings (**log-file**, **script-file**, **directory**, and **lp-path**) in that file. This command is useful for checkpointing attempted proofs. The command **thaw** *filename* causes LP to restore its state to that frozen in the file *filename.lpfrz*.

The command **write** *filename* causes LP to create an ASCII file named *filename.lp*, which can be executed to recreate the current system. LP writes declarations for all identifiers followed by commands to recreate the current registry and to assert the current facts. Rewrite rules that are written by the **write** command will be read as equations. Unlike **freeze**, **write** does not save information about the state of any uncompleted proofs. But unlike thawing a frozen file, which replaces all of LP's logical system, executing a written file adds information to the current system. Hence it can be used to combine axiomatizations.

The commands **push-settings** and **pop-settings** use a stack to save and restore the values of all settings (see Section 7.5) other than log and script files. The **write** command, for example, places these commands in *.lp* files so that named axioms can be loaded from files without affecting the current **name**, **activity**, and **immunity** settings.

7.1.2 Displaying information

The command **display** [*display-info*] [*names*] displays the requested information about the named objects. If no *names* are specified, it displays the requested information for the entire logical system.

display-info	Causes display of
conjectures	the named conjectures
facts	the named facts (the default)
ordering-constraints	the registry for operators in the named facts
proof-status	status of proofs involving the current conjecture
symbols	all identifiers in the named facts

The **display** command annotates inactive facts by printing the letter P in parentheses following their names. It annotates immune and ancestor-immune facts by printing the letters I and i in parentheses.

The command **history** [n], where n is a positive integer, causes LP to print a list of the n most recent commands executed by LP. The command **history all** causes LP to print a list of all commands. The command **history** with no arguments is treated by LP in the same fashion as the last **history** command (or as **history all** if there were no previous **history** commands). Histories differ from scripts created by the **set script** command in that, after a **thaw**, the history shows the commands that produced the thawed *.lpfrz* file, whereas the script shows the commands that have been executed since starting the script.

The **show** command can be used to provide information about rewriting. The command **show normal-form** *term* displays the normal form of *term*. When the **trace-level** setting is nonzero, it also displays the reduction sequence leading from the term to its normal form. The command **show unifiers** *term, term* displays a complete set of minimal unifiers for the two terms.

The command **statistics** [**time** | **usage** [*names*]] displays cumulative and recent (since the last display) information about the resources consumed by LP, as well as about the usage of rewrite and deduction rules. The default option is **time**, which displays information about the time spent by LP on various activities. The **usage** option displays how many times LP successfully applied each of the named rewrite and deduction rules, as well as how many non-identity critical pairs were computed from each rewrite rule. Setting **statistics-level** to 0 (the default is 2) suppresses the collection of all but summary statistics. Setting it to 1 suppresses the collection of usage statistics. Setting it to 3 causes LP to report the number of attempts made to apply each rewrite rule (but causes LP to run more slowly).

The **version** command displays information about how LP was installed (see Section 3.6).

7.1.3 Deleting information and saving space

The **clear** command causes LP to discard all information other than the current settings.

The command **delete** *names* causes LP to delete the named facts from its logical system. It can be used to get rid of unhelpful facts (e.g., unorderable or unnecessary critical-pair equations) or facts that have served their purpose and are no longer needed.

The **forget pairs** command causes LP to discard all information about which critical pairs have been computed. It also prevents LP from accumulating further such information until the next **complete** command is given. This command can save significant space when there are many rewrite rules. The commands **forget** and **forget pairs** are equivalent in Release 2.2 of LP.

7.2 Commands for axiomatizing theories

The following table summarizes LP's features for defining theories. These features are described in Sections 4–6.

Axioms and Facts	
assert ***equation***$^{+}_{[,]}$	Assert equations as axioms
assert ***deduction-rule***$^{+}_{[,]}$	Assert deduction rules as axioms
assert ***sort*** **generated by** ***operator***$^{+}_{,}$	Assert induction rule as an axiom
assert ***sort*** **partitioned by** ***operator***$^{+}_{,}$	Assert deduction rule as an axiom (e.g., set extensionality)
assert ac ***operator***	Assert associative-commutative axioms
assert commutative ***operator***	Assert commutative axiom
declare operators ***op-declaration***$^{+}_{[,]}$	Declare operators
declare sorts ***sort***$^{+}_{,}$	Declare sorts
declare variables ***var-declaration***$^{+}_{[,]}$	Declare variables
make ***fact-status names***	Change activity, immunity of facts and conjectures, where *fact-status* is one of **active**, **inactive**, **passive**, **immune**, **nonimmune**, or **ancestor-immune**

The following two tables specify the syntax for declaring and using variables and operators in LP. Identifiers can be overloaded provided that identifiers for the built-in logical operators are not overloaded by the user, and that the same identifier is not overloaded as a variable and a constant of the same sort.

Syntax for Variable Declarations		
var-declaration	::=	*variable-identifier*$^{+}_{,}$ ‘`:`’ *sort*
variable-identifier	::=	*identifier*
identifier	::=	*identifier-character*$^{+}$
identifier-character	::=	*letter* | *digit* | ‘_’ | ‘′’
variable	::=	*variable-identifier* [‘`:`’ *sort*]

Syntax for Operator Declarations		
op-declaration	::=	*operator-identifier*$^{+}_{,}$ ‘`:`’ *signature*
operator-identifier	::=	*prefix-identifier* | *infix-identifier*
prefix-identifier	::=	*identifier*
infix-identifier	::=	*infix-character*$^{+}$ | ‘`\`’ *identifier*
infix-character	::=	‘`!`’ | ‘`#`’ | ‘`$`’ | ‘`&`’ | ‘`*`’ | ‘`+`’ | ‘`-`’ | ‘`.`’ | ‘`/`’ | ‘`<`’ | ‘`=`’ | ‘`>`’ | ‘`@`’ | ‘`\`’ | ‘`^`’ | ‘`|`’ | ‘`~`’
signature	::=	*domain* ‘`->`’ *range*
domain	::=	*sort*$^{*}_{,}$
range	::=	*sort*
sort	::=	*identifier*
operator	::=	*operator-identifier* [‘`:`’ *signature*]

The following two tables specify the syntax for equations and deduction rules in LP. The sorts of subterms in a term, and of terms in an equation, must conform to the existing declarations for identifiers.

Syntax for Equations and Terms

equation	::=	*term* [*equals term*]
equals	::=	‘==’ |‘->’
term	::=	*equals-term* {*logical-infix-identifier equals-term*}*
equals-term	::=	*userOp-term* [‘=’ *userOp-term*]
userOp-term	::=	*subterm* {*infix-identifier subterm*}*
subterm	::=	*atomic-term* [‘:’ *sort*]
atomic-term	::=	*variable-identifier*
		| *prefix-identifier* [‘(’ *term*$^{+}_{,}$ ‘)’]
		| ‘(’ *term* ‘)’
logical-infix-identifier	::=	‘&’ | ‘|’ | ‘=>’ | ‘<=>’

Syntax for Deduction Rules

deduction-rule	::=	**when** [*quantifier*] *equation*$^{+}_{[,]}$ **yield** *equation*$^{+}_{[,]}$
quantifier	::=	**forall** *variable*$^{+}_{,}$
		| ‘(’ **forall** *variable*$^{+}_{,}$ ‘)’

7.3 Commands for proving theorems

The following table summarizes LP's inference mechanisms, as described in Sections 5–6.

Proofs and Inference Mechanisms	
apply *names to names*	Apply named deduction rules to named facts
cancel [**all**]	Cancel current conjecture [all conjectures]
complete	Run completion procedure
critical-pairs *names* ***with names***	Compute critical-pair equations between any rewrite rules in the first named set and any in the second
instantiate {*variable* **by *term***}$^+$, **in *names***	Instantiate variables by terms in named facts
normalize *names* [**with *names***]	Normalize named facts, immune or not, by all (or named) rewrite rules, active or not
normalize conjecture [**using *names***]	Normalize current conjecture by all (or named) rewrite rules, active or not
prove *conjecture* [**by *proof-method***]	Attempt to prove conjecture (using *proof-method*)
qed	Check that all conjectures have been proved
resume [**by *proof-method***]	Resume work on current conjecture (using *proof-method*)
rewrite *names* [**with *names***]	Rewrite each named fact, immune or not, by some (named) rewrite rule, active or not
rewrite conjecture [**using *names***]	Rewrite the current conjecture by some (named) rewrite rule, active or not
<> *number*	Confirm introduction of *number* subgoals in proof
[ ]	Confirm conclusion of step in proof

7.3.1 Proof methods

As discussed in Section 6, *conjectures* can be equations, deduction rules, or operator theories. LP recognizes the following *proof-methods* for backward inferences involving equational conjectures.

