LP, the Larch Prover: User and Reference Manual

Stephen J. Garland and John V. Guttag *
Massachusetts Institute of Technology †

December 30, 1994

Abstract

This guide provides an introduction to LP (the Larch Prover), Release 3.1. It describes how LP can be used to axiomatize theories in 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.

*Research supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-92-J-1795, by the National Science Foundation under grant CCR-9115797, and by the Digital Equipment Corporation.

†Authors' address: MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139. E-mail: garland@lcs.mit.edu, guttag@lcs.mit.edu

Contents

1 Introduction

LP is an interactive theorem proving system for multisorted first-order logic. It is currently used at MIT and elsewhere to reason about designs for circuits, concurrent algorithms, hardware, and software. Unlike most theorem provers, which attempt to find proofs automatically for correctly stated conjectures, LP is intended to assist users in finding and correcting flaws in conjectures—the predominant activity in the early stages of the design process.

LP works efficiently on large problems, has many important user amenities, and can be used by relatively naive users. It was developed and is being maintained by Stephen J. Garland and John V. Guttag at the MIT Laboratory for Computer Science.

For a general introduction to LP, see the following topics.

- Getting the current distribution of LP
- News about recent changes to LP
- Using LP
  - Some sample proofs
  - Logical syntax and semantics
  - Operational syntax and semantics
  - Forward inference mechanisms for using axioms
  - Backward inference mechanisms for proving conjectures
  - Commands recognized by LP
- Design philosophy
- Table of contents for this documentation

This is a preliminary version of the documentation for LP. A more polished version, both in hypertext and in hardcopy, will be available in early 1995.

1.1 Design philosophy

The philosophy behind LP is different from that underlying most theorem provers. Its design is based on the observation that initial attempts to state interesting conjectures correctly, and then prove them, hardly ever succeed on the first try. Sometimes the conjecture is wrong. Sometimes the formalization is incorrect or incomplete. Sometimes the proof strategy is flawed or not detailed enough. As a result, LP is designed to assist in reasoning by carrying out routine (and possibly lengthy) steps in a proof automatically and by providing useful information about why proofs fail, if and when they do.

Because conjectures are often flawed, LP is not designed to find difficult proofs automatically. For example, LP does not use heuristics to formulate additional conjectures that might be useful in a proof. Instead, LP makes it easy for users to employ standard techniques such as simplification and proofs by cases, induction, and contradiction, either to construct proofs or to understand why they have failed.

LP also provides support for rechecking proofs following changes in axioms, conjectures, or proof strategies. This support assists users in regression testing during proof construction: when users discover they must change a formalization in order to establish some conjecture, regression testing ensures that the change does not invalidate previously constructed proofs. This support also promotes proof re-use: users can edit old proofs to prove new conjectures, and LP will check that the progress of the proof stays "in sync" with the users' intentions.

1.2 A sample proof

We illustrate how to use LP by presenting some sample proofs along with explanatory comments. The proofs show how to derive some basic facts about finite sets from a simple axiomatization.

- Getting started
- Sample declarations for sorts, operators
- Sample axioms for finite sets
- Useful kinds of axioms
- Sample conjectures
- Two easy theorems
- Three theorems about union
- Alternative proofs of theorems about union
- Three theorems about subset
- An alternate induction rule
- Two final theorems about subset
- How to guide a proof

1.2.1 Getting started

To invoke LP, type `lp` at the Unix command prompt. LP will respond with the following message:

```
Welcome to LP (the Larch Prover), Release 3.1 (94/12/30).
Copyright (C) 1994, S. J. Garland and J. V. Guttag
Type ‘help lp’ (followed by a carriage return) for help.

LP1:
```

LP is a command-driven system. The last line of LP's welcoming message, `LP1:`, is a prompt for the first command. LP prompts users to enter subsequent commands with `LP2:`, `LP3:`, ...

You can type input directly in response to LP's prompts, or you can create a file of LP commands (e.g., set1.lp) and then use LP's execute command to cause LP to execute the commands in that file as if you had typed them directly. Here's the start of a session in which commands are executed from the file `set1.lp`:

```
LP1: execute set1

LP1.1: % Some simple theorems about finite sets

LP1.2:
```

The first command, `execute set1`, was typed by the user. The second was obtained from the file `set1.lp`. This command is a comment: LP ignores all input following a percent sign. The prompts `LP1.1:` and `LP1.2:` indicate that LP is taking input from the file being executed as a result of the command entered in response to the prompt `LP1:`.

1.2.2 Sample declarations

for sorts, operators

The first three commands in set1.lp declare symbols for use in axiomatizing the properties of sets of elements.

```
declare sorts E, S
declare variables e, e1, e2: E, x, y, z: S
declare operators
  {}:                 -> S
  {__}:           E    -> S
  insert:         E, S -> S
  __ \union __:   S, S -> S
  __ \in __:      E, S -> Bool
  __ \subseteq __: S, S -> Bool
  ..
```

The first declare command introduces names for two sorts, `E` and `S`. LP predefines the boolean sort `Bool`.

The second command introduces variables ranging over `E` and `S`. These variables will be used when stating axioms and conjectures.

The third command introduces symbols for the operators whose properties we will axiomatize. This command uses LP's multi-line input convention: if the arguments for a command do not fit on the line containing the command, they can be given on subsequent lines. Two periods (`..`) mark the end of the command.

The declaration for each operator specifies sorts for the operator's arguments and the sort of its result. Operators like {} with no arguments are constants. Operators like `insert` are used in functional notations like `insert(e, x)`. The placeholders in operators like {_} and _\in_ indicate where their arguments belong in notations like {`e`} and `e \in x`.

1.2.3 Sample axioms

for finite sets

The next several commands in set1.lp axiomatize the properties of finite sets of elements.

```
set name setAxioms
assert
  sort S generated by {}, insert;
  {e} = insert(e, {});
  ~(e \in {});
  e \in insert(e1, x) <=> e = e1 \/ e \in x;
  {} \subseteq x;
  insert(e, x) \subseteq y <=> e \in y /\ x \subseteq y;
  e \in (x \union y) <=> e \in x \/ e \in y
  ..
set name extensionality
assert \A e (e \in x <=> e \in y) => x = y
```

The set command directs LP to assign the names `setAxioms.1`, `setAxioms.2`, ... to the axioms introduced by the subsequent assert commands. When multiple axioms are asserted in a single command, they are separated by semicolons.

The axioms are formulated using declared symbols (for sorts, variables, and operators) together with logical symbols for equality (`=`), negation (`~`), conjunction (`/\`), disjunction (`\/`), implication (`=>`), logical equivalence (`<=>`), and universal quantification (`\A`). LP also provides a symbol for existential quantification (`\E`). LP uses a limited amount of \llink{precedence} when parsing formulas: for example, the logical operator `<=>` binds less tightly than the other logical operators, which bind less tightly than the equality operator, which bind less tightly than declared operators like `\in` and `\union`.

1.2.4 Useful kinds of axioms

The axioms in set1.lp fall into several categories:

Induction rules The first axiom, `sort Set generated by {}, insert`, asserts that all elements of sort `S` can be obtained by finitely many applications of `insert` to {}. It provides the basis for definitions and proofs by induction.

Explicit definitions The second axiom, `{e} = insert(e, {})`, is a single formula that defines the operator {__} (as a constructor for a singleton set).

Inductive definitions The next two pairs of axioms provide induction definitions of the membership operator `\in` and the subset operator `\subseteq`. Inductive definitions generally consist of one formula per generator.

Implicit definitions The final formula, `e \in (x \union y) <=> e \in x \/ e \in y`, in the first `assert` command, together with the other axioms, completely constrains the interpretation of the `\union` operator.

Constraining properties The second `assert` command formalizes the *principle of extensionality*, which asserts that any two sets with exactly the same elements must be the same set.

1.2.5 Sample conjectures

We will illustrate LP's proof mechanisms by proving the following sample conjectures:

```
set name setTheorems
prove e \in {e}
prove \E x \A e (e \in x <=> e = e1 \/ e = e2)
prove x \union {} = x
prove x \union insert(e, y) = insert(e, x \union y)
prove ac \union
prove e \in x /\ x \subseteq y => e \in y
prove x \subseteq y /\ y \subseteq x => x = y
prove (x \union y) \subseteq z <=> x \subseteq z /\ y \subseteq z
prove sort S generated by {}, {__}, \union
prove x \subseteq (x \union y)
prove x \subseteq x
```

Except for the fourth, the sample conjectures are like the sample axioms: they are either formulas or induction rules. The fourth, `ac \union`, is an abbreviation for the conjunction of the associative and commutative laws for the `\union` operator. It provides LP with useful operational information. For example, it allows LP to conclude that `{} \union x` is the same set as `x \union {}`; hence the third conjecture shows that both of these sets are the same as `x`.

The order in which we have stated these conjectures is not completely arbitrary. As we shall see, some of them are used to prove conjectures appearing later in the list.

1.2.6 Two easy theorems

The proof of the first theorem in set1.lp is straightforward.

```
set name setTheorems
prove e \in {e}
qed
```

The prove command directs LP to initiate the proof of a conjecture, and the qed command directs LP to confirm that its proof is complete. LP proves this conjecture automatically by using the user-supplied axioms as rewrite rules. When using a formula as a rewrite rule, either LP rewrites terms matching the entire formula to `true` or, when the principal connective of the formula is = (equals) or <=> (if and only if), LP rewrites terms matching the left side of the formula to terms matching the right. Occasionally LP will reverse the order of terms in an equality to ensure that the resulting set of rewrite rules does not produce nonterminating (i.e., infinite) rewriting sequences. Here's how LP proves the first conjecture:

```
e \in {e} ~> e \in insert(e, {})    by setAxioms.2
          ~> e = e \/ e \in {}      by setAxioms.4
          ~> true \/ e \in {}       by a hardwired axiom for =
          ~> true                   by a hardwired axiom for \/
```

Most proofs require user guidance. For example, to prove that for any two elements `e1` and `e2`, there is a set that contains exactly these two elements, the user must provide a "witness" for the existential quantifier in the second conjecture:

```
prove \E x \A e (e \in x <=> e = e1 \/ e = e2)
  resume by specializing x to insert(e2, {e1})
qed
```

Users can specify a proof method for a conjecture either as part of the prove command or in a separate resume command, which continues the proof of the conjecture.

1.2.7 Three theorems

about union

The next three theorems in set1.lp follow from the principle of extensionality.

```
prove x \union {} = x
  instantiate y by x \union {} in extensionality
qed
```

The instantiate command directs LP to form the substitution instance

```
\A e (e \in x <=> e \in x \union {}) => x = x \union {}
```

of the fact name `extensionality`. LP rewrites this formula automatically to

```
\A e (e \in x <=> e \in x \/ e \in {}) => x = x \union {}
```

using the definition of `\union`, then to

```
\A e (e \in x <=> e \in x \/ false) => x = x \union {}
```

using the definition of `\in`, and then successively to

```
\A e (e \in x <=> e \in x) => x = x \union {}
\A e (true) => x = x \union {}
true => x = x \union {}
x = x \union {}
```

using LP's hardwired axioms. LP orients this final simplification into the rewrite rule `x \union {} -> x`, which it uses to simplify the conjecture to `true`.

Two other theorems about union can also be proved by instantiating the extensionality axiom. Both proofs are left as exercises to the reader.

```
prove x \union insert(e, y) = insert(e, x \union y)
prove ac \union
```

1.2.8 Alternative proofs

of theorems about union

Explicit instantiation is not the only way to prove the three conjectures about union in set1.lp. Here's another technique:

```
prove x \union {} = x by contradiction
  critical-pairs *Hyp with extensionality
qed
```

The prove command directs LP to begin an attempt to prove the conjecture by contradiction. LP does this by adding the negation, `~(xc \union {} = xc)`, of the conjecture as a hypothesis to its logical system. It replaces the variable `x` in the conjecture by the constant `xc`, because the negation of a conjecture about *all* sets `x` is a statement about *some* particular set `xc`. LP also orients this new hypothesis into a rewrite rule, `xc \union {} = xc -> false`.

The critical-pairs command directs LP to use the rewrite rule obtained from this new hypothesis, whose name `setTheoremsContraHyp.1` matches the pattern `*Hyp`, together with that obtained from the fact whose name matches `extensionality`, to enlarge its set of rewrite rules. LP does this by finding a term that can be rewritten in two different ways by the two rewrite rules and then asserting that these two terms must be equal. LP finds such a term by unifying the left side of one rewrite rule with a subterm of the left side of the other. Here, LP unifies the left side of the hypothesized rewrite rule with a subterm of the extensionality principle (by mapping `x` to `xc \union {}` and `y` to `xc`) to obtain the formula

```
\A e (e \in (xc \union {}) <=> e \in xc) => xc \union {} = xc
```

The extensionality axiom rewrites this formula to `true`, whereas the hypothesized rewrite rule rewrites it to

```
\A e (e \in (xc \union {}) <=> e \in xc) => false
```

Hence these two results must be equivalent:

```
\A e (e \in (xc \union {}) <=> e \in xc) => false <=> true
```

As in our first proof of the conjecture, this formula rewrites to `true => false <=> true`, which rewrites to `false` using LP's hardwired axioms. This inconsistency finishes the proof by contradiction.

The file set1.lp uses this technique to prove the second and third theorems about union.

1.2.9 Three theorems

about subset

The next three theorems in set1.lp establish some basic properties of the subset relation. They illustrate how LP's proof techniques combine to establish conjectures. The user types a few commands to select proof strategies (e.g., a proof by induction or a proof by cases). LP generates subgoals appropriate for the selected strategies and fills in most of the details automatically. Sometimes the user needs to tell LP to work a little harder near the end of a proof to fill in the remaining details.

```
set proof-methods normalization, =>
prove e \in x /\ x \subseteq y => e \in y by induction on x
      resume by case ec = e1c
        complete
qed
```

The set command directs LP to use an automatic proof technique whenever the user does not specify one explicitly. Here, it directs LP to try to prove conjectures first by rewriting, and then to assume the hypotheses of implications and try again. This setting for `proof-methods` is a good alternative to the standard setting, which consists of `normalization` alone.

A proof by induction

The user directs LP to begin the proof of the conjecture by induction on the variable `x` instead of by using one of the automatic proof methods. LP creates two subgoals for the proof by induction, a *basis subgoal* for the generator {} of sort `S` and an *induction subgoal* for the generator `insert`; it also formulates an *induction hypothesis* that can be used in the proof of the induction subgoal.

```
Creating subgoals for proof by structural induction on 'x'
Basis subgoal:
  Subgoal 1: e \in {} /\ {} \subseteq y => e \in y
Induction constant: xc
Induction hypothesis:
  setTheoremsInductHyp.1: e \in xc /\ xc \subseteq y => e \in y
Induction subgoal:
  Subgoal 2: e \in insert(e1, xc) /\ insert(e1, xc) \subseteq y => e \in y
```

LP proves the basis subgoal automatically by rewriting the term `e \in` {} to `false`, but it needs further guidance to prove the induction subgoal.

A proof technique for use with implications

LP uses the automatic => proof technique to create a subgoal for the induction step:

```
Creating subgoals for proof of =>
New constants: e1c, ec, yc
Hypothesis:
  setTheoremsImpliesHyp.1:
     (ec = e1c \/ ec \in xc) /\ e1c \in yc /\ xc \subseteq yc
Subgoal:
  ec \in yc
```

The hypothesis for this subgoal is the result of reducing the hypothesis of the implication in the induction subgoal, after replacing its variables by new constants.

A proof by cases

The user must issue two more commands to finish the proof. The first directs LP to divide the proof into two `cases`, depending on whether the formula `ec = e1c` is true or not.

```
Creating subgoals for proof by cases
Case hypotheses:
  setTheoremsCaseHyp.1.1: ec = e1c
  setTheoremsCaseHyp.1.2: ~(ec = e1c)
```

```
Same subgoal for all cases:
  ec \in yc
```

LP finishes the first case by using the case hypothesis to rewrite the subterm `ec` of the subgoal `ec \in yc` to `e1c`; then it rewrites `e1c \in yc` to `true` using the second conjunct of the implication hypothesis.

In the second case, LP uses the case hypothesis and its hardwired rules to rewrite the first conjunct of the implication hypothesis to `ec \in xc`, at which point it gets stuck. The complete command directs LP to use what it knows to finish the proof by deriving a few critical-pair equations. First, LP derives `xc \subseteq y => ec \in y` as critical pair between `ec \in xc` and the induction hypothesis . Then it obtains `ec \in yc` as a critical pair between this fact and the last conjunct of the implication hypothesis. This finishes the proof by cases, the proof of the implication for the induction step of the conjecture, and the proof of the conjecture itself.

Two more theorems about subset

Similar techniques can be used to prove two more basic theorems about subset. In the first, the automatic => proof technique introduces `xc = yc` as a subgoal. To finish the proof, the user proves a lemma (using the <=> technique) suggested by the principle of extensionality.

```
prove x \subseteq y /\ y \subseteq x => x = y
    prove e \in xc <=> e \in yc by <=>
        complete
        complete
    instantiate x by xc, y by yc in extensionality
qed
```

In the second, LP fills in all the details in a proof by induction without requiring further guidance from the user.

```
prove (x \union y) \subseteq z <=> x \subseteq z /\ y \subseteq z by induction on x
qed
```

1.2.10 An alternate induction rule

In order to prove a final theorem about subsets, `x \subseteq x`, it helps to use a different induction principle than the one we asserted as an axiom. We can prove that induction rule as follows:

```
prove sort S generated by {}, {__}, \union
    resume by induction
      set name lemma
      critical-pairs *GenHyp with *GenHyp
      critical-pairs *InductHyp with lemma
qed
```

LP formulates an appropriate subgoal for the proof of this conjecture, together with additional hypotheses to be used in the proof, using a new operator `isGenerated`:

```
Creating subgoals for proof of induction rule
Induction subgoal hypotheses:
  setTheoremsGenHyp.1: isGenerated({})
  setTheoremsGenHyp.2: isGenerated({e})
  setTheoremsGenHyp.3:
     isGenerated(s) /\ isGenerated(s1) => isGenerated(s \union s1)
Induction subgoal:
  isGenerated(s)
```

The user then directs LP to attempt to prove `isGenerated(s)` by induction (on `s`) using the asserted induction rule. LP proves the basis subgoal automatically using the hypothesis `isGenerated({})`. The user guides the proof of the induction subgoal by causing LP to compute critical pairs. The first critical-pairs command causes LP to derive

```
lemma.1: isGenerated(s) => isGenerated(insert(e, s))
```

as a critical pair between the second and third `isGenerated` hypotheses. The second critical-pairs command causes LP to derive the induction subgoal, `isGenerated(insert(e, sc))`, as a critical pair between this lemma and the induction hypothesis. This finishes the proof of the induction rule.

1.2.11 Two final theorems

about subset The new induction rule, `setTheorems.9`, helps us prove the following theorem about subsets:

```
prove x \subseteq (x \union y) by induction on x using setTheorems
qed
```

The prove command directs LP to use the induction rule just proved to formulate subgoals for the proof by induction using the generators {}, {__}, and `\union` instead of the generators {} and `insert`:

```
Creating subgoals for proof by structural induction on ‘x’
Basis subgoals:
  Subgoal 1: {} \subseteq ({} \union y)
  Subgoal 2: {e} \subseteq ({e} \union y)
Induction constants: xc, xc1
Induction hypotheses:
  setTheoremsInductHyp.1: xc \subseteq (xc \union y)
  setTheoremsInductHyp.2: xc1 \subseteq (xc1 \union y)
Induction subgoal:
  Subgoal 3: (xc \union xc1) \subseteq (xc1 \union xc \union y)
```

LP establishes all three subgoals automatically.

Finally, we prove the last theorem by directing LP to compute a critical pair between the theorem just established and an earlier theorem, `x \union {} = x`.

```
prove x \subseteq x
  critical-pairs setTheorems with setTheorems
qed
```

1.2.12 How to guide a proof

Here are some things to try when LP and/or you get stuck trying to prove a conjecture.

- Try a proof method based on the form of the conjecture. For example, try `resume by <=>` if the conjecture is a logical equivalence. Sometimes such proof methods are useful "no-brainers".
- Think about why you believe the conjecture.
- If you've forgotten where you are in a proof, type

```
display proof    to see the current subgoal
display *Hyp     to see the current hypotheses
display          to see all facts
history          to see how you got where you are
```

- Be skeptical: maybe the conjecture isn't a theorem.
- Try a proof by cases, either to simplify the current subgoal or to make some hypothesis more useful.
- Look for a useful lemma to prove.
- Try using the critical-pairs command to derive consequences from the hypotheses, for example, by typing `critical-pairs *Hyp with *`.
- Try alternative proof strategies.
  - The cancel command lets you back up.
  - The freeze and thaw commands let you save LP's state in a file and get it back again.
- Think again about why you still believe the conjecture.
- Try explaining why LP must be broken to a colleague. (The colleague does not need to understand anything about LP. He or she simply needs to appear to be listening attentively.)
- If you still believe that LP is broken, send e-mail to `larch@lcs.mit.edu`.

1.3 Notational conventions

See:

- Syntax descriptions
- Symbols
- Keywords

1.3.1 Syntax descriptions

The syntax of LP is described using equations of the form

```
<term> ::= <simpleId> | <simpleId> "(" <term>+, ")"
```

that are interpreted according to the following conventions to mean that a <term> is either a simple identifier or a simple identifier followed by a parenthesized list of terms separated by commas.

`chars` or "`chars`" A terminal symbol

<chars> A construct defined by a syntax equation

e f An *e* followed by (whitespace and) an *f*

e | *f* An *e* or an *f*

e ∼ *f* An *e* that is not an *f*

(*e***)** An *e* as a syntactic unit, as in (*e* | *f*) *e*. Without parentheses, *e* | *f e* is interpreted as *e* | (*f e*).

[*e*] An optional *e*

***e** or *e**, or *e**[,]** Zero or more *e*'s, separated respectively by nothing, commas, or optional commas.

***e*+ or *e*+, or *e*+[,]** One or more *e*'s, separated respectively by nothing, commas, or optional commas.

1.3.2 Symbols

LP divides its input into a sequence of *symbols* (also known as *tokens*) of the following kinds.

Simple identifiers

A <simpleId> is a string of letters, digits, apostrophes, and underscores. Exceptions: a <simpleId> cannot consist of an underscore or an apostrophe alone, and it cannot contain two adjacent underscores. Examples: `Nat`, `10`, `x'`, `a_1`.

A <number> is a <simpleId> consisting entirely of digits. See also keyword.

Operator identifiers

An <opId> is a string of *operator characters*, that is, characters chosen from among ” `!#$&*+-./<=>?@^|~` ”, or it is the special symbol ” `/\` ”. Exception: The symbol `->` is not an <opId>. Examples: `+`, `~=`, `<=>`.

Escaped identifiers

An <escapedId> consists of the *escape character* ” `\` ” alone or the escape character followed by a simple identifier, an operator character, an underscore, another escape character, or a punctuation mark. Examples: `\in`, `\-`, `\_`, `\\`, `\(`.

Punctuation marks

Each of the characters in ” `:,;()[]{}%` ” is a separate token, unless it is immediately preceded by the escape character. The *marker* symbol ” `__` ” is also treated as a punctuation mark.

Whitespace

A space, tab, or newline terminates the preceding token and is otherwise ignored. Likewise, all characters from an unescaped percent sign through the following newline are treated as whitespace. See the comment command.

Illegal characters

The characters ” `"` ” and ” `` ` `` ”, as well as all control characters other than whitespace, are reserved for future use. They should not appear in LP scripts except within comments.

Strings

In certain contexts, LP reads its input without dividing it into tokens. A <string> is an arbitrary sequence of characters other than newlines. A <blank-free-string> is a <string> that does not contain any whitespace.

1.3.3 Keywords

The keywords `by`, `else`, `if`, `in`, `then`, `when`, and `yield` cannot be used as identifiers for sorts, variables, or operators.

The keywords `to` and `with` cannot be used as names for facts, or as components of a name pattern.

LP reserves other words (`ac`, `commutative`, `depth`, `on`, `sort`, `using` and `well`) in certain contexts within facts and conjectures, but does not prohibit their use as identifiers.

1.4 Command summary

LP is a command-driven system. Commands can be entered in upper, lower, or mixed case. They, and some of their arguments, can be abbreviated by unambiguous prefixes of their hyphen-separated components. LP prompts users for any missing arguments that it requires to execute a command. The syntax of each command is illustrated and described more fully in the description for that command.

Commands for creating axioms and facts

`assert` <**assertion**>+; [;] Assert axioms

`declare operators` <**opdec**>+[,] Declare operators

`declare sorts` <**sort**>+, Declare sorts

`declare variables` <**vardec**>+[,] Declare variables

`make` <**fact-status**> (<**names**> | `conjecture`) Change the activity or immunity of facts or conjecture

Forward inference commands

`apply` <**names**> `to` <**names**> Apply the named deduction rules to the named facts

`complete` Run the Knuth-Bendix completion procedure

`critical-pairs` <**names**> `with` <**names**> Find critical pair equations between each rewrite rule in the first named set and each in the second

`fix` <**variable**> `as` <**term**> `in` <**names**> Eliminate one existential quantifier in the named facts, replacing the quantified variable by a term

`instantiate` (<**variable**> `by` <**term**>)+, `in` <**names**> Instantiate variables and/or eliminate universal quantifiers in the named facts, replacing the free and quantified variables by the terms

`normalize` <**names**> [`with` [`reversed` <names>]] Normalize the named facts using the (reversals of the) hardwired and named rewrite rules

`rewrite` <**names**> [`with` [`reversed` <names>]] Rewrite some subterm of each named fact using a hardwired or (the reversal of) a named rewrite rule

Backward inference commands

`apply` **<names>** `to conjecture` Attempt to prove the current conjecture using the named deduction rules

`cancel` [`all` | `lemma`] Cancel the current conjecture [and others]

`normalize conjecture` [`with` **<names>**] Normalize the current conjecture using all hardwired and named rewrite rules

`prove` **<conjecture>** [`by` **<proof-method>**] Attempt to prove the conjecture using <proof-method>

`qed` Check that all conjectures have been proved

`resume` [`by` **<proof-method>**] Resume work on the current conjecture using <proof-method>

`rewrite conjecture` [`with` [`reversed` <names>]] Rewrite some subterm of the current conjecture using some hardwired or named rewrite rule

<> Confirm the start of a subgoal in a proof

`[]` Confirm the conclusion of a step in proof

Commands for user interaction

`clear` Discard all information except global settings

`delete` **<names>** Delete the named facts

`define-class` **$<class>** **<names>** Define an abbreviation for <names>

`display` [**<information-type>** [<names>]] Print information about the named facts

`execute` **<file>** Execute commands from <file>`.lp`

`execute-silently` **<file>** Same as `execute`, but suppressing all output

`forget` [`pairs`] Discard information to save space

`freeze` **<file>** Save the state of LP in <file>`.lpfrz`

`help` **<topic>** Print help about the topics

`history` [**<number>** | `all`] Print a history of [the <number> most recent] commands

`pop-settings` Restore the values of local LP settings

`push-settings` Remember the values of local LP settings

`quit`, `q` Exit from LP

`set` Print the current values of all LP settings

`set` **<setting-name>** Print the current value of a setting and prompt for a new value

`set` **<setting-name>** **<setting-value>** Change the value of a setting

`show normal-form` **<term>** Display the reduction of a term to normal form

`show unifiers` **<term>**`,` **<term>** Display all unifiers of two terms

`statistics` [**<statistics-option>**] Print statistics on runtime, storage, and rule usage `stop` Stop execution of command files

`thaw` <**file**> Restore a frozen state from <file>`.lpfrz`

`unset` [<**setting**> | `all`] Reset setting to its default value

`version` Display information about the current version of LP

`write` <**file**> [<**names**>] Write the registry and the named facts to <file>`.lp`

`%` <**comment**> Record a comment in the log and/or script file

Commands to control ordering

`order` [<**names**>] Orient the named formulas into rewrite rules

`register` <**ordering-constraints**> Provide constraints for orienting formulas

`unorder` [<**names**>] Turn the named rewrite rules back into formulas

`unregister` <**ordering-information**> Cancel the constraints for orienting formulas

1.4.1 Command arguments

Many commands take one or more arguments. If you know how to use a command, you can type its arguments on the same line as the command. If you don't, LP will prompt you for the arguments. For example, you can type an entire command

```
set display-mode
```

in response to LP's command prompt, or you can type an abbreviation of the command without any arguments and await a further prompt:

```
LP1: set display

The current display-mode is 'unambiguous'.

New display-mode:
```

In response to this prompt, you can type

- the missing argument, for example, `=>, normalization`
- a carriage return, which aborts execution of the command
- a question mark, which provides help such as

```
Legal display modes:
  qualified    unambiguous  unqualified
```

Commands that require a possibly lengthy argument allow you to enter it on more than one line. To do this, don't type the argument on the command line; instead, type it on the following lines, terminating your input with a line consisting of two periods (`..`). For example,

```
LP1: declare sort Nat

LP2: declare operators
Please enter operator declarations, terminated with a '..' line, or '?' for
help:
0: -> Nat
s: Nat -> Nat
..
```

If you need help in the middle of typing a lengthy argument, type a question mark on a line by itself.

1.4.2 Abbreviations for commands and arguments

LP allows users to abbreviate a command with an unambiguous prefix of that command. For example, the `declare` command can be abbreviated to `dec`, but not to `de` because `de` is also a prefix of the `delete` command.

LP also allows users to abbreviate certain command arguments with unambiguous prefixes. When the argument can be a hyphenated word, it is enough to supply unambiguous prefixes of the hyphenated components. Examples:

```
Command                            Abbreviation
-------                            ------------
declare operators 0, 1: -> Nat     dec op 0, 1: -> Nat
display rewrite-rules              dis r-r
```

When an argument to a command can contain reserved words (such as `formulas`) and names of facts, abbreviations used for reserved words must not conflict with any name-prefix established by the set command. For example, `disp form` is a legal abbreviation for the command `display formulas` unless the user has previously entered the command `set name form`, in which case LP interprets `disp form` as a request to display all facts whose name begins with the prefix `form` rather than to display all facts that are formulas.

1.4.3 Interrupting command execution

Often one wants to interrupt an execute, order, critical-pairs, or complete command (e.g., to change the registry, to display the system, to alter the trace-level, or to freeze the system). This can be done by typing `^G` (hold down the control key and type `g`), which will return the user to the LP command level (it may take several seconds for this to happen). The complete and order commands can be issued again to resume their execution.

2 Logical syntax and semantics

LP is a proof assistant for multisorted first-order logic. Except for the fact that it provides a rich set of notations for functions, LP is based on the syntax and semantics for first-order logic found in many textbooks. For a description of this syntax and semantics, see:

- Conventions for describing LP's syntax
- Symbols
- Sorts
- Variables

- Operators
- Quantifiers
- Terms
- Formulas

2.1 Sorts

A *sort* is a symbol that represents a non-empty set of objects. In LP, distinct sorts represent disjoint sets of objects. Sorts can be *simple* or *compound*. At present, LP accords no special treatment to compound sorts.

Syntax

```
<sort>          ::= <simple-sort> | <compound-sort>
<simple-sort>   ::= <simpleId>
<compound-sort> ::= <simpleId> "[" <sort>+, "]"
```

Examples

```
Nat
Set[Nat]
Map[Set[A],A]
```

Usage

LP automatically declares the sort `Bool` and treats it as representing a set containing two objects, `true` and `false`. LP also automatically declares several logical operators for this sort.

All sorts other than `Bool` must be declared in a declare sorts command. Except for `Bool` and `bool`, which denote the same sort, case is significant in sort identifiers. Thus `Nat` and `nat` are different sorts.

Since distinct sorts represent disjoint sets of objects, users who want to consider one "sort" (e.g., `Nat`) as a subset of another (e.g., `Int`) must resort to one of two devices. They can define the larger as a sort and the smaller by means of a unary predicate (e.g., `isNat:Int->Bool`). Alternatively, they can define both as sorts and introduce a mapping from one into the other (e.g., `asInt:Nat->Int`).

2.2 Variables

A *variable* is a symbol that represents an arbitrary element of some sort. A variable can be an unqualified variable identifier or a variable identifier qualified by a sort.

Syntax

```
<variable>   ::= <variableId> [ : <sort> ]
<variableId> ::= <simpleId>
```

Examples

```
x
x:Int
committee: Set[Person]
```

Usage

All variables must be declared in a declare variables command. Case is significant in variable identifiers. Thus `x` and `X` are different variables.

The same <variableId> can be overloaded to name two different variables, that is, two variables with different sorts. It can also be used to name a variable and an operator, provided the operator is not a constant of the same sort. LP uses context to disambiguate overloaded identifiers.

At times (e.g., when computing critical pairs or when proving an induction rule), LP creates variables, the identifiers for which consist of the first letter of the sort of the variable, followed by digits if necessary to avoid conflicts with constants and with other variables in the same term or formula.

2.3 Operators

An *operator* is a symbol that represents a total function. The signature of the operator specifies the domain and range of the function. Operators can be used with functional, infix, prefix, postfix, bracketed, and conditional notations.

Syntax

```
<operator>     ::= <opForm> [ : <signature> ]
<opForm>       ::= <functionId> | <simpleOpForm> | <bracketedOp> | <ifOp>
<functionId>   ::= <simpleId>
<simpleOpForm> ::= [ __ ] <simpleOp> [ __ ] | [ __ ] . <simpleId>
<simpleOp>     ::= <opId> | <escapedId>
<bracketedOp>  ::= [ __ ] <openSym> __*, <closeSym> [ __ ]
<openSym>      ::= { | "[" | \( | <tt>\<</tt>
<closeSym>     ::= } | "]" | \) | \>
<ifOp>         ::= if __ then __ else __
```

Note: All markers (__) may be omitted from a <simpleOpForm> when there is exactly one declared operator (with the indicated signature) that can be formed by adding some number of markers to the <simpleOpForm>.

Examples

```
<functionId>     f        0              gcd
<simpleOpForm>   -__      <tt>__<=>__</tt>    __\in__    __\Post    __.first
<simpleOp>       -          <tt><=></tt>        \in        \Post
<bracketedOp>    __[__]   {}
```

Usage

LP automatically declares certain logical, equality, and conditional operators. All other operators must appear in a declare operators command. Case is significant in operator identifiers. Thus `f` and `F` are different operators, as are `\in` and `\In`.

Identifiers for most operators can be overloaded, that is, they can be used to represent operators with different signatures or with markers in different places. LP uses context to disambiguate overloaded identifiers.

The *arity* of an operator is the number of its arguments, that is, the number of sorts in its domain. A *unary* operator has arity 1, a *binary* operator has arity 2, and a constant has arity 0.

See also

- Functional, infix, prefix, postfix, bracketed, and conditional notations
- Declarations for operators
- The use of operators in terms
- Precedence for parsing operators in terms

2.3.1 Signatures

The *signature* of an operator specifies its domain and range sorts.

Syntax

```
<signature> ::= <domain> -> <range>
<domain>    ::= <sort>*,
<range>     ::= <sort>
```

Examples

<blockquote>

`->Nat` Signature for a nullary operator (constant) of sort `Nat`

`Bool->Bool` Signature for a unary operator from sort `Bool` to sort `Bool`

`Nat,Nat->Nat` Signature for a binary operator from sort `Nat` to sort `Nat`

`Element,Set->Set` Signature for a binary operator from sorts `Element` and `Set` to sort `Set`

</blockquote>

Usage

Signatures appear in declarations and qualifications for operators.

2.3.2 Constants

A *constant* is a 0-ary operator. An identifier for a constant can be either a <simpleId> (e.g., `0` and `c`) or a bracketed operator (e.g., {} and `[]`).

The same identifier can be overloaded to name two different constants, that is, two constants with different sorts. A <simpleId> can also be used to name a variable and a constant, provided the variable does not have the same sort as the constant.

When LP formulates hypotheses for use in proving subgoals in a proof, it generally replaces all free variables in the hypotheses by constants. LP creates identifiers for these constants by appending the letter `c` and, if necessary, further digits to obtain an identifier that is not already in use. Thus LP may replace the variable `x` by the constants `xc`, `xc1`, ...

2.3.3 Equality operators

LP automatically declares an equality operator `=` and an inequality operator `~=` with signature `S,S:->Bool` for each sort `S`. These operators have the standard interpretation: two objects of some sort `S` are equal if and only if they are identical.

Operationally, LP uses equational term-rewriting to reduce a term `t` to a normal form `t'` such that the formula `t = t'` is true. For this purpose, LP:

- treats the equality operator as commutative except when `S` is `Bool`, in which case LP treats it as a synonym for the associative-commutative operator `<=>` for logical equivalence.
- uses two hardwired rewrite rules for each sort `S`
  - `x:S = x:S -> true`
  - `x:S ~= y:S -> ~(x = y)`
- registers the operators `=` and `~=` as having multiset status for the purpose of orienting formulas into rewrite rules.

2.3.4 Logical operators

Syntax

LP automatically declares the sort `Bool` and the following logical operators.

`true` A constant denoting the value `true` of sort `Bool`.

`false` A constant denoting the value `false` of sort `Bool`.

`~` A prefix operator, pronounced "not", denoting negation.

`/\` An infix operator, pronounced "and", denoting conjunction.

`\/` An infix operator, pronounced "or", denoting disjunction.

`=>` An infix operator, pronounced "implies", denoting implication.

`<=>` An infix operator, pronounced "if and only if", denoting equivalence. The operator `<=>` is a synonym for the equality operator `=:Bool,Bool->Bool`; it differs from `=` only in that it binds less tightly during parsing.

Semantics

LP treats /\, \/, and <=> as associative-commutative operators.

The meaning of the logical operators is given by the following hardwired rewrite rules.

```
~true      -> false              true => p   -> p
~false     -> true               false => p  -> true
                                 p => true   -> true
p /\ true  -> p                  p => false  -> ~p
p /\ false -> false

p \/ true  -> true               p <=> false -> ~p
p \/ false -> p                  p <=> true  -> p
```

LP uses these rewrite rules, together with the following, to simplify terms containing logical operators.

```
~(~p)      -> p                  p => p      -> true
                                 p => ~p     -> ~p
p /\ p     -> p                  ~p => p     -> p
p /\ ~p    -> false
                                 p <=> p     -> true
p \/ p     -> p                  p <=> ~p    -> false
p \/ ~p    -> true               ~p <=> ~q   -> p <=> q
```

These rewrite rules are not sufficient to reduce all boolean tautologies to `true`. There are sets of rewrite rules that are complete in this sense, but they require exponential time and space, and they can expand rather than simplify expressions that do not reduce to `true` or `false`. Hence they are not hardwired into LP. Users wishing to use such a set can execute the file ~`lp/boolequiv.lp` or the file ~`lp/boolxor.lp`.

To assist in orienting formulas into rewrite rules, LP places `true` and `false` at the bottom of the registry, and it registers /\, \/, and <=> as having multiset status.

See also

- Conditional operators
- Equality operators
- Formulas
- Quantifiers
- The hardwired-usage setting
- Proof methods based on the logical connectives

2.3.5 Conditional operators

LP automatically declares a conditional operator `if_then_else_` with signature `Bool,S,S->S` for each sort `S`. The meaning of this operator is given by the following two hardwired rewrite rules.

```
if true then x else y  -> x
if false then x else y -> y
```

LP several additional hardwired rewrite rules to simplify terms containing conditional operators. It always uses the following two rules:

```
if x then y else y -> y
if ~x then y else z -> if x then z else y
```

Unless the hardwired-usage setting dictates otherwise, it also uses the following rewrite rules:

```
if x then true else y  -> x \/ y
if x then false else y -> ~x /\ y
if x then y else true  -> x => y
if x then y else false -> x /\ y
```

Likewise, unless the hardwired-usage setting dictates otherwise, LP uses the if-simplification metarule

```
if t1 then t2[t1] else t3[t1] -> if t1 then t2[true] else t3[false]
```

to reduce terms when `t1` occurs as a subterm of `t2` or `t3`; here `t2[true]` is the result of substituting `true` for every occurrence of `t1` as a subterm of `t2`, and `t3[false]` is defined similarly.

See also

- Proving formulas containing conditional operators using the if-method
- Left-to-right status assigned to `if__then__else__` and used when orienting formulas into rewrite rules

2.3.6 Bracketed operators

LP allows users to employ *bracketed notations* for operators. These notations consist of opening and closing symbols interspersed with arguments (in terms) or markers (in declarations). LP recognizes the following opening and closing symbols:

- opening symbols: `[`, {, \(, \<
- closing symbols: `]`, }, \), \>

For example, the declarations

```
declare operators
  {}:                         -> Set
  {__}:       Nat             -> Set
  __[__]:     Array, Nat      -> Nat
  __[__,__]: Matrix, Nat, Nat -> Nat
  [__]__:     Nat, Matrix     -> Array
  ..
```

enable the following notations:

- empty set: {}
- singleton set: {`x`}
- element of an array: `a[n]`
- element of a matrix: `m[1, 2]`
- slice of a matrix: `[1]m`

2.3.7 Functional operators

LP allows users to declare symbols for use in ordinary functional notations in terms. A *function identifier*, or <functionId>, is just a simple identifier, that is, a <simpleId>. It can be used as a constant (e.g., `0`) or as an operator that takes a parenthesized list of arguments (e.g., `gcd(x, y)`).

2.3.8 Infix, prefix, and postfix operators

LP allows users to declare symbols that can be used for infix, postfix, and prefix notations. Users can declare a simple operator (i.e., a <simpleOp>) for such use by decorating it with one or two markers (__) to indicate the location(s) of its arguments. For example, the declarations

```
declare operators
  __+__, __\mod__: Nat, Nat -> Nat   % infix operators
  -__:             Nat      -> Nat   % prefix operator
  __!:             Nat      -> Nat   % postfix operator
  ..
```

allow the use of infix (e.g., `(x + 1) \mod n`), prefix (e.g., `~p` and `-x`), and postfix (e.g., `3!`) notations. Users can also declare a *selection operator* for use in postfix notations. For example, the declaration

```
declare operator __.first: Queue -> Element
```

allows the use of postfix notations such as `q.first`. See operator.