*Proof Methods for the **prove** and **resume** Commands*	
&-method	Applicable to t_1 & ... & t_n == $true$
⇒-method	Applicable to $t_1 \Rightarrow t_2 == true$
cases $t_1, \ldots, t_n$	Divides proof into cases $t_1, \ldots, t_n$
contradiction	Initiates proof by contradiction
default	Invokes method given by **proof-methods** setting
explicit-commands	Suspends proof pending explicit **resume** command
if-method	Applicable to $if(t_1, t_2, t_3) == t_4$
induction [[**on**] *variable*] [**depth** ***number***] [[**using**] *names*]	Initiates proof by induction
normalization	Initiates proof by reduction to normal form

7.3.2 Box checking

As discussed in Sections 2 and 6, LP generates <>'s and [ ]'s in the history and in the script file. It generates a line beginning with <> *number* whenever it creates *number* subgoals in a proof. It generates a line beginning with [ ] whenever it finishes the proof of a subgoal or a conjecture. After a successful proof, the number of [ ]'s in the history and script file equals the number of **prove** commands plus the number of subgoals that were created by LP.

LP ignores <> and [ ] commands when it is not executing a command file or when the **box-checking** setting is **off**. Otherwise, it checks these commands for errors, as follows. Whenever it generates <> *number* or [ ], LP checks that the next nonblank line in the command file begins with <> *number* or [ ]. The prompts <>? and [ ]? indicate that LP expects a confirming <> or [ ] in the command file. LP prints an error message if the confirming <> or [ ] is missing, or if an unexpected <> or [ ] appears in the command file.

Regardless of whether **box-checking** is **on** or **off**, LP does not copy <> and [ ] lines from its input to the history or to a script file. Instead, it puts into the history and script file the <> and [ ] lines that it produces as it creates and discharges goals. Thus, the history and the script file will be annotated in a way that correctly reflects the actual progress of the proof. For this reason, it is often useful to copy fragments of script files back into command files.

7.4 Commands for ordering equations into rewrite rules

Ordering Commands	
order [*names*]	Orient equations into rewrite rules
register ***constraints***	Constrain orientation of equations
unorder [*names*]	Turn rewrite rules back into equations
unregister registry	Discard all ordering constraints
unregister {**bottom** | **top**} *operator*$^{+}_{[,]}$	Discard indicated constraints

As discussed in Section 4.9, LP automatically orients equations into rewrite rules when the **automatic-**

ordering setting is **on**. When it is **off**, users must type explicit **order** commands to orient equations into rewrite rules. If no *names* are given with the **order** command, LP attempts to orient all equations into rewrite rules. If *names* are specified, LP attempts to orient only the named equations (including any new equations that LP generates during the ordering process, for example, as a result of applying a deduction rule to a newly reduced fact).

LP uses the method specified by the **ordering** setting to orient equations into rewrite rules. The **dsmpos** and **noeq-dsmpos** orderings use *constraints* provided by the **register** command, or suggested by LP, to help orient equations. LP adds suggested constraints to the registry automatically when the **automatic-registry** setting is **on** and upon user confirmation when it is **off**. The **polynomial** ordering also uses *constraints* supplied by the **register** command. The following table describes the syntax for specifying ordering *constraints*.

Ordering Constraints		
constraints	::=	{ **bottom** | **top** } *operator*$^+_{[,]}$
		| **height** *operator-set* { *height operator-set* }$^+$
		| **polynomials** ***operator polynomial***$^+_{[,]}$
		| **status** ***status operator***$^+_{[,]}$
operator-set	::=	*operator* | ‘`(`’ *operator*$^+_{[,]}$ ‘`)`’
height	::=	‘>’ | ‘=’ | ‘>=’
polynomial	::=	*polynomial-term* {‘`+`’ *polynomial-term*}*
polynomial-term	::=	*polynomial-factor* {‘`*`’ *polynomial-factor*}*
polynomial-factor	::=	*polynomial-primary* [‘`^`’ *number*]
polynomial-primary	::=	*variable* | *number* | ‘`(`’ *polynomial* ‘`)`’
status	::=	**left-to-right** | **multiset** | **right-to-left**

7.5 Settings

The following three tables present a summary of the settings that govern the behavior of LP. Details concerning the settings are located in the indicated sections of this guide and are also available from LP’s online help facility.

Settings, Part I

activity ***on-off***	Initial activity for axioms	
	Default: **on**	[5.6]
automatic-ordering *on-off*	Automatic ordering of new equations	
	Default: **on**	[4.9, 7.4]
automatic-registry ***on-off***	Automatic extensions to registry	
	Default: **on**	[4.9.4, 7.4]
box-checking ***on-off***	Checking <>, [ ] annotations of proofs in script files	
	Default: **off**	[7.3.2]
completion-mode ***mode***	Completion mode: **big**, **expert**, **standard**	
	Default: **standard**	[7.5.2]
directory ***string***	Name of directory for output files	
	Default: '.'	[3.6]
display-mode	Mode for displaying identifiers	
qualify-mode	Default: **unqualified**	[7.5.1]
immunity ***immunity***	Initial immunity for axioms	
	Default: **off**	[5.6]
log-file ***filename***	File *filename.lplog* for logging session	
	Default: none	[3.5]
lp-path ***string***	Search path for help, *.lp*, *.lpfrz* files	
	Default: `'.  ~ ~lp/axioms ~lp'`	[3.6]
name-prefix ***identifier***	Name prefix for facts, conjectures	
	Default: *user*	[3.4]
ordering-method	Method for orienting equations into rewrite rules	
ordering	Default: **dsmpos**	[4.9]
page-mode ***on-off***	Page mode for output	
	Default: **off**	[3.6]
prompt ***string***	Prompt for commands	
	Default: `'LP!: '`	[3.6]
proof-methods	Automatic proof methods for equational conjectures	
proof-method$^{+}_{[,\ ]}$	Default: **normalization**	[6.9, 7.3.1]
reduction-strategy ***mode***	Reduction order for terms: **inside-out**, **outside-in**	
	Default: **outside-in**	[7.5.2]
rewriting-limit ***number***	Bound on rewrites per normalization	
	Default: 1000	[7.5.2]
script-file ***filename***	File *filename.lpscr* for recording input	
	Default: none	[3.5]

Settings, Part II

statistics-level ***number***	Kinds of statistics kept	
	Default: 2	[7.1.2]
trace-level ***number***	Kinds of details printed	
	Default: 1	[3.6]
write-mode ***qualify-mode***	Mode for writing identifiers	
	Default: **qualified**	[7.5.1]

Values for Settings

filename	::=	*string*
immunity	::=	*on-off* | **ancestor**
on-off	::=	**on** | **off**
ordering	::=	**dsmpos** | **either-way** | **left-to-right** | **manual**
		| **noeq-dsmpos** | **polynomial** [*number*]
qualify-mode	::=	**qualified** | **unqualified** | **overloaded** | **unambiguous**
string	::=	*character*$^{+}$ | '`'`' *character** '`'`'

7.5.1 Settings that affect output

The **display-mode** and **write-mode** settings control the manner in which the **display** and **write** commands print qualifications in terms. These commands also qualify operators appearing outside of terms (e.g., in induction rules) if the mode is **qualified** or if the operator is overloaded.

Mode	Effect on terms
qualified	Qualify all subterms
unqualified	Qualify nothing
overloaded	Qualify subterms headed by overloaded identifiers
unambiguous	Qualify enough to enable reparsing

The default **display-mode** is **unqualified**, which takes less time and produces output that is easier to read.

The default **write-mode** is **qualified**, which guarantees that the output can be reparsed even in the presence of additional overloadings for identifiers. It is often desirable, however, to set the **write-mode** to **unambiguous** to shorten and improve the readability of *.lp* files. If a problem arises in executing a *.lp* file produced in this fashion (because it is being executed in a context that overloads one of its operators), the problem can be solved by starting up a new copy of LP, executing the *.lp* file, and writing it out again in **qualified** mode.

7.5.2 Settings that affect rewriting

The command **set rewriting-limit** ***number*** sets an upper bound on the number of reductions that LP will perform when normalizing a term with respect to a rewriting system that is not guaranteed to terminate. The default value of *number* is 1000.