2.4 Quantifiers

Terms and formulas in LP can contain existential and universal first-order quantifiers. Examples:

```
\A x \E y (x < y)
x < y => \E z (x < z /\ z < y)
```

The *existential quantifier* `\E x` is pronounced "there exists an `x`". The *universal quantifier* `\A x` is pronounced "for all `x`".

Syntax

```
<quantifier>    ::= <quantifierSym> <variable>
<quantifierSym> ::= \A | \E
```

Examples

```
\A x
\E i: Nat
```

Usage

All quantified variables must have been declared previously in a `declare variables` command.

The standard quantifier scope rules (from first-order logic) apply within terms and formulas. The *scope* of the leading quantifier in the terms `\A x t` and `\E x t` is the term `t`. An occurrence of a variable in a term is *bound* if it is in the scope of a quantifier over that variable; otherwise it is *free*. The free variables in a formula, rewrite rule, or deduction rule are understood to be quantified universally. LP suppresses the display of leading universal quantifiers.

LP uses the following hardwired rewrite rules to reduce terms containing quantifiers.

```
\A x t -> t
\E x t -> t
\E x \E x1 ... xn (x = t /\ P(x))           -> \E x1 ... \E xn P(t)
\A x \A x1 ... xn (~(x = t) \/ Q(x))        -> \A x1 ... xn Q(t)
\A x \A x1 ... xn (x = t /\ P(x) => Q(x))   -> \A x1 ... xn (P(t) => Q(t))
\A x \A x1 ... xn (P(x) => ~(x = t) \/ Q(x)) -> \A x1 ... xn (P(t) => Q(t))
```

Here `P(x)` and `Q(x)` are arbitrary terms, `t` is a term with no free occurrences of `x`, and `P(t)` and `Q(t)` are the results of substituting `t` for `x` in `P(x)` and `Q(x)` with appropriate changes of bound variables to prevent a quantifier in `P` or `Q` from capturing a variable in `t`.

The fix and instantiate commands, together with the generalization and specialization proof methods, enable users to eliminate quantifiers from facts and conjectures.

See also prenex form.

2.4.1 Prenex form

Every formula is logically equivalent to a *prenex formula*, that is, to a formula consisting of a string of quantifiers applied to a quantifier-free formula. For example, the following formulas are logically equivalent:

```
\E w \A x P(w, x) => \A y \E z P(z, y)
\A w \A y \E x \E z (P(w, x) => P(z, y))
\A w \A y \E x (P(w, x) => P(x, y))
```

Every quantifier in a formula can be classified as a *prenex-universal*, a *prenex-existential* quantifier, or neither depending on whether it corresponds to a universal (`\A`) quantifier, an existential (`\E`) quantifier, or more than one quantifier in a systematically derived logically equivalent prenex formula. The leading quantifiers in `\A x P` and `\E x P` are prenex-universal and prenex-existential quantifiers, respectively. If a quantifier is prenex-universal (prenex-existential) in `P`, then it is the same in `P /\ Q`, `P \/ Q`, `Q => P`, `\A y P`, and `\E y P`; it is prenex-existential (prenex-universal) in `~P` and `P => Q`; and it is neither in any other formula, e.g., in `P <=> Q`.

Formulas can be transformed systematically into prenex formulas using changes of bound variables and the following logical validities.

```
\A x P         <=>  P                            (x not free in P)
\E x P         <=>  P                            (x not free in P)
\A x ~P        <=>  ~\E x P
\E x ~P        <=>  ~\A x P
\A x (P /\ Q)  <=>  \A x P /\ \A x Q
\E x (P /\ Q)  <=>  \E x P /\ Q                  (x not free in Q)
\A x (P \/ Q)  <=>  \A x P \/ Q                  (x not free in Q)
\E x (P \/ Q)  <=>  \E x P \/ \E x Q
\A x (P => Q)  <=>  \E x P) => Q                 (x not free in Q)
\A x (P => Q)  <=>  P => \A x Q                  (x not free in P)
\E x (P => Q)  <=>  \A x P => \E x Q
F(\A x P)      <=>  (F(true) /\ \A x P)
                       \/ (F(false) /\ ~\A x P)
```

The instantiate and fix commands enable users to eliminate certain prenex-universal and prenex-existential quantifiers from facts. The generalization and specialization proof methods provide the corresponding functionality for conjectures.

2.4.2 Capturing variables

It is generally the case that, for any formula `P(x)` involving a variable `x` and any term `t`, the formula `P(t)` (i.e., the result of substituting `t` for each free occurrence of `x` in `P`) is a logical consequence of `P(x)`. However, this may not be the case if the substitution *captures* some free variable in `t`, that is, if `t` contains a free variable that becomes bound by some quantifier in `P`. For example, if `P(x)` is `\E y (x ~= y)`, then `P(y)` and `P(s(y))` are not logical consequences of `P(x)` because the free variable `y` in the terms `y` and `s(y)` is captured by the quantifier in `P`.

For this reason, LP automatically changes bound variables to avoid captures during rewriting, in response to the fix and instantiate commands, and in response to the generalization and specialization proof methods.

2.5 Terms

A *term* in multisorted first-order logic denotes an element of some sort. A term can be an *atomic* term (i.e., a variable or a constant), or it can be a *compound* term with one of the following forms:

- functional, e.g., `f(x)` or `min(x, y)`
- infix, e.g., `x + y`
- prefix, e.g., `-x`
- postfix, e.g, `x!` or `q.last`
- bracketed, e.g., `a[i]`
- conditional, e.g., `if x < 0 then -x else x`
- quantified, e.g., `\A x \E y (x < y)`

Syntax

```
<term>           ::= if <term> then <term> else <term>
                       | <subterm>
<subterm>        ::= <subterm> ( <simpleOp> <subterm> )+
                       | ( <simpleOp> | <quantifier> )* <simpleOp> <secondary>
                       | ( <simpleOp> | <quantifier> )* <quantifier> <primary>
                       | <secondary> <simpleOp>*
<secondary>      ::= <primary>
                       | [ <bracketPre> ] <bracketed> [ <bracketPost> ]
<primary>        ::= <primaryHead> <primaryTail>*
<primaryHead>    ::= <simpleId> [ "(" <term>+, ")" ]
                       | "(" <term> ")"
<primaryTail>    ::= "." <primaryHead> | <qualification>
<qualification> ::= ":" <sort>
<bracketPre>     ::= <primaryHead> | <primary> . <primaryHead>
<bracketed>      ::= <openSym> <term>*, <closeSym> [ <qualification> ]
<bracketPost>    ::= <primary> | . <primaryHead> <primaryTail>*
```

Restrictions: Terms must *type-check*, that is, all operators must be applied to arguments with sorts in the operator's signature. The following combinations of <simpleOp>s cannot appear in a <subterm> unless they are separated by <simpleOp>s that bind less tightly:

- more than one `=>`, `=`, or `~=`

- both /\ and \/
- both = and ~=
- two different user-declared <simpleOp>s

Usage

LP uses the following kinds of information to resolve potential ambiguities in terms:

- Operator precedence, when two <simpleOp>s appear in the same <subterm>
- Context, when overloaded operators appear in a term
- Explicit <qualification>s appearing in a term

2.5.1 Overloaded identifiers

LP allows users to overload identifiers for variables and operators. For example, LP allows the simple identifier `s` to be used both as a function `s:Nat->Nat` and as a variable `s:Set`, and it allows the operator identifier `-` to be used both as a prefix operator and as an infix operator. LP automatically overloads the predefined equality and conditional operators (`=`, `~=`, and `if_then_else_`), once for each declared sort. There are only two restrictions on overloading:

- The same identifier cannot be used to name a variable and a constant (i.e., a nullary operator) of the same sort.
- The logical, equality, and conditional operators (except for ~) cannot be overloaded with user-defined signatures.

LP disambiguates user input by using context to associate each identifier with a previous declaration. When context permits more than one association, users must supply additional annotations to resolve the ambiguity.

Disambiguating terms

Users can resolve potential ambiguities in terms by qualifying selected subterms with their sorts. For example, given the declarations

```
declare variables x, y, z: Bool, x, y, w: Nat
```

the terms `x = z` and `x = w` are unambiguous, but the term `x = y` is ambiguous. It can be disambiguated in several ways, for example, as `x:Nat = y` or as `x = y:Bool`. Given the additional declarations

```
declare operators f:Nat->Nat, f:Nat->Bool, f:Bool->Bool
```

the term `f(z)` is unambiguous, but the term `f(w)` needs to be disambiguated as either `f(w):Nat` or `f(w):Bool`, and the term `f(x)` needs to be disambiguated as one of `f(x:Bool)`, `f(x:Nat):Nat`, or `f(x:Nat):Bool`.

Another potential ambiguity in terms arises from the treatment LP accords to the period symbol (`.`). For example, given the declarations

```
declare operators
  1, min:                         -> Nat
  a:                              -> Array
  __.__:              Array, Nat -> Nat
  __ .size, __ .min: Array      -> Nat
  ..
```

the terms `a.size` and `a.1` are unambiguous, but `a.min` could represent an application of either the infix operator `.` or the postfix operator `.min`. In such cases, LP disambiguates `a.min` as containing a postfix operator; users who want `.` to refer to the infix operator can write `a.(min)`.

Disambiguating operators outside of terms

Users can resolve an ambiguous operator identifier by qualifying it with its signature or by decorating it with one or two markers (__) to indicate whether it is an infix, postfix, or prefix operator.

See also

- The display-mode setting
- The write-mode setting

2.5.2 Precedence of operators

Parsing

When parsing a term, LP binds some operators more tightly than others, thereby reducing the need for parentheses. The operators in the following list bind increasingly tightly:

- `if __ then __ else __`
- <=>
- =>
- /\ and \/
- = and ~=
- All infix, postfix, and prefix operators not otherwise mentioned in this list
- Postfix selection operators (of the form .<simpleId>) and `.` as an infix operator

Parentheses are required elsewhere to specify associativity, except that terms such as `t1 + t2 + ... + tn`, which involve a sequence of terms separated by a single infix identifier (other than =>, =, and ~=), do not need parentheses and are associated to the left.

Quantifiers bind more tightly than all operators.

Examples

```
Unparenthesized version    Interpretation
----------------------     --------------
a < b + c                  Error
p /\ q \/ r                Error
p => q => r                Error
x - y - z                  (x - y) - z
a = b + c => b < s(a)      (a = (b + c)) => (b < s(a))
a.b.c!                     ((a.b).c)!
~p /\ ~x.pre               (~p) /\ (~(x.pre))
\E x (x < c) => c > 0      (\E x (x < c)) => (c > 0)
\A x \E y x < y            (\A x \E y x) < y
```

Ordering

When orienting formulas into rewrite rules, LP sometimes uses a partial ordering on operators that is known in the literature as a *precedence relation*. See register.

2.6 Formulas

A *formula* is boolean-valued term.

Syntax

```
<formula> ::= <term>
```

Restriction: the sort of the <term> must be `Bool`.

Examples

```
x + s(y) = s(x + y)
x | y <=> \E z (y = x * z)
x < y \/ x = y \/ y < x
x < y => \E z (x < z /\ z < y)
```

Usage

Users can assert formulas, prove them, or derive them from other formulas by forward inference.

Operationally, LP uses formulas by orienting them (if possible) into a terminating set of rewrite rules. LP also automatically reduces all nonimmune formulas to normal form.

LP automatically rewrites formulas of the form `~p` to `p = false` and formulas of the form `~p = false` to `p`. Furthermore, it uses the following hardwired deduction rules to derive new formulas from existing formulas and rewrite rules.

```
lp_and_is_true:  when p /\ q    yield  p, q
lp_or_is_false:  when ~(p \/ q) yield  ~p, ~q
```

LP uses the names of these deduction rules when it reports their application, but they cannot be entered by the user when naming objects.

To display the formulas that LP has not converted into rewrite rules, type **`display formulas`** (or **`dis fo`** for short).

2.6.1 Equations

An *equation* is a formula that consists of a pair of terms separated by an equality operator.

Syntax

```
<equation> ::= <term> (= | <tt><=></tt>) <term>
```

Restriction: The two <term>s in an equation must have the same sort. This sort must be `Bool` if the equality operator is <=>.

Examples

```
x + 0 = x
x <= y <=> x < y \/ x = y
x | y <=> \E z (y = x*z)
```

Usage

LP treats equations in the same manner as it treats formulas. Indeed, any formula is logically equivalent to an equation: `F` is logically equivalent to `F <=> true`.

3 Operational syntax and semantics

The basis for each proof in LP is a logical system that contains a collection of facts (axioms, hypotheses, and previously proved theorems) available for use in that proof. There is a separate logical system, known as a proof context, for the proof of each conjecture and subgoal.

The facts in a logical system have both semantic content (expressed by formulas in first-order logic) and operational content. For more information about this operational content, see:

- Formulas
- Inconsistencies
- Rewrite rules
- Operator theories
- Induction rules
- Deduction rules

3.1 Inconsistencies

An *inconsistency* is a formula that is logically false.

LP automatically recognizes the formula `x ~= t`, where `t` is a term not involving the variable `x`, as inconsistent. Thus, LP rules out empty sorts. It also recognizes the formulas `false` and `b = t`, where `t` is a term not involving the boolean variable `b`, as inconsistent. Thus, the boolean sort contains two distinct values.

If an inconsistency arises during a proof by contradiction, that proof succeeds. If it arises during a proof by cases, the current case is deemed impossible.

3.2 Rewrite rules

A *rewrite rule* is an operational form for a formula. LP uses rewrite rules to rewrite terms to logically equivalent terms.

Syntax

```
<rewrite-rule> ::= <term> -> <term>
```

Restrictions: The two <term>s in a rewrite rule must have the same sort. The left side of the rule must not be a variable, the logical constant `true`, or the logical constant `false`. Every free variable in the right side must also occur free in left side.

Examples

```
x + 0 -> x
x <= y -> x < y \/ x = y
x | y -> \E z (y = x*z)
0 < 1 -> true
```

Usage

Users cannot assert or prove rewrite rules directly. Instead, they must assert or prove formulas, which LP then orients into rewrite rules. The logical content of a term is the equation that results from replacing `->` by `=`.

LP maintains the invariant that all active rewrite rules have been applied to all nonimmune formulas, rewrite rules, and deduction rules in the system.

Some rewrite rules are hardwired into LP to define properties of the logical connectives, the conditional and equality operators, and the quantifiers.

The restriction that the left side of a rewrite rule not be a variable prevents "rules" like `x -> f(x)` from leading to a nonterminating rewriting sequence such as `x -> f(x) -> f(f(x)) -> ...`. The restriction that it not be `true` or `false` preserves the meaning of those logical constants. The restriction that the right side not introduce a free variable is more technical. It prevents logically equivalent "rules" such as `f(x) -> f(y)` and `f(u) -> f(v)` from producing different results when applied to terms like `y + f(x)`.

3.3 Operator theories

An *operator theory* is logically equivalent to a set of equations involving a single operator. At present, LP supports two operator theories:

- the *commutative* theory, which is axiomatized by the commutative law `x + y = y + x`
- the *associative-commutative* (*ac*) theory, which is axiomatized by the commutative law together with the associative law `x + (y + z) = (x + y) + z`.

LP uses operator theories to circumvent problems with nonterminating rewriting systems. Because the commutative law cannot be oriented into a terminating rewrite rule, LP uses equational term-rewriting to match and unify terms modulo the ac and commutative operator theories.

Syntax

```
<operator-theory> ::= (ac | commutative) <operator>
```

Examples

```
ac +
commutative gcd
```

Usage

Users can assert or prove operator theories.

LP hardwires the logical connectives `/\`, `\/`, and `<=>` as associative-commutative operators. It hardwires the equality operator `=:S,S->Bool` as a commutative operator when the sort `S` is not `Bool`.

The fact `ac op` normalizes the fact `commutative op` to `true`.

To display the operator theories in LP's logical system, type `display operator-theories` (or `disp o-t` for short).

3.4 Induction rules

An *induction rule* is logically equivalent to an infinite set of formulas, which justify the use of proofs by induction.

Syntax

```
<induction-rule> ::= sort <sort> generated [ freely ] by <operator>+,
                   | well founded <operator>
```

Examples

```
sort Nat generated freely by 0, s
sort Set generated by {}, insert
sort Set generated by {}, {__}, \U
well founded <
```

Usage

Users can assert or prove induction rules.

Assertions like `sort Nat generated by 0, s` specify sets of generators for use in proofs by structural induction. The listed operators (e.g, `0` and `s`) must have the named sort (e.g., `Nat`) as their range. If none of the domain sorts of an operator is the same as its range sort, the operator is a *basis generator* (e.g., `0`); otherwise, it is an *inductive generator* (e.g, `s`). Structural induction rules are logically equivalent to infinite sets of first-order formulas of the form

```
P(0) /\ \A x (P(x) => P(s(x))) => \A x P(x)
```

where `P` is an arbitrary first-order formula.

The use of `freely` supplements a structural induction rule with a set of formulas asserting that the named operators are one-to-one and have disjoint ranges. For example, `sort Nat generated freely by 0, s` gives rise to the formulas `s(x) = s(y) <=> x = y` and `0 ~= s(x)`.

Assertions like `well founded <` specify a binary relation for use in proofs by well-founded induction. The listed operator must have a signature of the form `S,S->Bool` for some sort `S`. Well-founded induction rules are logically equivalent to infinite sets of first-order formulas of the form

```
\A x (\A y (y < x => P(y)) => P(x)) => \A x P(x)
```

where `P` is an arbitrary first-order formula.

To display the induction rules that LP currently has available for use, type `display induction-rules` (or `disp i-r` for short).

3.5 Deduction rules

A *deduction rule* is logically equivalent to a single formula in multisorted first-order logic. That formula has the form of a logical implication. LP uses deduction rules to derive consequences from other formulas and rewrite rules.

Syntax

```
<deduction-rule> ::= when <hypothesis>+, yield <conclusion>+,
<hypothesis>     ::= <formula>
<conclusion>     ::= <formula>
```

Examples

```
when x < y, y < z yield x < z
when p /\ q yield p, q
when \A e (e \in s1 <=> e \in s2) yield s1 = s2
```

Usage

Users can assert or prove deduction rules. A deduction rule is logically equivalent to the formula obtained by having the conjunction of its hypotheses imply the conjunction of its conclusions. Thus the first example is logically equivalent to the formula `x < y /\ y < z => x < z`.

A deduction rule can be *applied to* a formula or rewrite rule if there is a substitution that matches the first hypothesis of the deduction rule to the formula or rewrite rule. The result of applying a deduction rule with one hypothesis is the set of formulas obtained by applying the substitution(s) that matched its hypothesis to each of its conclusions. For example, applying the second example to `1 < f(x) /\ f(x) < 2` yields two formulas, `1 < f(x)` and `f(x) < 2`. Likewise, applying the third example to `x \in s <=> x \in (s \U s)` yields the formula `s = s \U s`.

The result of applying a deduction rule with more than one hypothesis is the set of deduction rules obtained by deleting its first hypothesis (i.e., the one that was matched) and applying the substitution(s) that matched this hypothesis to the remainder of the deduction rule. For example, applying the first example to the formula `a < b` yields the deduction rule

```
when b < z yield a < z
```

On the other hand, applying the logically equivalent deduction rule

```
when y < z, x < y yield x < z
```

to the same formula yields a different result, namely,

```
when x < a yield x < b
```

When applying deduction rules, LP produces the strongest valid conclusions by changing, when appropriate, variables in the conclusions that do not occur freely in the hypotheses. For example, applying the deduction rule

```
when P(x) yield Q(x, y)
```

to the formula `P(f(y))` yields the result `Q(f(y), y1)` and not the weaker result `Q(f(y), y)`.

The results of applying a deduction rule are given the default activity and immunity, as established by the set command.

LP maintains the invariant that all active deduction rules have been applied to all nonimmune formulas and rewrite rules in the system.

To display the deduction rules that LP currently has available for use, type `display deduction-rules` (or `disp d-r` for short).

See formulas for a list of deduction rules that are hardwired into LP. See also partitioned-by.

3.6 Partitioned-by rules

The *partitioned-by* construct provides a convenient syntax for expressing the fact that two objects of a given sort are the same if they cannot be distinguished by an operator in a given list. Operationally, a <partitioned-by> is equivalent to a deduction rule.

Syntax

```
<partitioned-by> ::= sort <sort> partitioned by <operator>+,
```

Restriction: Each of the <operator>s must have <sort> in its domain.

Examples

```
sort Set partitioned by \in
sort Stack partitioned by isEmpty, top, pop
```

Usage

Users can assert or prove a <partitioned-by>, which LP automatically translates into a deduction rule. For example, LP translates the examples into the deduction rules

```
when \A e (e \in s = e \in s1) yield s = s1
when isEmpty(s) = isEmpty(s1), top(s) = top(s1), pop(s) = pop(s1)
yield s = s1
```

In general, LP translates a statement like

```
sort E partitioned by op1:E->R, op2:E,A->R, op3:E,E,A->R
```

into to the deduction rule

```
when op1(e1) = op1(e2),
     \A a (op2(e1, a) = op2(e2, a)),
     \A a \A e3 (op3(e1, e3, a) = op3(e2, e3, a)),
     \A a \A e3 (op3(e3, e1, a) = op3(e3, e2, a))
yield e1 = e2
```

which uses auxiliary variables `e1`, `e2`, and `e3` of sort `E` and `a` and `a1` of sort `A`.