If LP exceeds the rewriting limit when normalizing a fact, it prints a warning message and immunizes that fact. If it exceeds the rewriting limit when normalizing a conjecture, the user can continue normalizing the conjecture by typing **resume** (after raising the rewriting limit, if desired).

The **set reduction-strategy** command controls the strategy used by LP to reduce terms. The default is **outside-in**, which causes LP to apply rewrite rules near the top of a term before it applies them near the bottom. In **inside-out** mode, LP still applies the built-in rewrite rules near the top of the term, but it applies other rewrite rules near the bottom before it applies them near the top.

The **set completion-mode** command controls the order in which completion tasks are executed and how much user interaction occurs. The default **standard** mode requires little user interaction, even when **automatic-registry** is **off**. However, it accomplishes this at the cost of computing critical pairs before extending the registry, which can be inefficient. For many completions, **expert** mode is better. It causes LP to extend the registry, and thereby to orient more equations, before it begins to compute critical pairs. When **automatic-registry** is **off**, it gives users more explicit control over the completion process. The **big** mode postpones the computation of critical pairs even farther, so that big equations are examined before critical pairs are computed.

8 Hints on using LP

This section contains a collection of hints that beginning users of LP may find helpful.

8.1 Preparing input and recording work

Start by using an editor to prepare a command file. Put all the declarations you expect to need at the beginning of the file. This allows LP to check your declarations before beginning any time consuming tasks. Put subproofs in separate command files. Structure the session as a sequence of **execute** commands. Freeze LP's state often. This makes it easier to try different alternatives when looking for a proof.

Although proofs are usually constructed interactively, successful proofs should be recorded in a cleaned-up command file. Always set scripting and logging on at the start of an LP session. (If you realize that you are not recording a session, start logging and then execute a **history all** command to get LP to print the commands already executed.) After executing a step of a proof, enter a comment recording information that may be helpful in cleaning up the LP-produced *.lpscr* file. If, for example, a **critical-pairs** command produced no useful critical pairs, record that fact in a comment.

Keep in mind that LP automatically indents and annotates *.lpscr* files. It is often useful to use an editor to replace parts of human-generated *.lp* files with material extracted from *.lpscr* files.

8.2 Formalizing axioms and conjectures

Be careful not to confuse variables and constants. If x is a variable and c is a constant, then $e(x)$ is a stronger assertion than $e(c)$. The first means $(\forall x)e(x)$. In the absence of other assertions involving c, the second means $(\exists c)e(c)$. If you don't know whether an identifier is a variable or a constant, type **display symbols** to find out.

Be careful about quantification. The expression $x = empty \Rightarrow x \subseteq y$ correctly (albeit awkwardly) captures the fact that the empty set is a subset of any set. However, its converse, $x \subseteq y \Rightarrow x = empty$, does not capture the fact that any set that is a subset of all sets is itself the empty set. That fact is expressed in first-order logic by the formula $(\forall x)[(\forall y)(x \subseteq y) \Rightarrow x = empty]$, which is equivalent to $(\forall x)(\exists y)[x \subseteq y \Rightarrow x = empty]$, and which can be expressed in LP by the deduction rule

when (**forall** y) $x \subseteq y$ **yield** $x == empty$

but not by any equation.

An axiom or conjecture of the form **when** A **yield** B has the same logical content as one of the form $A \Rightarrow B == true$, but different operational content. Consider Figure 15. LP will automatically derive the fact $g(a)$ from $f(a)$ by applying the deduction rule, but it will not derive $h(a)$ from $g(a)$ unless it is instructed to compute critical pairs.

A multiple-hypothesis deduction rule of the form **when** A, B **yield** C has the same logical content as a single-hypothesis rule of the form **when** A & B **yield** C. They differ operationally in that, if the user asserts or proves two equations that are matched by A and B, LP will apply the multiple-hypothesis rule but not the single-hypothesis rule.

```
declare variable x: Bool
declare operators
    a: → Bool
    f, g, h: Bool → Bool
    ..
assert when f(x) yield g(x)
assert
    g(x) ⇒ h(x)
    f(a)
    ..
qed
```

Figure 15: Deduction rules vs. implications

8.3 Ordering equations into rewrite rules

If you put some well-selected ordering constraints in the registry, LP will orient equations more quickly and with fewer surprises. Put the generators for a sort, such as 0 and s for *Nat*, at the bottom of the registry. Enter definitions, such as $P(x) == P1(x)\ \&\ P2(x)$, with $\rightarrow$ rather than $==$; otherwise they are likely to be reversed, because the right side appears more complex than the left side.

When a proof fails unexpectedly, look at the rewrite rules to see if any are ordered in surprising directions. If so, there are several potentially useful things to try.

- Set **automatic-registry off**, instruct LP to order only the offending equation, and choose one of the presented suggestions that order the equation as desired. Then add **register** commands corresponding to that suggestion to your command file and try running the proof again.
- Alternatively, rerun the proof at a trace level (e.g., 2) that prints out extensions to the registry; then use a text editor and the *.lplog* file to locate extensions dealing with operators appearing in the offending rewrite rule. This may suggest a set of **register** commands that will force the equations to be ordered as desired.
- Alternatively, rerun the proof with **automatic-registry** set **off** to find a set of suggestions that will order things the way you want them. Then add **register** commands with the appropriate suggestions to your command file, and execute it again with **automatic-registry** set **on**. This last step is important because proof scripts with **automatic-registry off** are not usually robust.

Occasionally, LP will fail to order a set of equations for which a terminating set of rewrite rules does indeed exist. At this point you should consider changing the ordering to use a more powerful ordering strategy (e.g., **dsmpos** rather than **noeq-dsmpos**) or an ordering strategy that makes no attempt to check termination (e.g., **left-to-right**). It is also worth keeping in mind that although LP will not automatically give operators equal height when using **noeq-dsmpos**, the **register** command can be used to do so explicitly.

8.4 Managing proofs

Prove as you would program. Design your proofs. Modularize them. Think about their computational complexity.

Be careful not to let variables disappear too quickly in a proof. Once they are gone, you cannot do a proof by induction. Start your inductions before starting proofs by cases, $\Rightarrow$, or **if**.

Splitting a conjecture into separate conjuncts (using the & proof method) early in a proof often leads to repeating work on multiple conjuncts, for example, doing the same case analysis several times.

To keep lemmas and theorems from disappearing (because they normalize to identities), make them immune. Typing the commands

set immunity on
prove . . . **by explicit-commands**
set immunity off
resume by . . .

when you begin the proof of a conjecture immunizes that conjecture (i.e., causes it to be immune once it becomes a theorem), but nothing else. Similarly, the commands

set immunity ancestor
instantiate . . . **in** . . .
set immunity off

help keep instantiations from disappearing when they are special cases of other facts.

When a proof gets stuck:

- Be skeptical. Don't be too sure your conjecture is a theorem.
- If the conjecture is a conditional, conjunction, or implication, try the corresponding proof method.
- Try computing critical pairs between hypotheses and other rewrite rules, for example, by typing **critical-pairs** $*$*Hyp* **with** $*$.
- Use a proof by cases to find out what is going on. Case on repeated subterms of the conjecture, on the antecedent of an implication in a rewrite rule, or on the test in an *if* in a rewrite rule.
- Display the hypotheses and check to see if any that you expected to see are missing or are not ordered in the way you expected.
- Look for a useful lemma to prove. See if replacing a repeated subterm in a subgoal by a variable results in a more general fact that you know to be true.
- Because LP automatically internormalizes facts, you may find that what you consider to be the information content of some user-supplied assertion has been "spread out" over several facts in the current logical system in a way that may not be easy to understand, particularly if the system contains dozens or hundreds of facts. Similarly, you may sometimes notice that LP is reducing (or has reduced) some expression in some way that you don't understand. The command **show normal-form** E, where E is the expression being mysteriously reduced, or where E is the original form of one side of an equation, will often be enlightening in such cases. Setting the trace level up to 6 will show which rewrite rules are applied in the normalization.

In the course of a proof, you may lose track of your place in the subgoal tree. This happens most often if LP has just discharged several subgoals in succession without user intervention and/or it has automatically introduced subgoals. The **display**, **resume**, and **history** commands can be used to help find your place.

- **display** $*$*Hyp* is an easy way to find your place in nested case analyses.
- **display proof-status** displays the entire proof stack; **display conjectures** ***names***, the named conjectures.
- **resume** shows just the current conjecture (normalized, if the **proof-methods** setting includes **normalization**).
- **history 20** (or some other number) displays an indented history, including LP-generated box and diamond lines.

8.5 Making proofs go faster

When LP seems too slow, use the **statistics** command to find out which activities are consuming a lot of time. If rewriting (particularly, unsuccessful rewriting) is costly, try one of the following.

- Immunize facts that you know to be irreducible. LP will not waste time trying to reduce them.
- Deactivate rewrite rules that are needed only occasionally.
- Make definitions passive and apply them manually.
- Avoid big terms, especially with **ac** operators. Seek different axiomatizations or proof strategies if they occur.

If ordering is costly, put ordering constraints in the registry, particularly if you have declared many operators. It may also help to put ordering constraints in the registry prior to a proof by cases to save the cost of having LP rediscover these constraints in each of the cases.

If unification or critical pairing is costly, try to use smaller rule lists as arguments to the **critical-pair** commands. Also, try to avoid computing critical pairs between rewrite rules that contain subterms such as t_1 & t_2 & $\ldots$ & t_5 with multiple occurrences of the same **ac** operator.

8.6 Overcoming installation problems

LP ordinarily expects the file *lp.help*, which contains messages for the **help** command, to reside in the directory */usr/local/lib/lp*. If it resides in some other directory named *dir*, invoke LP using the command line *lp* $-$*d dir* (or make *lp* an alias for *lp* $-$*d dir*).

8.7 Reporting bugs

There may still be a bug (or maybe even two) in LP. Please report any bugs that you find (preferably by e-mail). When reporting a bug, always include a sample command file that will provoke it and a *.lplog* file that illustrates it. To produce the *.lplog* file:

1. Follow your procedure for producing the bug until shortly before the commands that trigger the bug.
2. Type the commands

   set log *bugreport*
   history
   display

3. Enter the remaining commands necessary to exhibit the bug. Include comments where appropriate. If the last command sends LP into an infinite loop or manages to crash LP without closing the log file, precede it with an **unset log** command so that *bugreport.lplog* will be closed.

9 Current development

LP 2.2 is written in CLU and runs under Unix. Native CLU compilers exist for DEC VAXes as well as the 68000-based Sun and HP workstations. A portable CLU compiler translates CLU into C and enables LP to run on DEC, MIPS, and Sun RISC architectures. LP is available by anonymous ftp from larch.lcs.mit.edu.

LP 1.0 [11] was in relatively heavy use at several sites for several years. During that time LP changed dramatically, primarily in response to the needs of its users. LP continues to change in response to user needs. Specialized front-ends are being developed, for example, to assist in checking Larch specifications and in proving the correctness of circuits. Features for proving and using more general first-order formulas are also being developed, as are extensions of the notions of critical pairs and completion to encompass deduction rules. Our primary concern is to preserve the basic style and efficiency of proofs in LP.

Although LP is much faster than its ancestor Reve [22], performance continues to be an issue. Each increase in speed tempts LP's users to try larger examples. Frequently, these examples suggest other desirable user amenities or further opportunities for improvements in performance. The sample proof in Appendix B took three seconds on a DECstation 5000/200; more ambitious proofs, such as the transparency of a cache memory subsystem, take scores of minutes. A major goal for LP is to reduce as much as possible the costs of developing, executing, and maintaining such ambitious proofs.

10 Acknowledgements

LP is descended from the Reve term-rewriting program [22], which was implemented by Pierre Lescanne, Randy Forgaard, Kathy Yelick, and Dave Detlefs.

LP's users have contributed many useful criticisms and suggestions. We are particularly appreciative of the feedback supplied by Ursula Martin, Niels Mellergaard, Andres Modet, Jørgen Staunstrup, Mark Vandevoorde, and Frederic Voisin. Christine Choppy and Michel Bidoit read an earlier version of this guide and made several useful suggestions.

We owe a special debt to Jim Horning and Jim Saxe. They have made so many useful suggestions that we often think of them as co-designers of LP (which isn't to say that they share blame with us for LP's failings). They also contributed an enormous number of helpful and useful comments about several versions of this guide.

Appendices

A Equational term-rewriting tutorial

Sections 4 and 5 introduced several key notions (e.g., normal forms and critical-pair equations) concerning equational term-rewriting. This appendix provides further details concerning both the theoretical basis for equational term-rewriting and also its use in theorem proving. For a more comprehensive and formal introduction to rewriting, see [7].

In general, for any set A of axioms and any assertion a, we write $A \models a$ (a is a *logical* or *semantic consequence* of A) to mean that a is true in all models of A. For example, if A is the set of axioms for groups in Figure 4, then a model of A is known as a group, and the logical consequences of A are those assertions that are true in all groups.

For some sets A of axioms, there are procedures that can be used to decide whether or not $A \models a$. By Church's Undecidability Theorem [5], which shows that the first-order theory of the natural numbers under addition and multiplication is undecidable, such procedures do not exist for all A. This appendix describes one approach to finding such procedures for some sets A of equations.

Appealing directly to the definition of $\models$ is of little help, since A in general has infinitely many models, and one cannot check whether a holds in all of them (or even in one of them, if that model happens to be infinite). Hence our plan of attack is to define a more tractable relation $A \vdash a$ (a is a *syntactic consequence* of A, or a is *provable* from A), and then to show that $\vdash$ is both *sound* with respect to $\models$ (i.e., that $A \models a$ whenever $A \vdash a$) and *complete* with respect to A (i.e., that $A \vdash a$ whenever $A \models a$).

Gödel's Completeness Theorem [13] shows that such a notion of provability exists for first-order logic. This theorem provides a semidecision procedure (albeit an inefficient one) for first-order logic, that is, an effective procedure for enumerating the logical consequences of A when A itself is effectively enumerable.

A.1 Equational theories

An *equational theory* is a set of equations that is axiomatized by a set of equations, that is, the equational theory of a set E of equations is the set of all equations e such that $E \models e$. Birkhoff [2] proved that the equational theory of E can be characterized syntactically in terms of the congruence relation defined by E over its free word algebra (see Section 4.3). We write $E \vdash_= t == u$ to mean that t is congruent to u in the congruence relation determined by E. Birkhoff showed that $\vdash_=$ is sound and complete with respect to $\models$ for equational theories, that is, that $E \vdash_= e$ if and only if $E \models e$.

Like Gödel's Theorem, Birkhoff's Theorem provides a semidecision procedure for equational theories: $E \models e$ iff e can be proved from E by a series of steps, each of which is justified by reflexivity, symmetry, transitivity, or substitutivity of equals for equals. Figure 5 showed a sample informal derivation, from the axioms for groups in Figure 4, of the fact that e is its own inverse.

Equational reasoning, such as used in Figure 5, does not provide a decision procedure, because the appropriate series of steps in a proof that $E \models e$ must be found and cannot, in general, be computed from E and e. Indeed, some equational theories are undecidable (see [7]).

A.2 Term-rewriting systems

As discussed in Section 4.4, a rewriting system R is a set of equations that have been oriented into rewrite rules. When a rewriting system is terminating, all terms have normal forms; when it is also convergent, all terms have unique normal forms. Convergent rewriting systems provide decision procedures for equational theories: if R is convergent, then $R \models t_1 = t_2$ if and only if t_1 and t_2 have the same canonical form. Hence reduction to normal form provides a decision procedure for the equational theory of a convergent rewriting system. This section provides more details about convergent term-rewriting systems.

Two terms t and u are *joinable* in R if there exists a term w such that $t \rightsquigarrow_R^* w$ and $u \rightsquigarrow_R^* w$. R is *locally confluent* if for every t, u, v such that $t \rightsquigarrow u$ and $t \rightsquigarrow v$, the terms u and v are joinable. It is *globally confluent*, or simply *confluent* or *Church-Rosser*, if for every t, u, v such that $t \rightsquigarrow^* u$ and $t \rightsquigarrow^* v$, the terms u and v are joinable.

If R is confluent, the terminal form of any term, if it exists, is unique.[24] When a terminal form is unique, we call it a *canonical form*. When every term has a canonical form in $\rightsquigarrow$, we say that $\rightsquigarrow$ is *canonical*.

A canonical system is always confluent.[25] However, a confluent system need not be canonical (consider the nonterminating set $\{a \rightarrow b, b \rightarrow a\}$ of rewrite rules).