3.7 Naming

See

- Assigning a <name> to a fact
- Using <names> in commands
- Name classes

3.7.1 Assigning names to facts

Each fact in LP's logical system has a name consisting of an alphanumeric prefix followed by a series of numeric suffixes separated by periods.

Syntax

```
<name>      ::= <simpleId> <extension>*
<extension> ::= . <number>
```

Examples

```
arith.1
set.2.3
```

Usage

LP assigns new names to facts in response to the assert and fix commands, to critical pair equations in response to the critical-pairs and complete commands, and to conjectures in response to the prove command. These names have a single numeric <extension>, whose value increases each time a new name is required.

Users can specify the name prefix for a fact in the assert command, or for a conjecture in the prove command, by writing :<simpleId>: before the fact. Otherwise the value of the name-prefix setting governs the <simpleId> used as a prefix for new names.

Normalization, ordering, unordering, and proofs preserve the names of formulas, rewrite rules, and deduction rules. LP assigns subnames (i.e., names with an additional <extension>) to instantiations of formulas, rewrite rules, and deduction rules, to the results of applying deduction rules, and to hypotheses introduced during the proof of a conjecture. The names of the results of applying a deduction rule to a formula extend the name of the deduction rule if the deduction rule has more than one hypothesis; otherwise they extend the name of the formula.

One fact is called a *descendant* of another (and the latter is called an *ancestor* of the former) if the name of the first extends, by zero or more <extension>s, the name of the second. A *proper* ancestor (or descendant) of a fact is an ancestor (or descendant) with a different name.

The names of the hardwired deduction rules for formulas begin with `lp_`. The hardwired rewrite rules for the logical connectives, the conditional and equality operators, and the quantifiers do not have names, nor do conjectures introduced as subgoals in proofs.

See also

- Using <names> in commands
- Name classes

3.7.2 Naming facts in commands

Many LP commands require one or two sets of facts as arguments, which users describe in terms of the names and kinds of facts they contain.

Syntax

```
<names>          ::= <name-primary>+,
                       | <name-primary> ("/" <name-primary>)+
                       | <name-primary> ("~" <name-primary>)+
<name-primary>   ::= <name-pattern> | <class> | "(" [ <names> ] ")"
<name-pattern>   ::= <prefix-pattern> [ . ]
                       | <prefix-pattern> <extension>+ [ : <last> ] [ ! ]
<prefix-pattern> ::= ( <simpleId> | "*" )+
<last>           ::= <number> | last
```

Examples

```
arith, set.2:last
*Hyp
* ~ (nat, set)
```

Usage

Subsets of LP's logical system are described by expressions (called <names>) built up from primitive descriptions using operators for union (","), intersection ("/"), and difference ("~"). Parentheses specify the associativity of these infix operators. The empty set is named by `()`, and the set of all facts in LP's logical system is named by `*`.

Primitive <names> include <name>s (e.g., `set.2`) and <name-pattern>s of the following forms, where `N` is a <name> that may have asterisks embedded in its alphanumeric prefix (e.g., `*` and `*Hyp.1`), and where `m` and `n` are positive integers.

```
Pattern    Facts denoted by pattern
-------    ------------------------
N!         those with names that can be obtained from N by replacing the
           asterisks in N by alphanumeric characters
N          those denoted by N! and all their descendents
N.m:n!     those denoted by N.m!, N.m+1!, ..., N.n!
N.m:n      those denoted by N.m, N.m+1, ..., N.n
N.m:last!  those denoted N.m!, N.m+1!, ...
N.m:last   those denoted N.m, N.m+1, ...
```

See also <class>.

3.7.3 Name classes

A *class* is an expression that, when evaluated, produces a list of <name>s of facts.

Syntax

```
<class>          ::= <class-name> | <class-constant>
                       | <class-function> "(" <names> ")"
                       | contains-operator "(" <operator> ")"
                       | contains-variable "(" <variable> ")"
                       | copy "(" <class-name> ")"
<class-name>     ::= $ <simpleId>
<class-constant> ::= deduction-rules | formulas | induction-rules
                      | operator-theories | rewrite-rules | active | passive
                      | immune | nonimmune | ancestor-immune
<class-function> ::= ancestors | proper-ancestors | descendants
                      | proper-descendants | eval
```

Examples

```
$facts
contains-operator(+)
eval(* ~ $old)
```

Usage

The define-class command defines a <class-name> as an abbreviation for some <names>. Each <class>, other than ones beginning with `copy` or `eval`, also serves as an abbreviation for the names of facts matching a certain criterion. For example, `contains-operator(+)` describes the collection of all facts containing the operator `+`.

A <class-name> (or <class>) appearing in an argument to a command other than define-class is replaced by its definition (or evaluated) when the command is executed. Evaluation in the define-class command can be forced using one of the following operators:

`copy($class)` Replaces the <class-name> `$class` by its current definition

`eval(exp)` Replaces the <names> `exp` by an explicit list of the names of all facts in LP's logical system that match `exp`

See also

- Assigning a <name> to a fact
- Using <names> in commands

3.8 The assert command

The *assert command* adds axioms to LP's logical system.

Syntax

```
<assert-command> ::= assert <assertion>;+ [;]
<assertion>      ::= [:<simpleId>:] <fact>
<fact>           ::= <formula> | <deduction-rule> | <induction-rule>
                       | <operator-theory> | <partitioned-by>
```

Examples

```
assert
  e1 \in insert(e2, s) <=> e1 = e2 \/ e1 \in s;
  when \A e (e \in s1 <=> e \in s2) yield s1 = s2;
  Set generated by {}, insert;
  ac \U;
  Set partitioned by \in
  ..
```

Usage

LP adds the asserted facts to its logical system. It assigns a name to each of these facts using the current name-prefix setting, unless an assertion begins with :<simpleId>:, in which case LP uses that identifier as the name-prefix for that assertion.

LP takes certain default actions when it adds assertions to its logical system.

- It translates <partitioned-by>s into deduction rules.
- It normalizes asserted formulas and deduction rules (unless the immunity setting dictates otherwise).
- It attempts to orient asserted formulas into terminating rewrite rules (unless the automatic-ordering setting dictates otherwise).
- It uses new rewrite rules to normalize the other formulas and deduction rules (unless the activity setting dictates otherwise).
- It uses new deduction rules to deduce consequences from formulas and rewrite rules (again, unless the activity setting dictates otherwise).

- It registers operators asserted to be `ac` or `commutative` as having multiset status. If any of these operators had a non-ac (or noncommutative) polynomial interpretation, LP prints a warning and gives the operator a default ac (or commutative) polynomial interpretation.
- It turns all rewrite rules containing an operator asserted to be `ac` or `commutative` back into formulas, and it reflattens all facts containing these operators. This process can change these facts or even reduce them to `true`, as is the case for the commutativity equation.

3.9 The declare command

Except for certain predefined logical, equality, and conditional operators, all identifiers for sorts, variables, and operators must be declared in a *declare command* before any other use.

Syntax

```
<declare-command> ::= declare ( <sortDecs> | <varDecs> | <opDecs> )
<sortDecs>        ::= sorts <sort>+,
<varDecs>         ::= variables <varDec>+[,]
<varDec>          ::= <variableId>+, : <sort>
<opDecs>          ::= operators <opDec>+[,]
<opDec>           ::= <opForm>+, : <signature>
```

Restrictions: There must be at least one marker in each <opForm> that contains a <simpleOpForm>. The number of marker symbols in any <opForm> not containing a <functionId> must equal the number of sorts in the domain of the operator's signature. A period (`.`) can be declared as an infix operator, but not as a prefix or a postfix operator. The predefined operators cannot be overloaded with user-defined signatures.

Notes: The words `sorts`, `operators`, and `variables` can be abbreviated by any prefix, as well as by `ops` and `vars`.

Examples

```
declare sorts Nat, Set
declare variables i, j: Nat, s: Set
declare operators
  0:                      -> Nat
  s, __!:       Nat      -> Nat
  __+__, __*__: Nat, Nat -> Nat
  {}:                     -> Set
  {__}:         Nat      -> Set
  __\U__:       Set, Set -> Set
  ~__:          Set      -> Set
  ..
```

Usage

The `declare sorts` command introduces identifiers for use as sorts.

The `declare variables` command introduces variables with the indicated sorts.

The `declare operators` commands introduces operators with the indicated signatures. Marker symbols (__) in an <opForm> indicate where the arguments of the operator are placed when it is used in a term.

LP at times creates variables and operators in response to user commands. To avoid problems, it is a good idea for users to declare the identifiers they need before issuing commands (such as `prove`) that may cause LP to appropriate an identifier for its own use (e.g., by creating a variable `s1` of sort `Set` and thereby preventing the user from later using `s1` as a constant of sort `Set`).

The display symbols command produces a list of all declared identifiers.

3.10 The make command

The *make command* changes the activity or immunity of a collection of facts or of the current conjecture.

Syntax

```
<make_command> ::= make <fact-status> ( <names> | conjecture )
<fact-status>  ::= active | passive | immune | nonimmune | ancestor-immune
```

Note: The <fact-status> can be abbreviated.

Examples

```
make inactive rewrite-rules
make immune conjecture
```

Usage

LP automatically uses any rewrite rules made active by the `make` command to normalize terms appearing in the current conjecture or in nonimmune facts in LP's logical system. It also automatically applies any deduction rules made active by the `make` command.

LP automatically normalizes any rewrite rules and deduction rules made nonimmune by the `make` command, and it applies all active deduction rules to these deimmunized rules.

See the activity and immunity settings.

3.11 The order command

The *order command* directs LP to attempt to orient formulas into rewrite rules.

Syntax

```
<order-command> ::= order [ <names> ]
```

Examples

```
order
order nat
```

Usage

When the automatic-ordering setting is `on`, LP attempts to orient formulas into rewrite rules automatically. When it is `off`, LP orients formulas only in response to an explicit `order` or complete command. If no <names> are specified in an `order` command, LP attempts to orient all formulas in the system. If <names> are specified, LP attempts to orient only the named formulas (including any new formulas that LP generates during the ordering process, for example, as a result of applying a deduction rule to a newly reduced fact).

When a formula is an equation `t1 = t2` or `t1 <=> t2`, LP uses the current ordering-method, if possible, to orient it into the rewrite rule `t1 -> t2` or into the rewrite rule `t2 -> t1`. When a formula `f` is not an equation (i.e., when its principal operator is neither `=` nor `<=>`), LP orients it into the rewrite rule `f -> true` (or into the rewrite rule `f1 -> false`, if `f` has the form `~f1`).

If the current ordering method is a registered or polynomial ordering, the `order` command also attempts to orient the formulas into a provably terminating set of rewrite rules. If the current ordering method is a brute-force ordering, the `order` command may orient the formulas into a nonterminating set of rewrite rules.

Users can interrupt and resume the operation of the `order` command.

3.11.1 Brute force orderings

LP's *brute-force* ordering procedures give users complete control over the direction in which formulas are oriented into rewrite rules, but provide no guarantees about termination.

The `left-to-right` ordering

This ordering method causes LP to orient equations into rewrite rules from left to right, provided the results are valid rewrite rules. Equations that cannot be oriented from left to right are left unoriented.

The `either-way` ordering

This ordering method causes LP to orient equations into rewrite rules from left to right, provided the results are valid rewrite rules. If an equation cannot be oriented from left to right, but can be from right to left, the method causes it to be oriented in that direction; otherwise it is left unoriented.

The `manual` ordering

When the automatic-registry setting is `off`, this ordering method causes LP to interact with the user to select an orientation for each equation. When the setting is `on`, this ordering method does not orient any formulas into rewrite rules.

3.11.2 Registered orderings

LP provides two *registered orderings*, the dsmpos and noeq-dsmpos orderings, which are based on LP-suggested partial orderings of operators and which guarantee termination of the set of rewrite rules when no associative-commutative operators are present. Most users rely on these orderings to orient all formulas. In striking contrast to the brute-force orderings, they hardly ever cause difficulties by producing a nonterminating set of rewrite rules.

Registered orderings use information in a *registry* to orient formulas into rewrite rules. There are two kinds of information in a registry: height information and status information.

When the current registry does not contain enough information to orient a formula, LP will generate minimal sets of extensions to the registry, called *suggestions*, that permit the formula to be oriented. If the automatic-registry setting is `on`, LP picks one of these extensions to orient a formula without user interaction; it does not try all extensions. If the setting is `off`, LP will interact with the user, who must pick the desired extension.

3.11.3 The dsmpos ordering

The *dsmpos ordering* is a registered ordering used to orient formulas into rewrite rules. It defines a well founded partial order on terms from a partial order on operators, given by height constraints, and from status constraints. When no associative-commutative operators or quantifiers are present, the rewrite relation of the set of rewrite rules produced by the dsmpos ordering is embedded in this well founded partial order on terms and hence is guaranteed to terminate.

The `dsmpos` ordering computes complete sets of minimal extensions to LP's registry when necessary to orient a formula. The `noeq-dsmpos` is the same ordering, except that it does not suggest assigning equal heights to two operators; as a result, it is faster, but less powerful than `dsmpos`.

Let s and t be two terms, with s = *f(s1, ..., sm)* and t = *g(t1, ..., tn)*. Then $s >= t$ in the `dsmpos` ordering iff

- $si >= t$ for some i, or
- $f > g$ (i.e., f is higher than g in the registry) and $s > ti$ for all i, or
- either f and g have the same height in the registry, or $f >= g$ and $s > ti$ for all i, and in either case
  - if both f and g can have multiset status, then the multiset $\{s1, \ldots, sm\}$ is greater than or equal to the multiset $\{t1, \ldots, tn\}$, or
  - if both f and g can have lexicographic status, then $s > ti$ for all i and the sequence $< s1, \ldots, sm >$ is lexicographically greater than or equal to the sequence $< t1, \ldots, tn >$.

Here *M1* is greater than or equal to *M2* as a multiset iff *M2* has the same elements, though possibly with different multiplicities, as *(M1 - X) U Y*, where X and Y are multisets such that for any y in Y there is an x in X with $x > y$.

The `dsmpos` ordering is based on the recursive path ordering with status due to Dershowitz, Kamin, and Levy. The definition of the ordering >= is due to Forgaard. The generation of suggestions is due to Detlefs and Forgaard.

LP uses the following modification of the `dsmpos` ordering to orient equations that contain quantifiers. It replaces each quantifier (over a sort `S`) in an equation by a pseudo-operator with signature `S, Bool -> Bool`, and it replaces each bound variable by the constant `true`; in this way, it converts each subterm of the form `\E x P(x)` or `\A x P(x)` in an equation into a subterm `\E(true, P(true))` or `\A(true, P(true))`. (In general, the resulting term does not sort check.) Then LP tries to orient the transformed equation with the aid of registry suggestions for the pseudo-operators. It it succeeds, it orients the original equation in the same direction as its transformation.

3.11.4 Polynomial orderings

The polynomial ordering can be used to prove the termination of sets of rewrite rules involving associative-commutative operators. Because it 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.

Syntax

```
<polynomial-constraint> ::= polynomials <operator> <polynomial>*[,]
<polynomial>            ::= <polynomial-term> ( "+" <polynomial-term> )*
<polynomial-term>       ::= <polynomial-factor> ( "*" <polynomial-factor> )*
<polynomial-factor>     ::= <polynomial-primary> [ "^" <number> ]
<polynomial-primary>    ::= <variable> | <number> | "(" <polynomial> ")"
```

Examples

```
polynomials + x + y + 1, x + 2
```

Usage

The *polynomial ordering* is based on user-supplied interpretations of operators by sequences of polynomials. 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 principal 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 values of its variables).

The command `set ordering polynomial` *n* sets the current ordering-method to a polynomial ordering based on sequences of length *n*; the default value of `n` is 1.

The command `register polynomial` *op p1*, `...`, *pn* assigns the sequence *p1*, ..., *pn* of polynomials as the polynomial interpretation of the operator *op*. If no polynomials are specified, LP prompts the user to enter them on the following lines. The polynomials are entered like standard LP terms, using the binary operators `+`, `*`, \and fq{^} (for exponentiation), the variables in the prompt, 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.

Each operator has a default interpretation. Suggestions for assigning polynomials:

```
(1) f nullary                           I[f]   = 2
(2) f(x1,...,xn) -> t  [f not in t]     I[f]   = I[t] + 1
(3) h(f(t1,...,tn)) ->                  I[h]   = a*(x^i) with i > 1
       f(h(t1),...,h(tn))               I[f]   = x1 + ... + xn
(4) f associative                       I.1[f] = (a*x*y) + x with a > 0
    f(f(x,y),z) -> f(x,f(y,z))          I.2[f] = (a*(x^i)) + y with a, i > 0
(5) f associative                       I.1[f] = (a*x*y) + y with a > 0
    f(x,f(y,z)) -> f(f(x,y),z)          I.2[f] = x + (a*(y^i)) with a, i > 0
(6) f associative-commutative           I[f]   = (a*x*y) + (b*(x+y)) + c
                                                 with ac + b - b^2 = 0
(7) f, g associative-commutative        I[f]   = a*x*y with a > 0
    g distributive over f               I[g]   = x + y
                                     or I[f]     as in (6) with a > 0
                                        I[g]   = x+y+d     with d = b/a
(8) f should be rewritten to g          degree(I.1[f]) > degree(I.1[g])
\subsubsection{Interactive ordering mechanisms}
```

```
When the automatic-ordering setting is {\tt off}, LP will prompt users
to confirm any extensions to the registry when as
registered ordering is in use, or to select an action for
an equation LP is unable to orient.  When presented with a prompt like
  \begin{verbatim}
The following sets of suggestions will allow the equation to be oriented into
a rewrite rule:

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

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

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

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

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 assert two new equations that imply the original equation. This action is useful when an *incompatible* equation such as `x/x = y/y` cannot be oriented into a rewrite rule because each side contains a variable not present 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. The resulting system is a conservative extension of the old system.

`interrupt` Interrupts the ordering process and returns LP to the command level.

`kill` Deletes the problematic equation from the system. This action 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 the display ordering command) 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.

3.12 The register command

The *register command* provides LP with hints for use in orienting formulas into terminating sets of rewrite rules.

Syntax

```
<register-command>    ::= register <ordering-constraint>
<ordering-constraint> ::= <height-constraint> | <status-constraint>
                            | <polynomial-constraint>
```

Note: The first word in the <ordering-constraint> can be abbreviated.

Examples

```
register height f > g
register status multiset +
register polynomials + x + y + 1, x + 2
```

Usage

The `register` command adds constraints to a *registry* that LP uses in conjunction with ordering methods that attempt to orient formulas into a set of terminating rewrite rules. The height and status constraints are used by LP's registered orderings, and the polynomial constraints are used by LP's polynomial ordering.

When the automatic-registry setting is `on` and the ordering-method setting is a registered ordering, LP automatically adds height and status constraints to the registry, as necessary, to orient equations in order to ensure that the resulting set of rewrite rules is terminating.

The display ordering command displays the constraints in the registry. Ordering constraints can be removed from the registry with the

3.12.1 Operator height

Height constraints define a partial ordering on operators that induces a partial ordering on terms. This ordering, along with status constraints, is used by LP's registered orderings to orient equations into provably terminating sets of rewrite rules.

Syntax

```
<height-constraint> ::= height <operator-set> ( <height> <operator-set> )+
                          | ( bottom | top ) <operator>+[,]
<height>            ::= > | = | >=
<operator-set>      ::= <operator> | "(" <operator>+[,] ")"
```

Note: The first word in a <height-constraint> can be abbreviated.

Examples

```
height f > g
height => > (/\, \/) > true = false
bottom f
```

Usage

The height constraint `f > g` suggests that terms involving `f` are "more complicated," and should be rewritten to, terms containing `g`. The height constraint `f = g` suggests that terms involving `f` and `g` are equally complicated. The height constraint `f >= g` can be strengthened to either `f > g` or to `f = g`; it is inconsistent with the constraint `g > f`. A compound height constraint, such as the second example, suggests that `=>` is higher than `/\` and `\/`, both of which are higher than `true`, which has the same height as `false`.

The transitive closure of the height constraints in LP's registry defines a partial order on operators, which is used by the dsmpos ordering. The register command will reject any height constraint that is not a consistent extension to its registry.

The `bottom` (`top`) constraints suggest that LP extend its registry when it needs information about a listed operator by giving that operator a lower (higher) height than any non-bottom (non-top) operator. LP does not actually extend its registry until it needs this information. In general, putting operators at the bottom (top) of the registry causes terms to be reduced toward (away from) ones made out of these operators.

3.12.2 Operator status

The status of an operator determines which of its arguments is given the most weight when using a registered ordering to orient an equation containing that operator into a provably terminating rewrite rule.

Syntax

```
<status-constraint> ::= status <status> <operator>+[,]
<status>            ::= left-to-right | right-to-left | multiset
```

Note: The word `status` and the <status> can be abbreviated.

Examples

```
status left-to-right f, g
status multiset +
```

Usage

LP assigns a status to operators, if necessary, when using a registered ordering to orient equations into rewrite rules. Users can assign a status to an operator with the register status command.

The `left-to-right` and `right-to-left` statuses are called *lexicographic* statuses. The assign more weight, respectively, to the leftmost or rightmost arguments of an operator. They are useful when orienting the associativity equation `f(f(x,y),z) = f(x,f(y,z))`. If `f` has `left-to-right`, this equation would be oriented from left to right. If `f` has `right-to-left` status, it would be would be oriented from right to left.

The `multiset` status is appropriate for ac and commutative operators, because it gives all arguments equal weight.

The dsmpos ordering is defined on terms containing an operator without a defined status if the ordering would produce the same results no matter what status was given to that operator. This property allows LP or the user to define the status of an operator without invalidating the proof of termination for previously oriented rewrite rules.

3.13 The unorder command

The *unorder command* turns rewrite rules back into formulas.

Syntax

```
<unorder-command> ::= unorder [ <names> ]
```

Examples

```
unorder
unorder nat
```

Usage

The `unorder` command causes LP to turn the named rewrite rules back into formulas. If no names are specified, LP unorders all rewrite rules.

Even if the automatic-ordering setting is `on`, the unordered formulas are not immediately reordered into rewrite rules. This gives users an opportunity to change the ordering-method or the registry. The formulas will be oriented into rewrite rules in response to an explicit order command or in response to some other command that would invoke LP's automatic ordering (for example, the assert command).

3.14 The unregister command

The *unregister command* causes LP to discard information used to control the registered and polynomial orderings.

Syntax

```
<unregister-command>   ::= unregister <ordering-information>
<ordering-information> ::= registry | ( bottom | top ) <operator>+[,]
```

Note: The first word in the <ordering-information> can be abbreviated.

Examples

```
unregister bottom succ
unregister registry
```

Usage

The `unregister registry` command deletes all information used by the registered and polynomial orderings to orient formulas into rewrite rules.

The `unregister top` and `unregister bottom` commands remove the listed operators from the top and bottom of the registry used by the registered orderings.

4 Forward inference

LP provides mechanisms for proving theorems using both forward and backward inference. Forward inferences produce consequences from LP's logical system. Backward inferences produce subgoals whose proof will suffice to establish a conjecture.

LP provides four methods of forward inference, each of which derives consequences from the axioms in LP's logical system.

Normalization Whenever a new rewrite rule is added to its logical system, LP automatically renormalizes all formulas, rewrite rules, and deduction rules. It discards any formula or rewrite rule that normalizes to `true`. If all hypotheses of a deduction rule normalize to `true`, LP replaces the deduction rule by the formulas in its conclusions. If all conclusions of a deduction rule normalize to `true`, LP discards the deduction rule.

Users can make formulas, rewrite rules, and deduction rules immune or ancestor-immune to protect them from automatic normalization, both to enhance the performance of LP and to preserve particular forms for use in a proof. Users can also deactivate rewrite rules to prevent them from being applied automatically. The normalize and rewrite commands explicitly apply rewrite rules (whether or not active) to facts (whether or not immune).

Deduction Whenever a new deduction rule is added to its logical system, LP automatically applies that deduction rule to all formulas and rewrite rules. Likewise, whenever a formula or rewrite rule is normalized, LP automatically applies all deduction rules to the new normal form.

Inactive deduction rules are not used for automatic deduction, and immune formulas and rewrite rules are not subject to automatic deduction. Users can also apply deduction rules explicitly, for example, to immune formulas.

Critical pairs The computation of critical-pairs and the Knuth-Bendix completion procedure produce consequences from incomplete rewriting systems. It is generally impossible or overly costly to complete a rewriting system. However, the completion procedure can be used to look for inconsistencies in a system, or to perform the final steps in some proofs.

Quantifier elimination The instantiate command enables users to eliminate universal quantifiers, or to replace free variables by terms, in formulas, rewrite rules, and deduction rules. The fix command enables users to eliminate existential quantifiers in formulas and rewrite rulest to produce a conservative extension of LP's logical system.

4.1 The apply command

The *apply command* provides manual control over the application of (possibly passive) deduction rules to (possibly immune) formulas and rewrite rules. It also provides a means of backward inference for finishing the proof of a conjecture.

Syntax

```
<apply-command> ::= apply <names> to <target>
<target>        ::= <names> | conjecture
```

Examples

```
apply passive / deduction-rules to *
apply setExtensionality to conjecture
```

Usage

The first version of the `apply` command applies the named deduction rules, whether or not they are active, to the formulas and rewrite rules named as the <target>, whether or not they are immune.

The second version attempts to prove the current conjecture by explicit deduction using the named deduction rules. The attempt succeeds if the current conjecture matches a conclusion of a named deduction rule and the hypotheses of that deduction rule, under the matching substitution, reduce to `true`. For example, if LP's logical system contains the axioms

```
setAxioms.1:          e \in (x \U y) -> e \in x \/ e \in y
setExtensionality.1: when \A e (e \in s1 <=> e \in s2) yield s1 = s2
```

then the command `apply setExtensionality to conjecture` finishes the proof of the conjecture `x \U x = x`.

4.2 The complete command

The *complete command* provides a method of forward inference that extends the critical-pairs command.

Syntax

```
<complete-command> ::= complete
```

Examples

```
complete
```

Usage

The `complete` command enlarges LP's logical system by adding consequences of facts already in the system. It operates by computing critical pair equations between rewrite rules in the system, orienting them into new rewrite rules, and iterating this procedure until there are no further nontrivial critical pair equations.

When LP's logical system consists solely of active, nonimmune, quantifier-free equations and rewrite rules, the `complete` command attempts to transform the logical system into a convergent set of rewrite rules that has the same equational theory. The completion procedure is based on the Peterson and Stickle extension of the *Knuth-Bendix completion procedure.*

LP terminates the completion procedure if the current conjecture reduces to `true`.

A convergent rewriting system can be used to prove theorems by normalization using the prove command or to reduce terms to a canonical form using the show normal-form command. The completion-mode setting controls the operation of the completion procedure. You can interrupt and resume the operation of the completion procedure.

See also proofs by consistency.

4.3 The critical-pairs command

The *critical-pairs command* provides a method of forward inference that produces consequences from pairs of rewrite rules. This method can be used to increase the amount of confluence in LP's rewriting system.

Syntax

```
<critical-pairs-command> ::= critical-pairs <names> with <names>
```

Examples

```
critical-pairs *Hyp with *
```

Usage

The `critical-pairs` command directs LP to compute critical pair equations between the rewrite rules named by the first <names> and those named by the second. LP adds any nontrivial critical pair equations (i.e., nonidentities) to its logical system. If LP reduces the current conjecture to an identity upon orienting a critical pair equation into a rewrite rule, it terminates the critical pair computation. LP performs critical pair computations in the order determined by the combined sizes of the left sides of the rewrite rules.

LP also computes critical pairs in response to the complete command. LP keeps track of which rewrite rules have had their critical pairs computed, and does not recompute critical pairs between those rules. See also the forget command.

Critical pair equations between two rewrite rules result from using them to rewrite a common term in two different ways. Suppose that `R1` is the rewrite rule `l1 -> r1` and `R2` is `l2 -> r2`. Suppose also that `R1` and `R2` have no variables in common (LP ensures that this is the case by substituting fresh variables for those in `R2`). If `l2` can be unified with a nonvariable subterm of `l1`, then both `R1` and `R2` can be used to rewrite some substitution instance of `l1`. A *critical pair equation* between `R1` and `R2` is an equation relating the results of these two rewritings.

More precisely, if `s1` is a nonvariable subterm of `l1` that does not contain any variables bound by outer quantifiers in `l1`, if `sigma` is a substitution that unifies `l2` with `s1` and that does not introduce any variables bound by outer quantifiers in `l1`, and if `l1'` is the result of substituting `r2` for `s2` in `l1`, then `sigma(l1') = sigma(r1)` is a critical pair equation between `R1` and `R2`.

Examples:

```
    Rewrite rules                          Critical pair equations

1.  f(x) * f(y)         -> f(x * y)        b * f(y) = f(a * y)
    f(a)                -> b               f(x) * b = f(x * a)

2.  P(x) => Q(x)        -> true            true => Q(c) <=> true
    P(c)                -> true              ... which reduces to Q(c)

3.  \E x (f(x) = g(y)) -> true             \E x (f(x) = c)
    g(c)                -> c
```

When the principal operator of `l1` or `l2` is an ac operator, say `*`, LP generalizes the critical pair computation by computing *Petersen-Stickel extensions* `Ri'` having the form `li*x -> ri*x` of `R1` and `R2`, and then by computing critical pair equations between {`R1`, `R1'`} and {`R2`, `R2'`}. Thus, `i(x) * x -> e` has three nontrivial critical pair equations with itself when `*` is ac:

- `i(e) * e = e`, which results from unifying `i(x) * x` with the nonvariable subterm `y * z` of the left side of `i(y) * y * z -> z`,
- `e * x = i(i(x)) * e`, which results from the unifier `i(i(x)) * i(x) * x` of `i(x) * x * y` and `i(x') * x' * y'` (via the map from `x'` to `i(x)` and `y'` to `x`), and

- `e * i(x) = i(x*y) * e * y`, which results from the unifier `i(x*y) * i(x) * x * y` of the same terms (via the map from `x'` to `x*y` and `y'` to `i(x)`).

4.4 The fix command

The *fix command* provides a method of forward inference, which can be used to eliminate an existential quantifier from a fact in LP's logical system.

Syntax

```
<fix-command> ::= fix <variable> as <term> in <names>
```

Examples

```
fix x as s(0) in *Hyp
```

Usage

The `fix` command eliminates the unique accessible prenex-existential quantifier over the variable from the named facts and substitutes the term for all occurrences of the variable bound by that quantifier. For example, given the formulas

```
user.1: \E x \A y (f(y) = x)
user.2: \E z (x < z)
```

the commands `fix x as c in user` and `fix z as bigger(x) in user` produce the results

```
user.1.1: f(y) = c
user.2.1: x < bigger(x)
```

LP will reject a `fix` command unless the following conditions are satisfied.

- The named facts must contain exactly one accessible prenex-existential quantifier over the variable (because it is not sound to instantiate two different existential quantifiers by the same term).
- The term must have the form `sk(x1, ..., xn)`, where `sk` is a function identifier that does not appear in any fact or in the current conjecture, and where `x1`, ..., `xn` include all variables that are bound by outer (explicit or implicit) prenex-universal quantifiers in the formula containing the eliminated quantifier. See Skolem function.

LP automatically changes bound variables in the named facts, as needed, to avoid having them bind (or capture) variables that occur free in the term. This action, together with the above two conditions, guarantee that the results constitute a conservative extension to LP's logical system, i.e., that any consequence of the extended system is either a consequence of the original system or contains an occurrence of `sk`. The last condition prevents unsound derivations such as `c ~= c` from `\E x (x ~= c)` or `\A y (c = y)` from `\A y \E x (x = y)`.

4.5 The instantiate command

The *instantiate command* provides a method of forward inference, which can be used to substitute terms for free variables, or to eliminate universal quantifiers, from facts in LP's logical system.

Syntax

```
<instantiate-command> ::= instantiate (<variable> by <term>)+, in <names>
```

Examples

```
instantiate s2 by s1 \U s1 in setExtensionality
instantiate x by 0, y by 1 in lemmas
```

Usage

The `instantiate` command substitutes (simultaneously) the specified terms for the named free variables in the named formulas, rewrite rules, and deduction rules; if a named variable does not occur free in a named fact, but is bound by an accessible prenex-universal quantifier in that fact, then that quantifier is eliminated before the substitution is performed. LP automatically changes bound variables in the named facts, if needed, to avoid having them bind (or capture) variables that occur free in the specified terms.

LP normalizes any nonimmune results from an instantiation, discarding those that normalize to `true` and orienting any resulting formulas into rewrite rules if the automatic-ordering setting is `on`. The activity and immunity of the results are determined by the current values of the activity and immunity settings. When instantiating a rewrite rule, LP does not use that rule in normalizing the result.

4.6 The normalize command

The *normalize command* provides manual control over the application of (possibly passive) rewrite rules to (possibly immune) formulas, rewrite rules, and deduction rules. It also provides a means of backward inference.

Syntax

```
<normalize-command> ::= normalize <target> [ with [ reversed ] <names> ]
<target>            ::= <names> | conjecture
```

Examples

```
normalize immune / lemmas with *
normalize conjecture with distributiveLaws
```

Usage

The first version of the `normalize` command normalizes the formulas, deduction rules, and rewrite rules in the <target>, whether or not they are immune, using the hardwired rewrite rules together with the rewrite rules obtained as described below. Rewrite rules in the <target> that are also named by <names> are not normalized.

If `reversed` is present, all named formulas and rewrite rules, whether or not they are active, that can be oriented from right to left into legal rewrite rules are used with that orientation. If `reversed` is not present, all named rewrite rules and formulas, whether or not they are active, that can be oriented from left to right are used with that orientation. If no <names> are given, all rewrite rules and left-to-right orientable formulas are used.

The second version of the **normalize** command normalizes the current conjecture using the rewrite rules obtained as just described.

The **normalize** command is typically used to "open up" definitions using a set of passive rewrite rules. When **reversed** is present, the named rewrite rules should ordinarily be passive to prevent them from immediately undoing the result of the **normalize** command. See also the rewrite command.

4.7 The rewrite command

The *rewrite command* provides manual control over the application of (possibly passive) rewrite rules to (possibly immune) formulas, rewrite rules, and deduction rules. It also provides a means of backward inference.

Syntax

```
<rewrite-command> ::= rewrite <target> [ with [ reversed ] <names> ]
```

Examples

```
rewrite immune / lemmas with *
rewrite conjecture with distributiveLaws
```

Usage

The first version of the **rewrite** command rewrites some term in each of the formulas, deduction rules, and rewrite rules in the <target>, whether or not they are immune, using the hardwired rewrite rules together with the rewrite rules obtained as described below. Rewrite rules in the <target> that are also named by <names> are not rewritten.

If **reversed** is present, all named formulas and rewrite rules, whether or not they are active, that can be oriented from right to left into legal rewrite rules are used with that orientation. If **reversed** is not present, all named rewrite rules and formulas, whether or not they are active, that can be oriented from left to right are used with that orientation. If no <names> are given, all rewrite rules and left-to-right orientable formulas are used.

The second version of the **rewrite** command rewrites some term in the current conjecture using the rewrite rules obtained as just described.

The **rewrite** command is typically used to "open up" definitions using a set of passive rewrite rules or to undo an application of a rewrite rule. When **reversed** is present, the named rewrite rules should ordinarily be passive to prevent them from immediately undoing the result of the **rewrite** command. See also the normalize command.

4.8 Proofs by consistency

LP permits manual *proofs by consistency* (also known as proofs by *inductionless induction*) in logical systems that consist solely of quantifier-free equations and rewrite rules. Such proofs require that the *Huet-Hullot principle of definition* be satisfied, that is, that all ground (variable-free) terms be equal, with respect to the equations and rewrite rules in the system, to ground terms involving a set of free generators. In axiomatizations of abstract data types, the generators of the data type will usually have this property. Axiomatizations of sets usually fail to satisfy the property because the **insert** operator is not free.

A proof by consistency proceeds by adding an equation to a complete system and running the completion procedure. If that procedure terminates without generating an inconsistency, then the equation is valid in the initial model; if it terminates with an inconsistency, the equation is not valid; if it does not terminate, the equation may or may not be valid.

5 Backward inference

The prove command causes LP to initiate a proof of a conjecture by backward inference. LP provides nine methods of backward inference for proving formulas; in addition, it provides automatic methods of backward inference for proving deduction rules, induction rules, and operator theories. In each method, LP generates a set of subgoals, that is, lemmas to be proved that together are sufficient to imply the conjecture. For some methods, it also generates additional hypotheses that may be used to prove particular subgoals.

Proof by normalization Normalization rewrites conjectures. If a conjecture normalizes to `true`, it is a theorem. Otherwise the normalized conjecture becomes the subgoal to be proved.

Proof by cases A proof by cases divides a proof into specified cases, enabling the conjecture to be rewritten further using the case hypotheses.

Proof by contradiction Proofs by contradiction provide an indirect method of proof. If LP detects an inconsistency after adding the negation of the conjecture to its logical system, then it concludes that the conjecture is a theorem.

Proof by induction Proofs by induction are based on the induction rules associated with assertions that some sort is generated by a set of operators or that some binary relation is well founded.

Proof of conditionals Proofs of conditionals are simplified proofs by cases applicable to conjectures of the form `(if t1 then t2 else t3) = t4`, where `t4` does not begin with `if`.

Proof of conjunctions Proofs of conjunctions provide a way to reduce the expense of equational term rewriting when proving conjectures of the form `t1 /\ ... /\ tn`.

Proof of implications Proofs of implications are simplified proofs by cases applicable to conjectures of the form `t1 => t2`.

Proof of logical equivalence Proofs of logical equivalence are simplified proofs by cases applicable to conjectures of the form `t1 <=> t2`.

Proof by generalization Proofs by generalization provide a means of establishing a conjecture that contains a universal quantifier by proving an instance of the conjecture with the quantified variable replaced by an appropriate Skolem constant or function.

Proof by specialization Proofs by specialization provide a means of establishing a conjecture that contains an existential quantifier by proving an instance of the conjecture with the quantified variable replaced by a particular term.

Proofs using deduction rules The apply command provides a means of establishing a conjecture that matches the conclusion of a deduction rule.

The proof-methods setting enables users to specify which of these methods of backward inference are applied automatically and in what order.

5.1 Overview of proof methods for formulas

LP provides a variety of methods for proving formulas. Some represent standard proof techniques (proofs by cases, contradiction, induction, and normalization). Two (generalization and specialization) help establish conjectures containing quantifiers. Others assist in proving formulas with particular syntactic forms (implications, logical equivalences, conditionals, and conjunctions).

Syntax

```
<proof-method>          ::= <default-proof-method> | cases <term>+,
                              | contradiction | default
                              | generalizing <variable> from <term>
                              | induction [ [ on <variable> ]
                                 [ depth <number> ] [ [ using ] <names> ]
                              | specializing ( <variable> to <term> ) +,
<default-proof-method>  ::= /\-method | =>-method | <=>-method | if-method
                              | explicit-commands | normalization
```

Note: The first word of the <proof-method> can be abbreviated.

Examples

```
=>
cases x < 0, x = 0, x > 0
induction on x
```

Usage

Users can specify a method of proof in the prove command that introduces a conjecture, in a subsequent resume command, or in a list of default proof-methods that LP uses when no method is specified in a prove command or when it creates subgoals.

The `default` method specifies the use of the first applicable proof method in the value of the proof-methods setting. See the individual descriptions of the other proof methods for information about their use.

5.2 Proofs by normalization

LP uses active rewrite rules to normalize conjectures. If a conjecture normalizes to `true`, it is a theorem. Otherwise, the normalized conjecture becomes the current subgoal to be proved. For example, LP succeeds in proving the conjecture