A rewriting system is *convergent* if it is terminating and confluent. A convergent rewriting system is canonical.[26] If a rewriting system R is convergent, the joinability of two terms is decidable by reducing both terms to their canonical form and checking whether they are identical. Furthermore, if R is convergent and E is the set of all equations $u == v$ such that R contains either $u \rightarrow v$ or $v \rightarrow u$, then $E \models u == v$ if and only if u and v have identical canonical forms in R.

The following important characterization of confluence in terminating rewriting systems is easily proved by induction (see Figure 16).

Diamond lemma [23]: A terminating rewriting system is confluent if and only if it is locally confluent.

Termination is essential for the diamond lemma, as is shown by the locally confluent, but not confluent, set $\{a \rightarrow b, a \rightarrow c, c \rightarrow a, c \rightarrow d\}$ of rewrite rules.

Despite the obvious advantages of convergent rewriting systems, nonconvergent systems are more often used in practice. Some interesting theories, including all undecidable ones, cannot be described by convergent systems. Sometimes a convergent system exists, but finding it is impractical. Sometimes a convergent system exists and is easy to find, but is impractical to use.

A.3 Unification

Unification was first described by Herbrand [17] in 1930. It was put to practical use in 1965 by Robinson [27] as the basic step in resolution, and is now most widely used in computer science for logic programming and for type inference systems such as that in ML [16].

Figure 17 shows a simple recursive implementation of ordinary unification. (When associative-

[24]Proof: Suppose $t \rightsquigarrow^* t_1$ and $t \rightsquigarrow^* t_2$, where t_1 and t_2 are terminal. By confluence, there exists a v such that $t_1 \rightsquigarrow^* v$ and $t_2 \rightsquigarrow^* v$. Since t_1 and t_2 are terminal, t_1, t_2, and v must be identical.

[25]Proof: Suppose $t \rightsquigarrow^* t_1$ and $t \rightsquigarrow^* t_2$. Canonicity implies that t_1 and t_2 both have unique terminal forms, which must be the same as the unique terminal form of t.

[26]Proof: Suppose that R is terminating and confluent. Termination implies that every term has at least one terminal form. Confluence implies that this form is unique.

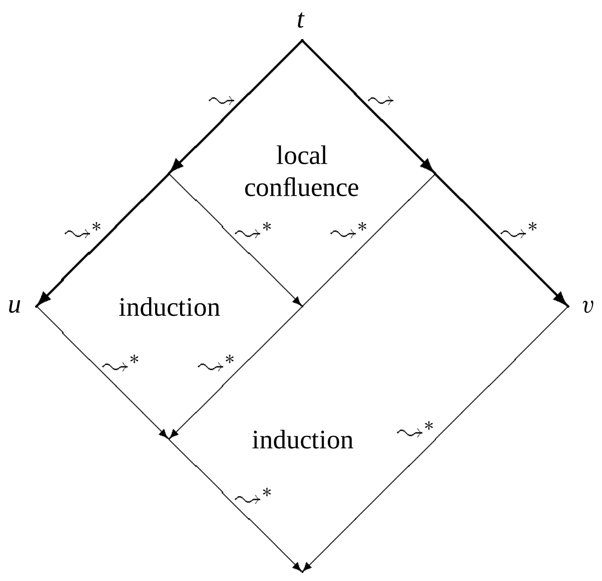

Figure 16: Proof of diamond lemma

commutative operators are present, unification becomes considerably more complicated. See [29].) Asymptotically better (e.g., linear) algorithms exist, but the ones that perform best in practice are similar to this. The algorithm can fail to unify two terms s and t in two ways: by a clash, which occurs when s and t are headed by different operators, and by a cycle, which prevents x from being unified with $f(x)$.

A.4 Critical pairs

The computation of critical-pair equations in ordinary term-rewriting was described in Section 5.3. In equational term-rewriting, it may be necessary to generalize the critical-pair computation to capture further ways in which a single term can be reduced. Such is the case when **ac** operators are present.

Suppose that $*$ is an **ac** operator, that l_1 is $t_1 * \ldots * t_n$, that $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ are two rewrite rules, and that $l_1 * x \rightarrow r_1 * x$ is not an instance of another rewrite rule. When computing critical-pair equations between these rewrite rules, we need to consider not only the overlaps of l_1 with l_2, but also the overlaps of $l_1 * x$ with l_2 as well. For example, there are no critical pairs between the rewrite rule $i(x) * x \rightarrow e$ and itself if $*$ is an ordinary operator. But if $*$ is **ac**, then we can unify $i(x) * x$ with the nonvariable subterm $y * z$ of $i(y) * y * z$ to obtain the critical-pair equation $i(e) * e == e$; furthermore, $i(i(x)) * i(x) * x$ and $i(x * y) * i(x) * x * y$ are unifications of $i(x) * x * y$ and $i(x') * x' * y'$, so that $e * x == i(i(x)) * e$ and $e * i(x) == i(x * y) * e * y$ are also critical-pair equations between $i(x) * x \rightarrow e$ and itself.

It is easy to see that for a finite rewriting system there are a finite number of critical pairs, and that these are effectively computable. This is important because of the following lemma.

Critical-pair lemma [19, 26]: A rewriting system is locally confluent if and only if every critical pair is joinable.

In conjunction with the diamond lemma, the critical-pair lemma provides a way to decide the confluence of terminating rewriting systems. First compute all critical-pair equations among the rewrite rules. If each of them normalizes to an identity, the rewriting system is locally, and therefore globally, confluent.

```
*unify* = **proc** ($s$, $t$: *term*) **returns** (*substitution*) **signals** (*failure*)
    **guard** $s$, $t$ are both variables ::
        **return**($\{s$ for $t\}$)
    **guard** $s$ is a variable and $t$ is not ::
        **if** $s$ occurs in $t$ **then** *failure(cycle)* **else return**($\{t$ for $s\}$)
    **guard** $t$ is a variable and $s$ is not ::
        **if** $t$ occurs in $s$ **then** *failure(cycle)* **else return**($\{s$ for $t\}$)
    **guard** $s$ is $f(s_1, \ldots, s_m)$, $t$ is $g(t_1, \ldots, t_n)$, and $f \neq g$ ::
        *failure(clash)*
    **guard** $s$ is $f(s_1, \ldots, s_n)$ and $t$ is $f(t_1, \ldots, t_n)$ ::
        $\sigma := \{\}$
        **for** $i := 1$ **to** $n$ **do** $\sigma := unify(\sigma(s_i), \sigma(t_i)) \circ \sigma$
            **resignal** *failure*
        **return**($\sigma$)
    **end** *unify*
```

Figure 17: Recursive implementation of unification

If any critical-pair equation does not normalize to an identity, the rewriting system is not locally (or globally) confluent. Note the vital role played by termination. It enables us to invoke the diamond lemma and consider only local confluence, and it assures us that the test for joinability of critical pairs terminates.

A.5 Completion

Given a method for orienting equations into terminating rewriting systems, the critical-pair computation can be used to *complete* a set of rewrite rules and thereby build a decision procedure for a set of equations. For example, we can add the nonjoinable critical-pair equation $e * z == i(y) * (y * z)$ to the three axioms for group theory and continue computing critical pairs in an attempt to arrive at a convergent set of rewrite rules.

In general, given a set of equations E, we can execute the nondeterministic procedure in Figure 18 to compute sets E_n and R_n of equations and rewrite rules such that $E_n \cup R_n$ has the same equational theory as E. If the computation reaches a point where all guards are false and E_n is empty, then R_n is convergent and can be used to decide the equational theory of E. If it reaches a point where all the guards are false and E_n is nonempty, then E_n contains consequences of E that cannot be oriented into rewrite rules without causing R_n to be nonterminating. If the computation never terminates, but is nonetheless fair (i.e., no guard remains true forever without its command being executed), then the equational theory of E is not decidable, but normalization with respect to the successive sets R_n of rewrite rules provides a semidecision procedure for this equational theory.

There are several reasons why the completion procedure may fail to orient an equation. Sometimes an equation cannot be oriented because each side contains some variable that the other does not. Such an equation is called *incompatible*. Sometimes an equation (e.g, $x + y == y + x$) cannot be oriented into a terminating rewrite rule. And sometimes decisions made when orienting other rewrite rules in R may prevent the remainder of E from being oriented while still preserving termination.[27]

[27] The original formulation of the completion procedure used a fixed ordering on terms to orient equations, so there were no

```
$R := \{\}$
do while any guard is true
    guard $s == t \in E \wedge terminates(R \cup \{s \to t\})$ ::
        $R := R \cup \{s \to t\}$
        $E := E - \{s == t\}$
    guard $s == t \in E \wedge terminates(R \cup \{t \to s\})$ ::
        $R := R \cup \{t \to s\}$
        $E := E - \{s == t\}$
    guard $u \rightsquigarrow_R^* s \wedge u \rightsquigarrow_R^* t \wedge \neg joinable_R(s, t)$ ::
        $E := E \cup \{s == t\}$
    guard $s == t \in E \wedge s \rightsquigarrow_R^+ u$ ::
        $E := E \cup \{u == t\} - \{s == t\}$
    guard $s == t \in E \wedge t \rightsquigarrow_R^+ u$ ::
        $E := E \cup \{s == u\} - \{s == t\}$
    guard $s == s \in E$ ::
        $E := E - \{s == s\}$
    end
```

Figure 18: Abstract completion procedure

To implement the abstract version of the completion procedure, it is necessary to replace the first two guards by conditions that are decidable (as noted earlier, termination is undecidable). The next subsection describes several decidable conditions that ensure termination. It is also possible to optimize the completion procedure, as follows. The third guard can be restricted to cases in which $\{s, t\}$ is a nonjoinable critical pair of the rewrite rules in R. That this is sufficient follows immediately from the critical-pair lemma. Another useful optimization is to keep all equations and rules in normal form with respect to R. Such a system is called *internormalized*. A procedure incorporating these optimizations was first described in [19] and is well known as the Knuth-Bendix completion procedure. The completion procedure was extended to handle associative-commutative operators in [26]. The description presented here is closer to the one appearing in [7] than to the earlier formulations.

A.6 Proving termination

Terminating rewriting systems are desirable for three reasons. If they are confluent, then their equational theories are decidable (by reduction to normal form). Furthermore, it is decidable whether terminating rewriting systems are confluent (by the critical-pair lemma), and we can try to complete them with the completion procedure when they are not confluent.

A relation $\rightsquigarrow$ is said to be *locally finite* if for every term t the set $\{u | t \rightsquigarrow u\}$ is finite. It is said to be *globally finite* if $\{u | t \rightsquigarrow^* u\}$ is finite. If R is finite, then $\rightsquigarrow_R$ is locally finite, but need not be globally finite.

A relation $\rightsquigarrow$ is *acyclic* if there is no term t such that $t \rightsquigarrow^+ t$. If $\rightsquigarrow$ is globally finite and acyclic, it must

nondeterministic ordering choices to be made. When the ordering is fixed, there is at most one convergent rewriting system corresponding to an equational theory [6].

be terminating.[28] The converse is not true: if $\rightsquigarrow$ is terminating, it is not necessarily globally finite.[29] However, a locally finite relation is terminating if and only if it is both globally finite and acyclic.[30] An important corollary is that the rewriting relation of a finite set of rules is terminating if and only if it is both globally finite and acyclic.

Although termination is undecidable, there are methods that can be used to prove the termination of many rewriting systems. Most seek to embed the (inverse of) the rewriting relation $\rightsquigarrow_R$ in a *well-founded* relation, that is, in a relation $>$ that has no infinite decreasing sequence $t_1 > t_2 > t_3 > \ldots$.

A.6.1 Simplification orderings

A *simplification ordering* [6] is a partial ordering $\sqsupset$ (i.e., a transitive, irreflexive binary relation) that is *monotonic*, in other words,

$$s \sqsupset t \Rightarrow f(\ldots, s, \ldots) \sqsupset f(\ldots, t, \ldots)$$

and that has the *subterm property*, in other words,

$$f(\ldots, t, \ldots) \sqsupset t$$

Consider, for example, orderings on the term algebra generated by 0, 1, and +. Let $num(f, u)$ be the number of times the function symbol f occurs in the term u. The ordering $s \sqsupset t$ if and only if $num(1, s) > num(1, t)$ is a simplification ordering. The ordering $s \sqsupset t$ if and only if $num(1, s) - num(0, s) > num(1, t) - num(0, t)$ is not a simplification ordering, because it does not have the subterm property.

A rewriting system R terminates if there exists a simplification ordering $\sqsupset$ such that $\sigma(s) \sqsupset \sigma(t)$ for all substitutions σ and all rewrite rules $s \rightarrow t$ in R (see [6]). This result provides a means of proving termination that is independent of the set of terms one might attempt to reduce. However, since the number of substitution instances of rewrite rules is usually infinite, it is hard to apply directly. This leads us to require that the ordering be *stable*, in other words, that

$$s \sqsupset t \Rightarrow \sigma(s) \sqsupset \sigma(t)$$

A rewriting system R terminates if there exists a stable simplification ordering $\sqsupset$ such that $s \sqsupset t$ for all rewrite rules $s \rightarrow t$ in R.

A.6.2 Registered orderings

A *registered ordering* is a function from registries to stable simplification orderings. A *registry* is a pair $\langle \pi, \psi \rangle$, where π is a precedence relation on operators and ψ a status map.

A *status* map is a partial mapping from operators to the set {multiset, left-to-right, right-to-left}. A *precedence* is a pair of binary relations $\langle \geq, \neq \rangle$, on operators such that $\geq$ is reflexive and transitive, $\neq$ is irreflexive and symmetric, and for any three operators f, g and h, $(f \geq g \wedge g \geq h \wedge (f \neq g \vee g \neq h)) \Rightarrow f \neq h$.

[28]Proof: Any nonterminating sequence $t_1 \rightsquigarrow t_2 \rightsquigarrow \ldots$ would either have to repeat some elements, in which case $\rightsquigarrow$ is cyclic, or contain infinitely many distinct elements, in which case $\rightsquigarrow$ is globally infinite.

[29]Consider the relation in which $1 \rightsquigarrow n$ for all $n > 1$.

[30]Proof: We have just shown one direction. Conversely, if $\rightsquigarrow$ is cyclic, then it is clearly not terminating. Suppose it is not globally finite. Since it is locally finite, some term t_1 has infinitely many descendents. By König's lemma, there is an infinite sequence $t_1 \rightsquigarrow t_2 \rightsquigarrow \ldots$ starting at that term.

Two operators are comparable under π if they are comparable under $\geq$. We define $f > g$ to mean $f \geq g$ and $f \neq g$, and $f = g$ to mean $f \geq g$ and $g \geq f$. The relation $>$ is a partial ordering. A precedence is total over a set of operators if and only if for all operators f and g in the set, $f > g, g > f$ or $f = g$.

When using a registered simplification ordering O, LP orients an equation $s == t$ into a rewrite rule only when s and t are related in the ordering generated by applying O to the current registry reg, that is, only if $s >_{O(reg)} t$ or $t >_{O(reg)} s$. If the equation has been entered using the syntax $s \rightarrow t$, the equation will be oriented only if $s >_{O(reg)} t$ in the ordering.

LP's **dsmpos** ordering is a registered simplification ordering. When using conventional (as opposed to equational) term-rewriting, it can be used to prove that the rewriting system terminates. It works as follows. Let s and t be two terms, with $s = f(s_1, \ldots, s_m)$ and $t = g(t_1, \ldots, t_n)$. Then $s \geq t$ in the **dsmpos** ordering iff

- $s_i \geq t$ for some i, or
- $f > g$ (in the registry) and $s > t_i$ for all i, or
- $f = g$ (in the registry), or $f \geq g$ (in the registry) and $s > t_i$ for all i, and
  - f and g can have multiset status and $\{s_1, \ldots, s_m\}$ is greater than or equal to $\{t_1, \ldots, t_n\}$ as a multiset, or
  - f and g have lexicographic status (i.e., right-to-left or left-to-right), $s > t_i$ for all i, and $\langle s_1, \ldots, s_m \rangle$ is greater than or equal to $\langle t_1, \ldots, t_n \rangle$ in lexicographic order.

Here M_1 is less than M_2 as a multiset if and only if for every element m_1 that occurs with greater multiplicity in M_1 than in M_2 there is an element m_2 such that $m_1 < m_2$ and m_2 occurs with greater multiplicity in M_2 than in M_1.

Note that if an operator has multiset status, the ordering produced by the registered ordering treats the arguments of that operator as a multiset, that is, their order is irrelevant. If an operator has left-to-right or right-to-left status, a lexicographic ordering is produced in which either the leftmost or rightmost arguments are given extra weight. Consider, for example, the equation $f(a, b) == f(b, a)$, where $a > b$ in the precedence. This equation cannot be ordered if f has multiset status; it will be ordered to $f(a, b) \rightarrow f(b, a)$ if f has left-to-right status and to $f(b, a) \rightarrow f(a, b)$ if f has right-to-left status.