```
{e} \subseteq insert(e, s)
```

from the axioms

```
{e} = insert(e, {})
e \in insert(e1, x) <=> e = e1 \/ e \in s
{} \subseteq s
insert(e, x) \subseteq y <=> e \in y /\ x \subseteq y
```

by using them to reduce it to `true`. But the conjecture `x \subseteq x` is irreducible, and LP treats it as a subgoal to be proved by some other proof method.

Passive rewrite rules can be applied explicitly to a conjecture by the normalize and rewrite commands. These commands can be used to control when definitions are expanded, or when nonsimplifying rewrite rules (such as distributivity) are applied.

5.3 Proofs by cases

Conjectures can often be simplified by dividing a proof into cases. When a conjecture reduces to `true` 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 F by cases F1, ..., Fn` directs LP to prove a formula `F` by division into `n` cases defined by the formulas `F1`, ..., `Fn` (or into two cases, `F1` and `~F1` if `n = 1`). The command `resume by cases F1, ..., Fn` directs LP to resume the proof of the current conjecture by division into cases.

A proof by cases involves `n+1` subgoals. If `n > 1`, the first subgoal involves proving `F1 \/ ... \/ Fn` to show that the cases exhaust all possibilities. If `n = 1`, LP generates a default second case of `~F1`, but does not generate a disjunction as the first subgoal. Then, for each case `i`, LP generates a subgoal `F'` and an additional hypothesis `Fi'` by substituting new constants for the free variables of `Fi` in both `F` and `Fi`. If an inconsistency results from adding a case hypothesis, that case is impossible and the subgoal is vacuously true. Otherwise the subgoal must be shown to follow from the axioms supplemented by the case hypothesis. The names of the case hypotheses have the form <simpleId>`CaseHyp.`<number>, where <simpleId> is the current value of the name-prefix setting.

In each case of a proof by cases, LP first adds the case hypothesis without using it to reduce the other rewrite rules in the system. Only if this action fails to reduce the desired conclusion to `true` does LP use the case hypothesis to reduce the other rewrite rules.

5.4 Proofs by contradiction

Proofs by contradiction provide an indirect method of proof. The command `prove F by contradiction` directs LP to prove a formula `F` by deriving an inconsistency from LP's logical system supplemented by the hypothesis `~F'`, where `F'` is the result of substituting new constants for the free variables in `F`. (This hypothesis is logically equivalent to the negation of `F` because introducing new constants is equivalent to replacing the implicit universal quantifiers by existential quantifiers.) The name of the hypothesis has the form <simpleId>`ContraHyp.`<number>, where <simpleId> is the current value of the name-prefix setting.

The command `resume by contradiction` directs LP to resume the proof of the current conjecture by contradiction.

5.5 Proofs by induction

The command `prove F by induction on x using IR` directs LP to prove a formula `F` by induction on the variable `x` using the named induction rule. The name of the variable and/or that of the induction rule can be omitted if they can be inferred (e.g., because induction is possible on only one variable in `F` and there is only one induction rule for the sort of that variable). The keyword `on` is optional. The keyword `using` can be omitted if the name of a variable is given.

The command `resume by induction` directs LP to resume the proof of the current conjecture by induction.

LP supports proofs both by structural and well-founded induction. Induction rules beginning with `generated by` provide the basis for proofs by structural induction. Induction rules beginning with `well founded` provide the basis for proofs by well-founded induction.

LP generates appropriate subgoals for each kind of proof by induction. Some of those subgoals introduce additional hypotheses, known as *induction hypotheses*. The names of the induction hypotheses have the form <simpleId>`InductHyp.`<number>, where <simpleId> is the current value of the name-prefix setting.

5.5.1 Structural induction

A proof of a formula `F` by structural induction on a variable `x` of sort `S` is based on an induction rule that specifies a set `G` of generators for `S`, that is, a set of operators with range `S`. LP generates subgoals for the basis and induction steps in a proof by structural induction, as follows.

- The *basis subgoals* involve proving the formulas that result from substituting the basis generators in `G` for `x` in `F`. Basis generators are those with no arguments of sort `S`; fresh variables are used for variables of other sorts, as in {`e`}.
- LP introduces additional hypotheses for the *induction subgoals* by substituting one or more new constants for `x` in `F`. Each induction subgoal involves proving a formula that results from substituting a nonbasis generator of `G` (applied to these constants) for `x` in `F` (e.g., `s(xc)` or `xc \U xc1`).

Examples

```
assert sort Nat generated by 0, s
prove 0 + x = x by induction
  Basis subgoal:        0 + 0 = 0
  New constant:         xc
  Induction hypothesis: 0 + xc = xc
  Induction subgoal:    0 + s(xc) = s(xc)
```

```
assert sort Set generated by {}, {__}, \U
prove x \subseteq x by induction
  Basis subgoals:        {} \subseteq {},    {e} \subseteq {e}
  New constants:         xc,                 xc1
  Induction hypotheses: xc \subseteq xc,    xc1 \subseteq xc1
  Induction subgoal:    (xc \U xc1) \subseteq (xc \U xc1)
```

See also

- Proofs by multilevel induction
- Proofs by well-founded induction

5.5.2 Multilevel structural induction

The command `prove F by induction on x depth n using IR` directs LP to prove a formula `F` by n-level structural induction; by default, n is 1. For example, proving a formula `F(x)` by depth 2 induction using the induction rule `sort Nat generated by 0, s`, involves proving two basis subgoals, `F(0)` and `F(s(0))`, and one induction subgoal, `F(s(s(xc)))`, given `F(xc)` and `F(s(xc))` as induction hypotheses.

In general, LP constructs the basis and induction steps using the set G of generators for the sort `S` of `x` specified by the induction rule `IR`, as follows.

Definition 1. A (G,`F`,C)-*ground term*, where C is a set of constants of sort `S`, is a quantifier-free term of sort `S` in which all operators are either in C or are inductive generators in G, no variable has sort `S`, no variable occurs more than once, and no variable also occurs in `F`.

Definition 2. A $(G,\mathtt{F})$-*ground term* is a $(G,\mathtt{F},B)$-ground term, where B is the set of basis generators in G.

Definition 3. The *depth* of a quantifier-free term is 0 if the term consists of a variable; otherwise it is one more than the maximum depth of its arguments.

Definition 4. A $(G,\mathtt{F},\{\mathtt{c1},\ldots,\mathtt{cm}\})$-ground term is *canonical* if it contains exactly one occurrence of each of `c1`, ..., `ck` for some $k <= m$.

The basis step involves proving all formulas of the form `F(x gets t)` where `t` is an $(G,\mathtt{F},B)$-ground term of depth at most n.

The induction step introduces a set $C = \{\mathtt{c1},\ldots,\mathtt{cm}\}$ of new constants of sort `S`, where m is the maximum number of arguments of sort `S` in a generator in G raised to the power n. The induction step involves proving all formulas of the form `F(x gets t)`, where `t` is a canonical $(G,\mathtt{F},C)$-ground term of depth $n{+}1$. The induction hypotheses available in the induction step consist of all formulas of the form `F(x gets t)`, where `t` is a $(G,\mathtt{F},C)$-ground term of depth at most n.

When n is 1, the induction hypotheses consist of all formulas of the form `F(x gets c)`, where `c` is in C; and the induction step involves proving all formulas of the form `F(x gets t)`, where `t` is a canonical $(G,\mathtt{F},C)$-ground term of depth 2.

Examples:

```
Gener-  Level   Basis             Induction                   Induction
ators           Subgoals          Hypotheses                  Subgoals

0, s      1     f(0)              f(c)                        f(s(c))

0, s      2     f(0)              f(c)                        f(s(s(c)))
                f(s(0))           f(s(c))

0, 1, +   1     f(0)              f(c1)                       f(c1+c2)
                f(1)              f(c2)

0, 1, +   2     f(0)              f(c1), f(c2)                f(c1+(c2+c3)
                f(1)              f(c3), f(c4)                f((c1+c2)+c3)
                f(0+0)            f(c1+c1), f(c1+c2), ...     f((c1+c2)+(c3+c4))
                f(0+1)            f(c2+c1), f(c2+c2), ...
                f(1+0)            f(c3+c1), f(c3+c2), ...
                f(1+1)            f(c4+c1), f(c4+c2), ...


nil       1     f(nil)            f(c)                        f(cons(x, c))
cons

nil       2     f(nil)            f(c)                        f(cons(x,cons(y,c))
cons            f(cons(x,nil))    f(cons(x,c))
```

5.5.3 Well founded induction

To prove a formula `F` by well-founded induction over a variable `x` of sort `S` using a relation `R` that has been asserted or proved to be well founded, LP generates a single subgoal that involves proving the formula `F(xc)` using the additional hypothesis `R(x, xc) => F(x)`, where `xc` is a new constant of sort `S`.

Example

```
assert well founded <
prove 0 + x = x by induction

  New constant:         xc
  Induction subgoal:    0 + xc = xc
  Induction hypothesis: x < xc => 0 + x = x
```

5.5.4 Proofs by generalization

The command `prove F by generalizing x from t` directs LP to prove a formula `F` by creating a single subgoal in which the unique accessible (explicit or implicit) prenex-universal quantifier over the variable `x` has been eliminated from `F` and all occurrences of `x` bound by that quantifier have been replaced by `t`.

The command `resume by generalizing x from t` directs LP to resume the proof of the current conjecture by this method.

This proof method, which eliminates a universal quantifier from a conjecture, is the dual of the fix command, which eliminates an existential quantifier from a fact. It is subject to restrictions on `x` and `t` as for the fix command.

This command is unlikely to be used when `F` contains free variables other than `x`. When `x` is the only free variable in `F`, the only restriction is that `t` be a constant that does not occur in `F` or in any fact in LP's logical system. For example, the command

```
prove \A x (f(x) = c) by generalizing x from d
```

is allowed when `d` is a new constant and reduces the proof of the conjecture to establishing the subgoal `f(d) = c`. See Skolem-constant.

5.5.5 Proofs by specialization

The command `prove F by specializing x to t` directs LP to prove a formula `F` by creating a single subgoal in which all accessible prenex-existential quantifiers over the variable `x` have been eliminated from `F` and all occurrences of `x` bound by those quantifier have been replaced by `t`.

The command `resume by generalizing x from t` directs LP to resume the proof of the current conjecture by this method.

This proof method, which eliminates existential quantifiers from a conjecture, is the dual of the instantiate command, which eliminates universal quantifiers from facts.

For example, the command

```
prove \A x \E y (x < y) by specializing y to s(x)
```

reduces the proof of the conjecture to establishing the subgoal `x < s(x)`.

5.5.6 Proofs of conjunctions

Proofs of conjunctions can be slow because the operator /\ is associative and commutative, and matching modulo such operators is inherently slow. The command `prove t1 /\ ... /\ tn by /\` provides a way to reduce this expense by directing LP to prove each of the conjuncts as a separate subgoal.

The command `resume by /\` directs LP to resume the proof of the current conjecture using this method. It 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 lengthening the proof considerably, for example, when the same sequence of commands or the same lemma is needed to prove more than one conjunct.

5.5.7 Proofs of implications

Proofs of implications can be carried out using a simplified form of a proof by cases. The command

```
prove t1 => t2 by =>
```

directs LP to prove the subgoal `t2'` using the hypothesis `t1'`, where `t1'` and `t2'` are obtained as in a proof by cases, that is, by substituting new constants for the free variables of `t1` in both `t1` and `t2`. The name of the hypothesis has the form <simpleId>`ImpliesHyp.`<number>, where <simpleId> is the current value of the name-prefix setting.

For example, given the axioms `a => b` and `b => c`, the command `prove a => c by =>` orients the hypothesis `a` into a rewrite rule `a -> true` and attempts to prove `c` as a subgoal. The proof succeeds automatically: LP orients the hypothesis into a rewrite rule `a -> true`, uses it to normalize the first axiom to `b`, orients the result into a rewrite rule `b -> true`, and uses it to normalize the second axiom to `c`, which establishes the subgoal.

The command `resume by =>` directs LP to resume the proof of the current conjecture using this method. It is applicable only when the current conjecture has been reduced to an implication.

Users should beware of using the `=>` proof method prematurely, because it causes LP to replace all free variables in the conjecture by constants, which makes it impossible to continue the proof by induction on one of those variables.

5.5.8 Proofs of logical equivalence

The command `prove t1 <=> t2 by <=>` directs LP to prove the conjecture by proving two implications, `t1 => t2` and `t2 => t1`. LP substitutes new constants for the free variables in both `t1` and `t2` to obtain terms `t1'` and `t2'`, and it creates two subgoals: the first involves proving `t2'` using `t1'` as an additional hypothesis, the second proving `t1'` using `t2'` as an additional hypothesis. The names of the hypotheses have the form <simpleId>`ImpliesHyp.`<number>, where <simpleId> is the current value of the name-prefix setting.

The command `resume by <=>` directs LP to resume the proof of the current conjecture using this method. It is applicable only when the current conjecture has been reduced to a formula of the form `t1 <=> t2` or of the form `t1 = t2` when `t1` and `t2` are boolean-valued terms.

5.5.9 Proofs of conditionals

Proofs of formulas involving the conditional operator `if` can be carried out using a simplified proof by cases. The commands

```
prove if t1 then t2 else t3 by if-method
prove (if t1 then t2 else t3) = t4 by if-method
```

direct LP to prove the conjectures by division into two cases, `t1` and `~t1`. LP substitutes new constants for the free variables of `t1` in all terms `ti` to obtain terms `ti'`. In the first case, it assumes `t1'` as an additional hypothesis and attempts to prove `t2'` (or `t2' = t4'`) as a subgoal. In the second case, it assumes `t1' =`

`false` as an additional hypothesis and attempts to prove `t3'` (or `t3' = t4'`). The names of the hypotheses have the form <simpleId>`IfHyp.`<number>, where <simpleId> is the current value of the name-prefix setting.

The command `resume by if-method` directs LP to resume the proof of the current conjecture using this method. It is applicable only when the current conjecture has been reduced to a formula of the form `if t1 then t2 else t3` or of the form `(if t1 then t2 else t3) = t4`, where `t4` does not begin with `if`.

5.5.10 Proofs of deduction rules

LP permits users to prove deduction rules as well as assert them. It automatically generates a single subgoal when asked to initiate a proof of a deduction rule. The subgoal involves showing that the conjunction of the hypotheses of the deduction rule implies the conjunction of its conclusions.

For example, the command

```
prove when \A x:Elem (x \in s:Set <=> x \in t:Set) yield s = t
```

causes LP to generate a single subgoal which involves proving the formula

```
\A x:Elem (x \in s:Set <=> x \in t:Set) => s = t
```

LP also generates a single subgoal when asked to initiate a proof of a partitioned-by. This subgoal is the one associated with the translation of the partitioned-by into a deduction rule.

5.5.11 Proofs of induction rules

LP permits users to prove induction rules as well as assert them. To prove a structural induction rule such as

```
sort Set generated by {}, {__}, \U
```

LP creates a single subgoal that involves proving the formula `isGenerated(x)` using the hypotheses

```
isGenerated({})
isGenerated({e})
isGenerated(s1) /\ isGenerated(s2) => isGenerated(s1 /\ s2)
```

where `isGenerated` is a new operator with signature `Set->Bool`. The names of the hypotheses have the form <simpleId>`GenHyp.`<number>, where <simpleId> is the current value of the name-prefix setting. User guidance is generally required to finish the proof, for example, by using the induction rule `sort Set generated by {}, insert`.

To prove a structural induction rule such as

```
sort Nat generated freely by 0:->Nat, f:Nat->Nat, g:Nat->Nat
```

LP also attempts to prove the subgoals

```
f(n) ~= 0          f(n) = f(n1) <=> n = n1          f(n) ~= g(n1)
g(n) ~= 0          g(n) = g(n1) <=> n = n1
```

in addition to the subgoal `isGenerated(n)`.

To prove a well-founded induction rule such as `well founded <`, LP creates a single subgoal that involves proving the formula `isGenerated(x)` using the hypothesis

```
\A n1 (n1 < n => isGenerated(n1)) => isGenerated(n)
```

5.5.12 Proofs of operator theories

LP permits users to prove operator theories as well as assert them. Proving that an operator `+` is commutative involves proving a single subgoal consisting of the formula `x + y = y + x`. Proving that an operator is associative-commutative involves proving an additional subgoal consisting of the formula `x + (y + z) = (x + y) + z`.

5.6 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 method is useful in several situations.

- It can be used 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.
- It can be used to delay the start of a proof until the user has had an opportunity to make the conjecture immune, so that it is not reduced once it has been proved.

5.6.1 The apply command

The *apply command* provides manual control over the application of (possibly passive) deduction rules to (possibly immune) formulas and rewrite rules. It also provides a means of backward inference for finishing the proof of a conjecture.

Syntax

```
<apply-command> ::= apply <names> to <target>
<target>        ::= <names> | conjecture
```

Examples

```
apply passive / deduction-rules to *
apply setExtensionality to conjecture
```

Usage

The first version of the `apply` command applies the named deduction rules, whether or not they are active, to the formulas and rewrite rules named as the <target>, whether or not they are immune.

The second version attempts to prove the current conjecture by explicit deduction using the named deduction rules. The attempt succeeds if the current conjecture matches a conclusion of a named deduction rule and the hypotheses of that deduction rule, under the matching substitution, reduce to `true`. For example, if LP's logical system contains the axioms

```
setAxioms.1:          e \in (x \U y) -> e \in x \/ e \in y
setExtensionality.1: when \A e (e \in s1 <=> e \in s2) yield s1 = s2
```

then the command `apply setExtensionality to conjecture` finishes the proof of the conjecture `x \U x = x`.

5.6.2 The cancel command

The *cancel command* enables users to abort proofs or to change proof methods.

Syntax

```
<cancel-command> ::= cancel [ all | lemma ]
```

Examples

```
cancel
cancel all
cancel lemma
```

Usage

The `cancel` command without either modifier cancels the proof of the current conjecture and suspends work on other proofs until an explicit resume command is issued. If the current conjecture is a subgoal for proving a formula, LP pops the proof stack back to the parent of the subgoal and sets its proof method to `default`; if it is a subgoal for a nonformula (e.g., for an induction rule), LP also cancels the proof of the parent of the subgoal.

The command `cancel all` cancels the proofs of all conjectures.

The command `cancel lemma` cancels the proof of the conjecture introduced by the last prove command, together with the proofs of all subgoals

5.6.3 The normalize command

The *normalize command* provides manual control over the application of (possibly passive) rewrite rules to (possibly immune) formulas, rewrite rules, and deduction rules. It also provides a means of backward inference.

Syntax

```
<normalize-command> ::= normalize <target> [ with [ reversed ] <names> ]
<target>            ::= <names> | conjecture
```

Examples

```
normalize immune / lemmas with *
normalize conjecture with distributiveLaws
```

Usage

The first version of the `normalize` command normalizes the formulas, deduction rules, and rewrite rules in the <target>, whether or not they are immune, using the hardwired rewrite rules together with the rewrite rules obtained as described below. Rewrite rules in the <target> that are also named by <names> are not normalized.

If `reversed` is present, all named formulas and rewrite rules, whether or not they are active, that can be oriented from right to left into legal rewrite rules are used with that orientation. If `reversed` is not present, all named rewrite rules and formulas, whether or not they are active, that can be oriented from left to right are used with that orientation. If no <names> are given, all rewrite rules and left-to-right orientable formulas are used.

The second version of the `normalize` command normalizes the current conjecture using the rewrite rules obtained as just described.

The `normalize` command is typically used to "open up" definitions using a set of passive rewrite rules. When `reversed` is present, the named rewrite rules should ordinarily be passive to prevent them from immediately undoing the result of the `normalize` command. See also the rewrite command.

5.6.4 The prove command

The *prove command* initiates the proof of a conjecture.

Syntax

```
<prove-command> ::= prove <assertion> [ by <proof-method> ]
```

Examples

```
prove e \in {e}
prove i * (j + k) = (i * j) + (i * k) by induction on i
prove sort Set generated by {}, insert
```

Usage

The `prove` command initiates an attempt to prove a conjectured assertion using the specified method. If no method is specified, LP uses the one determined by the current value of the proof-methods setting. LP assigns a name to the conjecture using the current name-prefix setting, unless the assertion begins with :<simpleId>:, in which case LP uses that identifier as the name-prefix for the assertion. If and when the proof of a conjecture succeeds, LP adds the conjecture to its logical system and uses it as if it had been asserted by the assert command. The activity and immunity of the conjecture, however, are determined when it is introduced by a `prove` command, not when it is proved; these attributes can be changed using the make command.

LP maintains a stack of *proof contexts* for conjectures whose proofs are not yet complete. Each proof context consists of a conjecture, a logical system of facts available for the proof, and values for the local settings that govern the proof. The conjecture in the topmost proof context on the stack is known as the *current conjecture*.

The `prove` command pushes a new proof context on top of the stack. Certain proof methods create *subgoals* for proving a conjecture. LP associates a separate proof context with each subgoal, and it adds appropriate additional facts, called *hypotheses*, to the logical system in that proof context.

The user can cancel the proof of a conjecture with the cancel command, which pops the stack of proof contexts. Or the user can resume the proof of the current conjecture with the resume command (for example, to specify a new method of proof or after proving a lemma). Whenever a proof succeeds or is canceled, LP pops the stack of proof contexts, restores its logical system and settings to those in effect before work began on the conjecture (thereby discarding any lemmas proved while working on the conjecture), adds the conjecture to the system if it was proved, and resumes work on the new current conjecture. As soon as LP can establish the current conjecture, it terminates any forward inference mechanisms (such as internormalization of the rewriting system or the computation of critical-pair equations) that may be in progress.

See also

- Proof methods available for proving formulas
- Proofs of deduction rules, including those associated with `partitioned by`.
- Proofs of induction rules
- Proofs of operator theories
- Using the normalize and rewrite commands with the current conjecture

5.6.5 The qed command

The *qed command* constitutes a claim that all proofs are finished.

Syntax

```
<qed-command> ::= qed
```

Examples

```
qed
```

Usage

The `qed` command checks that the proof stack is empty. If it is not, LP prints an error message and halts execution of all `.lp` files.

See also

- Box checking

5.6.6 The resume command

The *resume command* allows the user to resume work on the current conjecture.

Syntax

```
<presume-command> ::= resume [ by <proof-method> ]
```

Examples

```
resume by induction on i
resume by cases x < 0, x = 0, x > 0
```

Usage

The **resume** command resumes work on the current conjecture using the specified method. If no method is specified, LP uses the method in effect when the proof was suspended.

See also

- The prove command

5.6.7 The rewrite command

The *rewrite command* provides manual control over the application of (possibly passive) rewrite rules to (possibly immune) formulas, rewrite rules, and deduction rules. It also provides a means of backward inference.

Syntax

```
<rewrite-command> ::= rewrite <target> [ with [ reversed ] <names> ]
```

Examples