When asked to orient an equation that cannot be oriented using its current registry, LP attempts to find a minimal extension to the registry that allows the equation to be oriented. If LP succeeds, it can extend the registry and orient the equation. The algorithm in Figure 19 is an abstraction of the one used by LP to orient a set of equations into a set of rewrite rules. This algorithm relies on the fact that the ordering $>_{O(reg)}$ is monotonic with respect to extensions to reg.

The function *Extensions* computes the set of all minimal extensions to the registry that make it possible to order the pair of terms.

```
% $reg$ is the current registry
% $R$ is the current set of rewrite rules
% $E$ is the current set of equations
% $O$ is the current registered ordering

for each $s == t \in E$ do
    guard $s >_{O(reg)} t$ ::
        $R := R \cup \{s \rightarrow t\}$
        $E := E - \{s \rightarrow t\}$
    guard $t >_{O(reg)} s$ ::
        $R := R \cup \{t \rightarrow s\}$
        $E := E - \{s \rightarrow t\}$
    guard neither $s >_{O(reg)} t$ nor $t >_{O(reg)} s$ ::
        $Ext : Set[Registry] := Extensions(reg, s, t)$
        if $isEmpty(Ext)$ then failure else
            $reg := choose(Ext)$
            $R := R \cup if(s >_{O(reg)} t, \{s \rightarrow t\}, \{t \rightarrow s\})$
            $E := E - \{s \rightarrow t\}$
    end
```

Figure 19: Abstract ordering procedure

B Sample proof

Figure 20 shows the contents of a file *sample.lp* that contains LP commands for proving two simple theorems about sets. The following transcript shows the output produced by LP as it executes *sample.lp*. In addition to our usual typesetting conventions (see Section 3.1), we have underlined all input to LP, whether typed by the user or read from the file *sample.lp*. We have also condensed the transcript slightly by omitting some of the less interesting LP output, but none of the input.

```
declare sorts Elem, Set
declare variables e, e’: Elem
declare variables x, y, z: Set
declare operators
  empty:               -> Set
  singleton: Elem      -> Set
  \union:    Set, Set  -> Set
  \in:       Elem, Set -> Bool
  insert:    Elem, Set -> Set
  ..

set name set
assert ac \union
assert Set generated by empty, singleton, \union
assert
  e \in empty == false
  e \in singleton(e’) == e = e’
  e \in (x \union y) == e \in x | e \in y
  insert(e, x) == singleton(e) \union x
  ..

set name extensionality
assert Set partitioned by \in
display extensionality

set name thm
prove x = x \union x
  instantiate s1 by x, s2 by x \union x in extensionality
  qed

set proof-methods =>, normalization
prove e \in x => insert(e, x) = x by induction
  resume by cases ec \in xc, ec \in xc1
    critical-pairs thmCaseHyp with thmInductHyp
    critical-pairs thmCaseHyp with thmInductHyp
  qed
quit
```

Figure 20: File *sample.lp* of commands for proof

```
LP1: execute sample
LP1.1: declare sorts Elem, Set
LP1.2: declare variables e, e′: Elem
LP1.3: declare variables x, y, z: Set
LP1.4: declare operators
       empty: → Set
       singleton: Elem → Set
       ∪: Set, Set → Set
       ∈: Elem, Set → Bool
       insert: Elem, Set → Set
       ..
LP1.5:
LP1.6: set name set
LP1.7: assert ac ∪
LP1.8: assert Set generated by empty, singleton, ∪
LP1.9: assert
       e ∈ empty == false
       e ∈ singleton(e′) == e = e′
       e ∈ (x ∪ y) == e ∈ x | e ∈ y
       insert(e, x) == singleton(e) ∪ x
       ..
LP1.10:
LP1.11: set name extensionality
LP1.12: assert Set partitioned by ∈
LP1.13: display extensionality
Deduction rules:
extensionality.1: when (forall e) e ∈ s1 == e ∈ s2 yield s1 == s2
LP1.14:
LP1.15: set name thm
LP1.16: prove x == x ∪ x
Conjecture thm.1: x == x ∪ x
Proof suspended.
LP1.17: instantiate s1 by x, s2 by x ∪ x in extensionality
Deduction rule extensionality.1 has been instantiated to deduction rule extensionality.1.1,
       when (forall e) e ∈ x == e ∈ (x ∪ x) yield x == x ∪ x
which was normalized to equation extensionality.1.1.1, x == x ∪ x
Conjecture thm.1: x == x ∪ x
[ ] Proved by normalization.
LP1.18: qed
All conjectures have been proved.
LP1.19:
```

LP1.20: **set proof-methods $\Rightarrow$, normalization**

LP1.21: **prove** $e \in x \Rightarrow insert(e, x) = x$ **by induction**

Conjecture *thm.2*: Subgoals for proof by induction on 'x'
Basis subgoals:
  thm.2.1: $e \in empty \Rightarrow empty = insert(e, empty) == true$
  thm.2.2: $e \in singleton(e') \Rightarrow insert(e, singleton(e')) = singleton(e') == true$
Induction constants: xc, $xc1$
Induction hypotheses:
  thmInductHyp.1: $e \in xc \Rightarrow insert(e, xc) = xc == true$
  thmInductHyp.2: $e \in xc1 \Rightarrow insert(e, xc1) = xc1 == true$
Induction subgoal:
  thm.2.3: $e \in (xc1 \cup xc) \Rightarrow insert(e, xc1 \cup xc) = xc1 \cup xc == true$

Subgoal *thm.2.1*: Subgoal for proof of $\Rightarrow$
New constant: ec
Hypothesis:
  thmImpliesHyp.1: $ec \in empty == true$
Subgoal:
  thm.2.1.1: $empty = insert(ec, empty) == true$

Subgoal *thm.2.1.1*: $empty = insert(ec, empty) == true$
[ ] Proved by inconsistent hypothesis.

Subgoal *thm.2.1*: $e \in empty \Rightarrow empty = insert(e, empty) == true$
[ ] Proved $\Rightarrow$.

Subgoal thm.2.2: Subgoal for proof of $\Rightarrow$
New constants: ec, $e'c$
Hypothesis:
  thmImpliesHyp.2: $ec \in singleton(e'c) == true$
Subgoal:
  thm.2.2.1: $insert(ec, singleton(e'c)) = singleton(e'c) == true$

Deduction rule *lp_equals_is_true* has been applied to equation *thmImpliesHyp.2* to yield equation *thmImpliesHyp.2.1*, $e'c == ec$, which implies *thmImpliesHyp.2*.

Subgoal *thm.2.2.1*: $insert(ec, singleton(e'c)) = singleton(e'c) == true$
[ ] Proved by normalization.

Subgoal *thm.2.2*: $e \in singleton(e') \Rightarrow insert(e, singleton(e')) = singleton(e') == true$
[ ] Proved $\Rightarrow$.

Subgoal *thm.2.3*: Subgoal for proof of $\Rightarrow$
New constant: ec
Hypothesis:
  thmImpliesHyp.3: $ec \in (xc1 \cup xc) == true$
Subgoal:
  thm.2.3.1: $insert(ec, xc1 \cup xc) = xc1 \cup xc == true$

Subgoal *thm.2.3.1*: $insert(ec, xc1 \cup xc) = xc1 \cup xc == true$
  Current subgoal: $singleton(ec) \cup xc1 \cup xc = xc1 \cup xc == true$
Proof suspended.

LP1.22: **resume by cases** $ec \in xc$, $ec \in xc1$

Subgoal *thm.2.3.1*: Subgoals for proof by cases
First subgoal:

Cases.1: $ec \in xc \mid ec \in xc1 == true$
Case hypotheses:
thmCaseHyp.1.1: $ec \in xc == true$
thmCaseHyp.1.2: $ec \in xc1 == true$
Subgoal for cases:
thm.2.3.1.1:2: $singleton(ec) \cup xc1 \cup xc = xc1 \cup xc == true$

Subgoal *Cases.1*: $ec \in xc \mid ec \in xc1 == true$
[ ] Proved by normalization.

Added hypothesis *thmCaseHyp.1.1* to the system.

Subgoal *thm.2.3.1.1*: $singleton(ec) \cup xc1 \cup xc = xc1 \cup xc == true$
Proof suspended.