```
rewrite immune / lemmas with *
rewrite conjecture with distributiveLaws
```

Usage

The first version of the `rewrite` command rewrites some term in each of the formulas, deduction rules, and rewrite rules in the <target>, whether or not they are immune, using the hardwired rewrite rules together with the rewrite rules obtained as described below. Rewrite rules in the <target> that are also named by <names> are not rewritten.

If `reversed` is present, all named formulas and rewrite rules, whether or not they are active, that can be oriented from right to left into legal rewrite rules are used with that orientation. If `reversed` is not present, all named rewrite rules and formulas, whether or not they are active, that can be oriented from left to right are used with that orientation. If no <names> are given, all rewrite rules and left-to-right orientable formulas are used.

The second version of the `rewrite` command rewrites some term in the current conjecture using the rewrite rules obtained as just described.

The `rewrite` command is typically used to "open up" definitions using a set of passive rewrite rules or to undo an application of a rewrite rule. When `reversed` is present, the named rewrite rules should ordinarily be passive to prevent them from immediately undoing the result of the `rewrite` command. See also the normalize command.

5.6.8 The box and diamond commands

A `[]` (*box*) or a <> (*diamond*) appearing as the first nonblank characters of an input line begins a checkable comment, which LP uses to ensure that replayed proofs do not diverge from the way users expect them to proceed.

Syntax

```
<diamond-command> ::= <tt><></tt> <character>*
<box-command>     ::= [] <character>*
```

Examples

```
<> induction step
[] induction step
```

Usage

LP generates `[]`'s and <>'s in the history and script files. Users generally copy the <> and `[]` annotations supplied by LP into their command files rather than attempt to generate these annotations themselves. LP generates a <> whenever it introduces a subgoal in a proof, and it generates a `[]` whenever it finishes the proof of a subgoal or of 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 <>'s.

LP checks <>'s and `[]`'s when it executes commands from a `.lp` file and the box-checking setting is `on`. Whenever it generates a <> or a `[]`, LP checks that the next nonblank line in the `.lp` file contains a corresponding <> or `[]` command. The special LP prompts `<>?` and `[]?` indicate that LP expects a confirming <> or `[]` in the `.lp` file. LP prints an error message if the confirming <> or `[]` is missing, or if an unexpected <> or `[]` appears in the `.lp` file.

Regardless of whether box-checking is on or off, LP does not copy <> and `[]` commands from its input to the history or to a script file. Instead, it puts into the history and script file the <> and `[]` commands that it produces as it creates and discharges subgoals. Thus, the history and the script file will be annotated in a way that correctly reflects the actual progress of the proof.

See also the qed command.

6 Other commands

LP is a command-driven system. Commands can be entered in upper, lower, or mixed case. They, and some of their arguments, can be abbreviated by unambiguous prefixes of their hyphen-separated components. LP prompts users for any missing arguments that it requires to execute a command. The syntax of each command is illustrated and described more fully in the description for that command.

Commands for creating axioms and facts

`assert` **<assertion>+;** [;] Assert axioms

`declare operators` **<opdec>+**[,] Declare operators

`declare sorts` **<sort>+,** Declare sorts

`declare variables` **<vardec>+**[,] Declare variables

`make` **<fact-status>** (**<names>** | `conjecture`) Change the activity or immunity of facts or conjecture

Forward inference commands

`apply` **<names>** `to` **<names>** Apply the named deduction rules to the named facts

`complete` Run the Knuth-Bendix completion procedure

`critical-pairs` **<names>** `with` **<names>** Find critical pair equations between each rewrite rule in the first named set and each in the second

`fix` **<variable>** `as` **<term>** `in` **<names>** Eliminate one existential quantifier in the named facts, replacing the quantified variable by a term

`instantiate` (**<variable>** `by` **<term>**)+, `in` **<names>** Instantiate variables and/or eliminate universal quantifiers in the named facts, replacing the free and quantified variables by the terms

`normalize` **<names>** [`with` [`reversed` <names>]] Normalize the named facts using the (reversals of the) hardwired and named rewrite rules

`rewrite` **<names>** [`with` [`reversed` <names>]] Rewrite some subterm of each named fact using a hardwired or (the reversal of) a named rewrite rule

Backward inference commands

`apply` **<names>** `to conjecture` Attempt to prove the current conjecture using the named deduction rules

`cancel` [`all` | `lemma`] Cancel the current conjecture [and others]

`normalize conjecture` [`with` **<names>**] Normalize the current conjecture using all hardwired and named rewrite rules

`prove` **<conjecture>** [`by` **<proof-method>**] Attempt to prove the conjecture using <proof-method>

`qed` Check that all conjectures have been proved

`resume` [`by` **<proof-method>**] Resume work on the current conjecture using <proof-method>

`rewrite conjecture` [`with` [`reversed` <names>]] Rewrite some subterm of the current conjecture using some hardwired or named rewrite rule

<> Confirm the start of a subgoal in a proof

`[]` Confirm the conclusion of a step in proof

Commands for user interaction

`clear` Discard all information except global settings

`delete` **<names>** Delete the named facts

`define-class` **$<class>** **<names>** Define an abbreviation for <names>

`display` [**<information-type>** [<names>]] Print information about the named facts

`execute` **<file>** Execute commands from <file>`.lp`

`execute-silently` **<file>** Same as `execute`, but suppressing all output

`forget` [`pairs`] Discard information to save space `freeze` **<file>** Save the state of LP in <file>`.lpfrz`

`help` **<topic>** Print help about the topics

`history` [**<number>** | `all`] Print a history of [the <number> most recent] commands

`pop-settings` Restore the values of local LP settings

`push-settings` Remember the values of local LP settings

`quit`, `q` Exit from LP

`set` Print the current values of all LP settings

`set` **<setting-name>** Print the current value of a setting and prompt for a new value

`set` **<setting-name>** **<setting-value>** Change the value of a setting

`show normal-form` **<term>** Display the reduction of a term to normal form

`show unifiers` **<term>**, **<term>** Display all unifiers of two terms

`statistics` [**<statistics-option>**] Print statistics on runtime, storage, and rule usage

`stop` Stop execution of command files

`thaw` **<file>** Restore a frozen state from <file>`.lpfrz`

`unset` [**<setting>** | `all`] Reset setting to its default value

`version` Display information about the current version of LP

`write` **<file>** [**<names>**] Write the registry and the named facts to <file>`.lp`

`%` **<comment>** Record a comment in the log and/or script file

Commands to control ordering

`order` [**<names>**] Orient the named formulas into rewrite rules

`register` **<ordering-constraints>** Provide constraints for orienting formulas

`unorder` [**<names>**] Turn the named rewrite rules back into formulas

`unregister` **<ordering-information>** Cancel the constraints for orienting formulas

6.1 The clear command

The *clear command* resets the entire state of LP.

Syntax

```
<clear-command> ::= clear
```

Examples

```
clear
```

Usage

The `clear` command causes LP to discard all information except for the values of the global settings.

6.2 The comment command

The *comment command* provides a way for users to annotate scripts and logs.

Syntax

```
<comment-command> ::= % <string>
```

Examples

```
% Axioms for finite sets
```

Usage

LP ignores all input from the character (`%`) introducing a comment to the end of the line. These characters appear in any script or log file active at the time of the command, but do not otherwise affect LP's operation.

6.3 The define-class command

The *define-class command* introduces an abbreviation for a set of named facts. This abbreviation can be used wherever <names> are required as an argument to a command.

Syntax

```
<define-class-command> ::= define-class <class-name> [ <names> ]
```

Examples

```
define-class $facts nat, set
define-class $facts1 copy($facts)
define-class $old eval(*)
```

Usage

The `define-class` command defines <class-name> as an abbreviation for <names>. If no <names> are specified, the command prints the current definition of <class-name>. The examples define the following abbreviations:

- `$facts` is an abbreviation for `nat, set`. The command `display $facts` is equivalent to the command `display nat, set`.
- `$facts1` is also an abbreviation for `nat, set`. Without the `copy` operation, it would be an abbreviation for `$facts`, whose meaning would change if `$facts` was redefined.

- `$old` is an abbreviation for the list of names of all facts in LP's logical system when the `define-class` command was executed; the `eval` operation forces the evaluation of its argument `*`. The command `display $old` displays the current forms of any of those facts that are still in LP's logical system when the `display` command is executed.

See also <name>.

6.4 The delete command

The *delete command* discards facts from LP's logical system.

Syntax

```
<delete-command> ::= delete <names>
```

Examples

```
delete rewrite-rules
delete myLemma, junk
```

Usage

The `delete` command deletes the named facts from the 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.

6.5 The display command

The *display command* displays information about LP's logical system.

Syntax

```
<display-command>  ::= display [ <information-type> ] [ <names> ]
<information-type> ::= classes | conjectures | facts | names
                         | ordering-constraints | proof-status | symbols
```

Examples

```
display
display *Hyp
display ordering-constraints contains-operator(+)
```

Usage

The `display` command displays information of the requested <information-type> about all facts and conjectures with names matching <names>. If <information-type> is omitted, it is assumed to be `facts`.

display classes Displays the definitions of all <class-name>s.

display conjectures [<names>] Displays all unproved conjectures [with the specified names].

display facts [<names>] Displays all facts in the current proof context [with the specified names]. Indicates immune facts by an `(I)` following their names, ancestor-immune statements by an `(i)`, and passive facts by a `(P)`.

display names Displays all name-prefixes introduced in the current proof context.

display ordering-constraints [<names>] Displays the registry for all operators [in the named facts] in the current proof context, unless the value of the ordering-method setting is **polynomial**, in which case it displays the polynomial interpretations of these operators.

display proof-status Displays the status of the current conjecture and all other conjectures for which it is a subgoal (of a subgoal ...).

display symbols [<names>] Displays all sorts, operators, and variables [in the named facts] in the current proof context. The command `display symbols` displays all symbols in the current proof context, whereas `display symbols *` only displays those that occur in some fact.

6.6 The execute command

The *execute command* causes LP to execute commands from a specified file.

Syntax

```
<execute-command> ::= ( execute | execute-silently ) <file>
<file>            ::= <blank-free-string>
```

Examples

```
execute myProof
```

Usage

The `execute` command causes LP to execute the commands in the file named <file>`.lp` (unless <file> contains a period, in which case LP does not supply the suffix `.lp`) on the current LP search path. The `execute` command is ordinarily used to execute commands from a file that was created by the set script or write commands, but any text file may be specified.

Further `execute` commands may occur in `.lp` files, but recursive `.lp` files are not allowed. Once a `.lp` file has been exhausted, LP resumes accepting input from the previous `.lp` file or from the user if no other file is being executed. If LP encounters an error while executing a file, or if the user interrupts LP, LP takes subsequent input from the terminal.

The `execute-silently` command is just like `execute`, except that it produces no output.

6.7 The forget command

The *forget command* causes LP to discard certain information about which computations LP has already performed.

Syntax

```
<forget-command> ::= forget [ pairs ]
```

Note: The argument of the `forget` command can be abbreviated.

Examples

```
forget
```

Usage

The `forget` command causes LP to discard all information about which critical pairs have already been computed in the current proof context. It also prevents LP from accumulating further such information in the current proof context, and in all subsequently created subcontexts, until the next complete command is given.

The `forget` command can save significant space when there are many rewrite rules. It is useful when we are interested proving conjectures without appealing to the critical-pairs or complete commands.

6.8 The freeze and thaw commands

The *freeze* and *thaw commands* enable LP to save and restore its state.

Syntax

```
<freeze-command> ::= (freeze | freeze-current) <file>
<thaw-command>   ::= thaw <file>
```

Examples

```
freeze case1
thaw case1
```

Usage

The **freeze** command saves LP's entire state, excluding the statistics and global settings, but including all proof contexts, in the file named <file>**.lpfrz** (unless <file> contains a period, in which case LP does not supply the suffix **.lpfrz**) in LP's current working directory. If a file with that name already exists, its previous contents are erased; if it does not exist, it is created.

The **freeze-current** command does the same, except that it only saves the state of the current proof context. This command is faster and uses less disk space than the **freeze** command. It is useful primarily for trying different strategies for proving the current conjecture.

The **thaw** command restores the state of LP from that saved previously using the **freeze** command in the file named <file>**.lpfrz** (unless <file> contains a period, in which case LP does not supply the suffix **.lpfrz**) on the current LP search path. The **thaw** command will not thaw files that were frozen using an out-of-date version of LP.

These commands are useful for checkpointing attempted proofs. They are also useful for saving and restoring completed or partially-completed systems.

See also

- Forgetting information to save space before a `freeze`
- Writing commands into a file to recreate the current proof context

6.9 The help command

The *help command* provides information about the use of LP. This same information can also be viewed using a hypertext reader such as Mosaic, starting at the home page in ~`lp/help/overview.html`.

Syntax

```
<help-command> ::= help <topic>
<topic>        ::= <blank-free-string>
```

Examples

```
help ?
help commands
```

Usage

The `help` command provides a detailed explanation of the requested topic, which can be specified by an unambiguous prefix. The command `help ?` produces a terse list of topics for which help is available. The command `help commands` produces a summary of the LP commands. The command `help lp` produces a list of general topics for which help is available.

If you don't get information on the topic you expected after typing a command like `help rewrite`, try typing `help rewrite-` to see if there is further information about related topics (e.g., `rewrite-command` or `rewrite-rule`).

6.10 The history command

The *history command* produces a list of the commands executed by LP.

Syntax

```
<history-command> ::= history [ <number> | all ]
```

Examples

```
history
history 10
history all
```

Usage

When supplied with an argument, the **`history`** command prints a list of the <number> most recent commands executed by LP or of all commands executed by LP. When not supplied with an argument, it behaves in the same fashion as the last **`history`** command (or as **`history all`** if it is the first **`history`** command).

LP annotates the history by commenting out erroneous commands, by marking the beginning and end of executed files, by marking the creation of subgoals and the completion of proofs, and by indenting the history to reveal its proof structure.

After a thaw command, LP's current history is replaced by the history that led to the corresponding freeze. Thus a script file provides a record of the commands executed during the current LP session, and the history provides a record of commands that will recreate LP's current state.

Users who want a script that will recreate LP's current state, but who have forgotten to issue a set script command, can issue a set log command instead followed by a **`history all`** command to capture the script in the log file.

6.11 The push-settings and pop-settings commands

The *push-settings* and *pop-settings commands* save and restore the values of the settings that control LP's operation.

Syntax

```
<push-settings-command> ::= push-settings
<pop-settings-command>  ::= pop-settings
```

Examples

```
push-settings
pop-settings
```

Usage

The **`push-settings`** command saves the current values of all local settings by pushing them on a stack. The **`pop-settings`** command restores the values of these settings by popping them off the stack.

The write command places the `push-settings` and `pop-settings` commands in `.lp` files so that named axioms can be loaded from these files without affecting the current name-prefix, activity, and immunity settings.

6.12 The quit command

The *quit command* causes LP to terminate.

Syntax

```
<quit-command> ::= quit | q
```

Examples

```
quit
```

Usage

The `quit` command causes LP to halt, returning the user to the operating system. Any script or log file is closed.

6.13 The set command

The *set command* controls various aspects of LP's behavior.

Syntax

```
<set-command>  ::= set [ <setting-name> [ <setting-value> ] ]
<setting-name> ::= activity | automatic-ordering | automatic-registry
                     | box-checking | completion-mode | directory
                     | display-mode | hardwired-usage | immunity
                     | log-file | lp-path | name-prefix | ordering-method
                     | page-mode | prompt | proof-methods
                     | reduction-strategy | rewriting-limit | script-file
                     | statistics-level | trace-level | write-mode
<setting-value> ::= <string>
```

Note: The `setting-name` can be abbreviated.

Examples

```
set
set proof-methods
set script session
```

Usage

The `set` command with no arguments prints the current values of all settings. The `set` command with a <setting-name> and no <setting-value> displays the current value of the setting and then requests a new value; responding with a carriage return leaves the value of the setting unchanged. The `set` command with both a <setting-name> and a <setting-value> sets the value of one of the following settings.

Settings marked with `(L)` are local to the current proof context. If such a setting is changed, for example, during the proof of one case in a proof by cases it reverts to its previous value upon termination of that case in the proof. Settings marked with `G` are global and remain in effect until changed by the user. All settings have default values, which can be restored by the unset command.

`(L) activity ( on | off )` New assertions are active if `on` (the default).

`(L) automatic-ordering ( on | off )` Formulas are oriented automatically into rewrite rules if `on` (the default).

(L) automatic-registry (on | off) The registry is extended automatically during ordering if on (the default).

(G) box-checking (on | off) Markings (<>, []) for proof steps are checked in command files if on (the default).

(L) completion-mode (big | expert | standard) Controls the action of the complete command (default standard).

(G) directory <file> Output files are created in this directory (default ".").

(L) display-mode (qualified | unqualified | unambiguous) Controls the qualification of identifiers and terms by the display command (default unambiguous).

(L) hardwired-usage <number> Turns off some hardwired rules.

(L) immunity (on | off | ancestor) Controls the immunity of new assertions (default off).

(G) log-file <file> Creates file <file>.lplog for log of session (default none).

(G) lp-path <string> Defines search path for help, .lp, .lpfrz files (default . ~/lp).

(L) name-prefix <simpleId> Defines prefix for names of assertions, conjectures (default user).

(L) ordering-method <ordering> Defines method for orienting formulas into rewrite rules (default noeq-dsmpos).

(G) page-mode (on | off) Output is paged if on (default off).

(G) prompt <string> Defines LP's command prompt (default 'LP! ').

(L) proof-methods <default-proof-method>+[,] Defines list of default proof methods for conjectured formulas (default normalization).

(L) reduction-strategy (inside-out | outside-in) Controls application of rewrite rules to terms (default outside-in).

(L) rewriting-limit <number> Bounds number of steps in possibly infinite rewrites (default 1000).

(G) script-file <file> Creates file <file>.lpscr for record of input (default none).

(G) statistics-level <number> Controls the kinds of statistics kept by LP (default 2).

(G) trace-level <number> Controls the amount of detailed output produced by LP (default 1).

(L) write-mode (qualified | unqualified | unambiguous) Controls the qualification of identifiers and terms by the write command (default qualified).

6.13.1 The activity setting

LP automatically uses all *active* rewrite rules for normalization and all active deduction rules for deduction. By contrast, it uses *passive* facts only upon explicit user command. The activity of newly asserted or conjectured facts is governed by the *activity* setting. The activity of previously asserted or conjectured facts can be changed by the make command. Unless instructed otherwise, LP treats all facts as active.

Syntax

```
<set-activity-command> ::= set activity ( on | off )
```

Examples

```
set activity off
```

Usage

The `activity` setting applies to facts and conjectures created by the apply, assert, critical-pairs, fix, instantiate, and prove commands in the current proof context. Such facts are *active* if the setting is `on` (the default) and *passive* (or *inactive*) if it is `off`. Passive facts are indicated by a parenthesized letter `(P)` in the output of the display command.

LP automatically uses all active rewrite rules to reduce terms to normal form, and it automatically uses all active deduction rules to deduce consequences from formulas and rewrite rules. LP does not make automatic use of passive facts. Instead, LP applies passive rewrite rules only in response to the normalize or rewrite commands, and it applies passive deduction rules only in response to the apply command.

Facts retain their activity or passivity when they are normalized, and formulas retain their activity or passivity when oriented into rewrite rules. Likewise, passive conjectures remain passive when proved.

There are two main uses for passive facts in LP.

- Users can improve the performance of LP by declaring facts to be passive when they are known to be inapplicable. When LP subsequently adds new formulas to the system, it will not attempt to reduce them by passive rules or to apply passive deduction rules to them.
- Users can control the application of problematic rewrite and deduction rules by declaring them to be passive. For example, users may wish to apply a complete set of boolean reductions when they believe it will simplify a formula (e.g., to `true`), but they may be reluctant to have LP apply those reductions automatically to all formulas (lest the reductions produce large, unreadable intermediate forms).

6.13.2 The automatic-ordering setting

The *automatic-ordering* setting controls whether LP orients formulas into rewrite rules with or without the user having to issue an explicit order command.

Syntax

```
<set-automatic-ordering-command> ::= set automatic-ordering ( on | off )
```

Examples

```
set auto-ord off
```

Usage

If `automatic-ordering` is `on` (the default), then LP attempts to orient all formulas (asserted axioms, proved conjectures, results of deductions, instantiations, critical pair equations, and hypotheses for subgoals in proofs) automatically into rewrite rules. If it is `off`, then the user must issue an explicit `order` command to orient formulas into rewrite rules. The value of this setting is local to the current proof context.

6.13.3 The automatic-registry setting

The *automatic-registry* setting controls whether LP asks the user to confirm extensions to the registry when formulas are oriented into rewrite rules.

Syntax

```
<set-automatic-registry-command> ::= set automatic-registry ( on | off )
```

Examples

```
set auto-reg off
```

Usage

If `automatic-registry` is `on` (the default), then LP automatically extends its registry, if necessary, when using a registered ordering to orient formulas into rewrite rules. If it is `off`, LP prompts the user to select one of a list of possible extensions whenever the registry must be extended. When it is `off`, LP also gives users the opportunity to divide incompatible equations and to orient formulas manually.

6.13.4 The box-checking setting

The *box-checking* setting governs whether or not LP checks for the presence of annotations marking the beginning and end of proofs steps in files being executed by the execute command.

Syntax

```
<set-box-checking-command> ::= set box-checking ( on | off )
```

Examples

```
set box on
```

Usage

LP checks <>'s and `[]`'s when it executes commands from a `.lp` file and `box-checking` is `on`. It ignores these annotations when `box-checking` is `off`.

See the box and diamond commands for details.

6.13.5 The completion-mode setting

The *completion-mode setting* sets the operating mode for the completion procedure.

Syntax

```
<set-completion-mode-command> ::= set completion-mode <completion-mode>
<completion-mode>             ::= standard | expert | big
```

Note: The <completion-mode> can be abbreviated.

Examples

```
set completion-mode expert
```

Usage

The `completion-mode` setting affects the order in which completion tasks are executed in the current proof context. It also affects the amount of user interaction during the completion procedure.

- The *standard* mode (the default) computes critical pair equations before extending the registry. When the automatic-registry setting is `off`, this reduces the amount of interaction required from the user to extend the registry; however, it can be inefficient.
- The *expert* mode gives higher priority to orienting formulas into rewrite rules than to computing critical pair equations. When the automatic-registry setting is `off`, this gives the user more explicit control over the completion process by prompting for guidance more often. In particular, user interaction to order unordered formulas is requested before the system attempts to compute new critical pairs.
- The *big* mode postpones the computation of critical pairs even farther, so that big formulas are examined before critical pairs are computed.

6.13.6 The directory setting

The *directory setting* sets the working directory for LP.

Syntax

```
<set-directory-command> ::= set directory <file>
```

Examples

```
set directory ~/proofs
```

Usage

The working directory for LP is the directory in which it creates files in response to the freeze, set log, set script, and write commands. A period (`.`) in the setting of lp-path refers to this directory.

The default working directory is the directory in which LP was invoked. The `set directory` command changes the working directory in all proof contexts.

6.13.7 The display-mode setting

The *display-mode setting* controls the manner in which the `display` command outputs identifiers and terms.

Syntax

```
<set-display-command> ::= set display-mode <qualification-mode>
<qualification-mode>  ::= qualified | unambiguous | unqualified
```

Note: The <qualification-mode> can be abbreviated.

Examples

```
set display-mode qualified
```

Usage

The `display-mode` setting controls the ouput of identifiers and terms by the display command in the current proof context.