LP1.23: <u>**critical-pairs *thmCaseHyp* with *thmInductHyp***</u>

A critical pair between rewrite rules *thmCaseHyp.1.1* and *thmInductHyp.1* is
thm.3: $singleton(ec) \cup xc = xc == true$

Deduction rule *lp_equals_is_true* has been applied to equation *thm.3* to yield equation *thm.3.1*, $singleton(ec) \cup xc == xc$, which implies *thm.3*.

Critical pair computation abandoned because a theorem has been proved.

Subgoal *thm.2.3.1.1*: $singleton(ec) \cup xc1 \cup xc = xc1 \cup xc == true$
[ ] Proved by normalization.

Added hypothesis *thmCaseHyp.1.2* to the system.

Subgoal *thm.2.3.1.2*: $singleton(ec) \cup xc1 \cup xc = xc1 \cup xc == true$
Proof suspended.

LP1.24: <u>**critical-pairs *thmCaseHyp* with *thmInductHyp***</u>

A critical pair between rewrite rules *thmCaseHyp.1.2* and *thmInductHyp.2* is
thm.4: $singleton(ec) \cup xc1 = xc1 == true$

Deduction rule *lp_equals_is_true* has been applied to equation *thm.4* to yield equation *thm.4.1*, $singleton(ec) \cup xc1 == xc1$, which implies *thm.4*.

Critical pair computation abandoned because a theorem has been proved.

Subgoal *thm.2.3.1.2*: $singleton(ec) \cup xc1 \cup xc = xc1 \cup xc == true$
[ ] Proved by normalization.

Subgoal *thm.2.3.1*: $insert(ec, xc1 \cup xc) = xc1 \cup xc == true$
[ ] Proved by cases $ec \in xc$, $ec \in xc1$.

Subgoal *thm.2.3*: $e \in (xc1 \cup xc) \Rightarrow insert(e, xc1 \cup xc) = xc1 \cup xc == true$
[ ] Proved $\Rightarrow$.

Conjecture *thm.2*: $e \in x \Rightarrow insert(e, x) = x == true$
[ ] Proved by induction on 'x'.

LP1.25: <u>**qed**</u>

All conjectures have been proved.

LP1.26: <u>**quit**</u>

References

[1] Ben Cherifa, A. and Lescanne, P. “An actual implementation of a procedure that mechanically proves termination of rewriting systems based on inequalities between polynomial interpretations,” *Proceedings of the 8th International Conference on Automated Deduction*, Oxford, England, *Lecture Notes in Computer Science* 230, Springer-Verlag, July 1986, 42–51.

[2] Birkhoff, G. “On the structure of abstract algebras,” *Proceedings of the Cambridge Philosophical Society* 31 (1935), 433-454.

[3] Boyer, R. S. and Moore, J S. *A Computational Logic*, New York: Academic Press, 1979.

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

[5] Church, A. “An unsolvable problem of elementary number theory,” *American Journal of Mathematics* 58 (1936), 345–363.

[6] Dershowitz, N. “Orderings for term-rewriting systems,” *Theoretical Computer Science* 17:3 (March 1982), 279–301.

[7] Dershowitz, N. and Jouannaud, J.-P., “Rewrite systems,” *Handbook of Theoretical Computer Science*, Volume B, Chapter 15, North-Holland, 1989.

[8] Detlefs, D. and Forgaard, R. “A procedure for automatically proving the termination of a set of rewrite rules,” *Proceedings of the First International Conference on Rewriting Techniques and Applications*, Dijon, France, *Lecture Notes in Computer Science* 202, Springer-Verlag, May 1985, 255–270.

[9] Garland, S. J. and Guttag, J. V. “Inductive methods for reasoning about abstract data types,” *Proceedings of the 15th ACM Conference on Principles of Programming Languages*, San Diego, California, January 1988, 219–228.

[10] Garland, S. J., Guttag, J. V. and Staunstrup, J. “Verification of VLSI circuits using LP,” *Proceedings of the IFIP WG 10.2 Conference on the Fusion of Hardware Design and Verification*, North Holland, 1988, 329–345.

[11] Garland, S. J. and Guttag, J. V. “An overview of LP, the Larch Prover,” *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, Chapel Hill, N.C., *Lecture Notes in Computer Science* 355, Springer-Verlag, 1989, 137–151.

[12] Garland, S. J., Guttag, J. V. and Horning, J. J. “Debugging Larch Shared Language specifications,” *IEEE Transactions on Software Engineering* 16:9 (September 1990), 1044–1057. Also available from Digital Equipment Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, as Report 60 (July 1990).

[13] Gödel, K. “Die Vollständigkeit der Axiome des logischen Funktionenkalküls,” *Monatshefte für Mathematik und Physik* 37 (1930), 349–360.

[14] Guttag, J. V. and Horning, J. J. “Report on the Larch Shared Language” and “A Larch Shared Language Handbook,” *Science of Computer Programming* 6:2 (March 1986), 103–157.

[15] Guttag, J. V., Horning, J. J., and Modet, A. "Report on the Larch Shared Language, Version 2.3" Digital Equipment Corporation Systems Research Center Report 58, May 1990.

[16] Harper, R. *Report on Standard ML*, Report ECS-LFCS-86-14, Department of Computer Science, University of Edinburgh, 1986.

[17] Herbrand, J. *Recherches sur la théorie de la démonstration*, Travaux de la Société des Sciences et des Lettres de Varsovie, Classe III sciences mathématiques et physiques 33 (1930), 128 pp. Translation in *Logical Writings*, Harvard University Press, 1971.

[18] Hsiang, J. and Dershowitz, N. "Rewrite methods for clausal and nonclausal theorem proving," *Proceedings of the 10th EATCS International Colloquium on Automata, Languages, and Programming*, Barcelona, Spain, *Lecture Notes in Computer Science* 154, Springer-Verlag, July 1983, 331–346.

[19] Knuth, D. E. and Bendix, P. B. "Simple word problems in universal algebras," in *Computational Problems in Abstract Algebra*, J. Leech (ed.), Pergamon Press, Oxford, England, 1969, 263–297.

[20] Lamport, L. *LaTeX: A Document Preparation System*, Addison-Wesley Publishing Company, 1986.

[21] Lamport, L. "A Temporal Logic of Actions," Digital Equipment Corporation Systems Research Center Report 57, April 1990.

[22] Lescanne, P. "REVE: a rewrite rule laboratory," *Proceedings of the 8th International Conference on Automated Deduction*, Oxford, England, *Lecture Notes in Computer Science* 230, Springer-Verlag, July 1986, 695–696.

[23] Newman, M. H. A. "On theories with a combinatorial definition of 'equivalence'," *Annals of Mathematics* 59: 4 (October, 1942), 223-243.

[24] Paulson, L. C. *Logic and Computation: Interactive Proof with Cambridge LCF,* Cambridge University Press, Cambridge, 1987.

[25] Paulson, L. C. "The foundation of a generic theorem prover," Technical Report No. 130, University of Cambridge Computer Laboratory, March 1988.

[26] Peterson, G. L. and Stickel, M. E. "Complete sets of reductions for some equational theories," *Journal of the ACM* 28:2 (Apr. 1981), 233–264.

[27] Robinson, J. A. "A machine-oriented logic based on the resolution principle," *Journal of the ACM* 12 (1965), 23–41.

[28] Saxe, J. B., Garland, S. J., Guttag, J. V., and Horning, J. J., "Using transformations and verification in circuit design," Digital Equipment Corporation Systems Research Center Report 78, September 1991.

[29] Siekmann, J. H. "An introduction to unification theory," *Formal Techniques in Artificial Intelligence: A Sourcebook*, R. B. Banerji (ed.), North-Holland, 1990, 369–424.

[30] Staunstrup, J., Garland, S. J., and Guttag, J. V. "Localized verification of circuit descriptions," *Proceedings of an International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, *Lecture Notes in Computer Science* 407, Springer-Verlag, 1989, 349–364.

[31] Stickel, M. E. “A case study of theorem proving by the Knuth-Bendix method: discovering that $x^3 = x$ implies ring commutativity,” *Proceedings of the 7th International Conference on Automated Deduction*, Napa, California, *Lecture Notes in Computer Science* 170, Springer-Verlag, May 1984, 248–258.

[32] Zhegalkin, I. I. “On a technique of evaluation of propositions in symbolic logic,” *Matematisheskii Sbornik* 34:1 (1927), 9–27.