```
display-mode      effect
------------      ------
qualified         print qualifications for all subterms, identifiers
unqualified       print no qualifications
unambiguous       print enough qualifications to enable reparsing
```

The default display-mode is `unambiguous`.

See also

- Overloaded identifiers
- The write-mode setting

6.13.8 The hardwired-usage setting

The *hardwired-usage setting* provides users with limited control over LP's use of hardwired rewrite and deduction rules.

Syntax

```
<set-hardwired-usage-command> ::= set hardwired-usage <number>
```

Examples

```
set hardwired-usage 8
```

Usage

LP hardwires selected rewrite rules for the logical and conditional operators. For debugging purposes, the `set hardwired-usage` command can be used to turn off some of these hardwired rewrite rules in the current proof context. The following powers of 2, if they occur in the binary representation of <number>, turn off the indicated rule.

2 Turns off the rewrite rule `x => false -> ~x`

4 Turns off the rewrite rule `x <=> false -> ~x`, but adds the rewrite rule `x <=> y <=> ~y -> ~x`.

8 Turns off the rewrite rules with left side `if x then y else z` when `y` or `z` is `true` or `false`

16 Turns off the if-simplification metarule

6.13.9 The immunity setting

LP automatically subjects all *nonimmunized* facts in its logical system to normalization and deduction. By contrast, it subjects *immune* facts to these operations only upon explicit user command. The immunity of newly asserted or conjectured facts is governed by the *immunity* setting. The immunity of previously asserted or conjectured facts can be changed by the make command. Unless instructed otherwise, LP does not treat any fact as immune.

Syntax

```
<set-immunity-command> ::= set immunity ( on | off | ancestor )
```

Note: The `immunity` setting can be abbreviated.

Examples

```
set immunity on
```

Usage

The *immunity* setting applies to facts and conjectures created by the apply, assert, critical-pairs, fix, instantiate, and prove commands in the current proof context. Such facts are *immune* if the setting is `on`, *nonimmune* if it is `off`, and *ancestor-immune* if it is `ancestor`. Immune facts are indicated by a parenthesized letter `(I)` in the output of the display command; ancestor-immune facts are indicated by `(i)`.

LP automatically reduces all terms in nonimmune formulas, rewrite rules, and deduction rules to normal form, and it automatically applies all active deduction rules to all nonimmune formulas and rewrite rules. LP behaves differently, however, with respect to immune and ancestor-immune facts.

- LP does not automatically reduce immune facts to normal form, and it does not automatically apply deduction rules to them. Instead, LP reduces them only in response to the normalize or rewrite commands, and it uses them for deduction only in response to the apply command.
- LP does not automatically reduce an ancestor-immune fact by any ancestor of that fact, and it does not automatically apply a deduction rule that is an ancestor of a fact to that fact. One fact is an *ancestor* of another if its name is a prefix of the other's. Thus, if rewrite rule `a.1.2.3` is ancestor-immune, it will not be reduced by rewrite rule `a.1.2` (from which it may have been obtained by instantiation), and it will not have deduction rule `a.1` (from which `a.1.2` may have been obtained by instantiation) applied to it.

Facts retain their immunity when they are normalized, and formulas retain their immunity when oriented into rewrite rules. Likewise, immune conjectures may be reduced during an attempt to prove them, but are added to the system as immune facts in their original form when proved.

There are several reasons to make facts immune or ancestor-immune in LP.

- Immune facts retain their original form and may be more readable than reduced versions of those facts.
- Immune rewrite rules may be useful in critical-pair computations.
- Setting immunity `on` or to `ancestor` makes it possible to preserve instantiations that might otherwise normalize to identities.
- Users can improve the performance of LP by declaring facts to be immune when they are known to be irreducible. When LP subsequently adds new rewrite rules to the system, it will not attempt to reduce the immune facts using these rules.

However, there are also disadvantages to making too many rules immune.

- The presence of immune rules can degrade the performance of LP, because additional rewrite rules increase the cost of normalization. It can also increase the amount of nonconfluence in a rewriting system.
- An immune deduction rule with a reducible hypothesis may not match formulas as expected, because the deduction rule is applied only after the formulas have been normalized.

6.13.10 The log-file setting

The *log-file setting* provides a means of recording the current LP session in a file.

Syntax

```
<set-log-file-command> ::= set log-file <file>
```

Examples

```
set log session
```

Usage

The `set log` command causes LP to start recording the terminal session in a file named <file>`.lplog` (unless <file> contains a period, in which case LP does not supply the suffix `.lplog`) in LP's current working directory. Any previous contents of this log file are lost. If LP was already logging to a file, that file is closed before opening the new log file. Logging is ended by the quit or unset log commands.

6.13.11 The lp-path setting

The *lp-path setting* specifies a list of directories for LP to search when looking for input from a file.

Syntax

```
<set-lp-path-command> ::= set lp-path <string>
```

Examples

```
set lp-path . ~/myAxioms ~lp
```

Usage

The `set lp-path` command specifies a search path for LP to use when looking for help, `.lp`, or `.lpfrz` files. Its default value is "`. ~lp/axioms`". A period (`.`) in the value of `lp-path` refers to LP's current working directory. A tilde (~) in the value of `lp-path` refers to the user's home directory. An initial directory `~lp` refers to the directory in which LP's runtime support was installed; see the installation instructions, the command-line options, and the version command for the location of this directory.

6.13.12 The name-prefix setting

The *name-prefix setting* specifies the identifier used to construct names for newly asserted facts and conjectures.

Syntax

```
<set-name-prefix-command> ::= set name-prefix <simpleId>
```

Examples

```
set name nat
```

Usage

The `set name-prefix` command directs LP to use the <simpleId> as the prefix for any new names generated in the current proof context. After a command such `set name nat`, LP assigns the names `nat.1`, `nat.2`, ... in sequence to facts and conjectures.

6.13.13 The ordering-method setting

The *ordering-method setting* specifies the method used to orient formulas into rewrite rules.

Syntax

```
<set-ordering-method-command> ::= set ordering-method <ordering>
<ordering>                    ::= <registered-ordering>
                                    | either-way | left-to-right
                                    | manual | polynomial [ <number> ]
<registered-ordering>         ::= dsmpos | noeq-dsmpos
```

Note: The <ordering> can be abbreviated.

Examples

```
set ordering dsmpos
set ordering polynomial 2
```

Usage

The `set ordering-method` command sets the method used to orient formulas into rewrite rules in the current proof context. The methods based on the registered and polynomial orderings attempt to guarantee that the resulting set of rewrite rules is terminating. The other brute-force orderings give users more control over the direction in which equations are oriented into rewrite rules, but they do not guarantee termination. The registered orderings are the easiest to use. The default ordering method is `noeq-dsmpos`.

When a set of rewrite rules is known to terminate (because of the ordering used to orient them), but the new ordering does not establish termination, LP issues a warning that the termination proof will be lost when the next rewrite rule is added to the system.

6.13.14 The page-mode setting

The *page-mode setting* controls whether or not LP pauses to enable users to read its output before it is scrolled off the screen.

Syntax

```
<set-page-mode-command> ::= set page-mode ( on | off )
```

Examples

```
set page-mode on
```

Usage

When `page-mode` is `off` (the default), LP does not pause during output. When it is `on`, LP displays output a screen at a time. After LP displays each screenful of output, it prompts the user with `--More--` to type a character indicating what to do next. The options are as follows:

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

Most Unix systems also allow users to control output is by using the `^S` and `^Q` keys; `^S` stops output, and `^Q` resumes printing.

6.13.15 The prompt setting

The *prompt setting* defines a string that LP uses to prompt users to enter a command.

Syntax

```
<set-prompt-command> ::= set prompt <prompt>
<prompt>             ::= <blank-free-string> | ‘ <string> ’
```

Examples

```
set prompt ‘[!] ’
set prompt ‘>> ’
```

Usage

By default, <prompt> is `‘LP!:  ’`, which causes LP to issue prompts of the form ”`LP1:  `”, ”`LP2:  `”, ... To enter a prompt that begins or ends with a space, enclose it within `‘’` marks.

LP replaces the first exclamation mark (`!`) in <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 script (`.lp`) file by appending a period followed by consecutive integers to the number of the execute command; thus command 5.2.3 is the third command in the script file executed in response to the second command in the script file executed in response to the fifth command typed by the user.

6.13.16 The proof-methods setting

The *proof-methods* setting provides a list of default proof methods for LP to use when attempting to prove a formula.

Syntax

```
<set-proof-methods-command> ::= set proof-methods <default-proof-method>+[,]
```

Note: Each <default-proof-method> can be abbreviated.

Examples

```
set proof =>, normalization
```

Usage

The `set proof-methods` command provides a list of default proof methods for the current proof context. LP uses the first method in the list that applies to the conjecture. The default list is `normalization`. Any method (other than `contradiction`) that does not mention a variable or constant can appear on the list. If the proof-method list is `explicit-commands`, then LP will await a resume command before beginning the proof.

See also

- The prove command
- The /\-method

- The =>-method
- The <=>-method
- The if-method
- The explicit-commands method
- The normalization method

6.13.17 The reduction-strategy setting

The *reduction-strategy setting* controls the order in which LP applies rewrite rules when reducing a term.

Syntax

```
<set-reduction-strategy-command> ::= set reduction-strategy
                                       ( inside-out | outside-in )
```

Note: The `reduction-strategy` can be abbreviated.

Examples

```
set reduction in
```

Usage

When the `reduction-strategy` is `outside-in` (the default), LP attempts to reduce a term by attempting to rewrite the entire term before attempting to reduce any of its subterms. If it is `inside-out`, LP still applies the hardwired rewrite rules outside-in, but it attempts to apply other rewrite rules to the subterms of a term before it applies them to the entire term.

The `reduction-strategy` is local to the current proof context.

6.13.18 The rewriting-limit setting

The *rewriting-limit setting* 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.

Syntax

```
<set-rewriting-limit-command> ::= set rewriting-limit <number>
```

Examples

```
set rewriting-limit 50
```

Usage

The `rewriting-limit` setting is local to the current proof context. Its default value is 1000.

If LP exceeds the rewriting limit when normalizing a formula, rewrite rule, or deduction rule, it immunizes that fact. If it exceeds the rewriting limit when attempting to prove a conjecture by normalization or rewriting, the user can continue normalizing the conjecture by typing resume (after raising the rewriting limit, if desired).

6.13.19 The script-file setting

The *script-file setting* provides a means of recording user input in a file.

Syntax

```
<set-script-file-command> ::= set script-file <file>
```

Examples

```
set script session
```

Usage

The `set script` command causes LP to start recording user input in a file named <file>`.lpscr` (unless <file> contains a period, in which case LP does not supply the suffix `.lpscr`) in LP's current working directory. Any previous contents of this log file are lost. If LP was already scripting to a file, that file is closed before opening the new one is opened. Scripting is ended by the quit or unset script command, which is not recorded in the script file.

LP annotates the script file by commenting out erroneous 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 to reveal its proof structure.

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).

See also

- The history command

6.13.20 The statistics-level setting

The *statistics-level setting* controls the amount of statistics that LP records about its operation.

Syntax

```
<set-statistics-level-command> ::= set statistics-level <number>
```

Examples

```
set statistics-level 3
```

Usage

The `statistics-level` setting is an integer between 0 and 3; 2 is the default. LP gathers increasingly many statistics at each level, as follows.

- **0** Summary statistics only: total running time and memory usage, including the number of garbage collections.
- **1** Detailed statistics about the time spent, both successfully and unsuccessfully, attempting to orient formulas into rewrite rules, to apply rewrite rules, to apply deduction rules, and to unify terms (in response to the critical-pairs command). Also, the time spent controlling LP's inference mechanisms.
- **2** The number of successful applications of each rewrite and deduction rule, as well as the number of nontrivial critical pairs involving each rewrite rule.
- **3** The number of attempted applications of each rewrite rule. Level 3 imposes a considerable computational burden, for example, up to 10% in some applications.

6.13.21 The trace-level setting

The *trace-level setting* controls the amount of detail that LP outputs concerning its operation.

Syntax

```
<set-trace-level-command> ::= set trace-level <number>
```

Examples

```
set trace-level 3
```

Usage

The `trace-level` setting is an integer between 0 and 8; 1 is the default. LP prints increasing amounts of information at each level, as follows.

- **0** Only user interactions and the final results of commands.
- **1** The creation and deletion of facts: how many new facts have been asserted, when facts are deleted because they reduce to `true`, when application of a deduction rule yields a nontrivial result, when a nontrivial critical pair or instantiation has been added to the system, and when a deduction rule is normalized to a set of formulas.

  Also, the size of the system at regular intervals, and when a critical pair computation is abandoned because a theorem has been proved.
- **2** Ordering actions: when a formula is oriented into a rewrite rule, when a rewrite rule is turned back into a formula because its left side was reduced, when the registry is extended, and when an incompatible formula cannot be oriented.

  Also, the size of the system at more frequent intervals.

3 Reduction actions: a formula, deduction rule, or the right side of a rewrite rule has been reduced as a result of adding a new rewrite rule to the system.

Also, the accumulated running time at periodic intervals.

4 Internormalization actions: processing new facts (by normalizing them and applying deduction rules), applying new rewrite rules, orienting formulas into rewrite rules, and computing critical pairs during completion. Also, postponing the orientation of formulas because of their size or because they cannot be oriented using the current registry.

5 Detailed information about critical pairs, deduction rules, and instantiations: instantiations that leave a formula, rewrite rule, or deduction rule unchanged; instantiations and the results of deduction rules that reduce to `true`, and unreduced critical pair equations along with their normal forms.

6 Successful application of a rewrite rule (debugging information). The output produced at this and higher trace levels is not particularly well coordinated with the output produced by lower trace levels.

7 Attempted application of a rewrite rule (debugging information).

8 Times at which the events reported at levels 6 and 7 occur.

6.13.22 The write-mode setting

The *write-mode setting* controls the manner in which the `write` command outputs identifiers and terms.

Syntax

```
<set-write-mode-command> ::= set write-mode <qualification-mode>
```

Examples

```
set write-mode qualified
```

Usage

The `write-mode` setting controls the output of identifiers and terms by the write command in the current proof context.

```
write-mode        effect
----------        ------
qualified         print qualifications for all subterms, identifiers
unqualified       print no qualifications
unambiguous       print enough qualifications to enable reparsing
```

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 a new copy of LP, executing the `.lp` file, and writing it out again in `qualified` mode.

See also

- Overloaded identifiers
- The display-mode setting

6.14 The show command

The *show command* displays the results of certain operations without affecting the state of LP's logical system.

Syntax

```
<show-command> ::= show normal-form <term>
                 | show unifiers  <term>,  <term>
```

Note: The first argument of the `show` command can be abbreviated.

Examples

```
show n-f e \in (s \U s)
show unifiers e*x, i(y)*y
```

Usage

The `show normal-form` command displays a normal form of the term with respect to the currently active rewrite rules. If the trace-level is nonzero, LP also displays successive reductions of the term leading to the normal form.

The `show unifiers` command displays a complete set of most general unifiers of two terms. It displays the unifying substitutions along with the unifications of the terms, and it uses unification algorithms appropriate to the theories associated with the operators in the terms.

6.15 The statistics command

The *statistics* command displays information about the resources consumed by LP, as well as about the usage of rewrite and deduction rules.

Syntax

```
<statistics-command> ::= statistics [ <statistics-option> ]
<statistics-option>  ::= time | usage [ <names> ]
```

Note: The first word of the <statistics-option> command can be abbreviated.

Examples

```
statistics
stat usage nat
```

Usage

The `statistics` command displays cumulative and recent (since the last display) information about LP's operation. The default option is `time`, which displays information about the time used by LP together with the current size of the heap, the amount of free space available, and the number of garbage collections that have occurred.

The `usage` option displays information about the use of the rewrite and deduction rules matched by <names>, which defaults to "*". For rewrite rules, the information includes the number of successful applications, the number of attempted applications, and the number of nontrivial critical pairs computed from the rule. For deduction rules, the information includes the number of successful applications. In the display, each name is preceded by `(rr)` or `(dr)` to indicate whether the named fact is a rewrite rule or a deduction rule. If two or more facts had the same name, then the display show separate statistics for each incarnation of the name (for example, if two theorems, `thm.1` and `thm.2`, were proved by cases, then two separate rewrite rules would have received the name `thmCaseHyp1.1`).

See also

- The statistics-level setting

6.16 The stop command

The *stop command* returns control over LP to the user.

Syntax

```
<stop-command> ::= stop
```

Examples

```
stop
```

Usage

The `stop` command causes LP to stop executing commands from files, thereby enabling the user to enter commands again from the terminal.

6.17 The unset command

The *unset command* returns settings to their default values.

Syntax

```
<unset-command> ::= unset ( <setting-name>  | all )
```

Note: The argument to the `unset` command can be abbreviated.

examples

```
unset script
unset all
```

The `unset` command sets the value of <setting-name> to its default value. In particular, `unset script` stops recording user input in a script file and `unset log` stops recording the session in a log file. The `unset all` command sets the value of all settings to their defaults.

See also

- The set command for a description of the legal settings and their default values

6.18 The version command

The *version command* redisplays the identifying information that LP prints when it starts up.

Syntax

```
<version-command> ::= version
```

Examples

```
version
```

Usage

The `version` command causes LP to identify which version of LP is being run and how it was installed. In particular, it reports the maximum size for LP's heap (which limits the size of LP's logical system and proof stack) and the identity of the directory `~lp` that LP searches for help messages and examples.

6.19 The write command

The *write command* creates a file containing commands that can be executed to recreate LP's current logical system.

Syntax

```
<write-command> ::= write <file> [ <names> ]
```

Examples

```
write axioms
write intSet int, set
```

Usage

The `write` command creates a file named <file>`.lp` (unless <file> contains a period, in which case LP does not supply the suffix `.lp`) in LP's current working directory. This file can be executed later with the execute command to recreate the named set of facts (or the whole system, if no names are specified). In particular, LP writes declarations for all identifiers in the named set of facts followed by commands to register ordering constraints for the facts and to assert the facts.

Unlike the freeze command, the `write` command does not save information about the state of 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.

Rewrite rules written by the `write` command are asserted as formulas. The numeric extensions in the names assigned to facts may change when the `.lp` file is executed; hence facts that are ancestor-immune may behave differently when the system is recreated.

See also

- The write-mode setting
- The display command

7 Hints on using LP

There are a number of hints that beginning users of LP may find helpful. These hints fall into the following categories.

- Preparing input and recording work
- Formalizing axioms and conjectures
- Orienting formulas into rewrite rules
- Managing proofs
- Making proofs go faster

7.1 Preparing input and recording work

Use an editor to prepare a command file containing declarations and assertions. Then execute that file to check that LP can read what you have typed. If you have made any mistakes, edit the command file to fix them.

Put all the declarations you expect to need at the beginning of your command file. This allows LP to check your declarations before beginning any time consuming tasks.

Although proofs are usually constructed interactively, successful proofs should be recorded in cleaned-up command files. Structure your proofs using a sequence of execute commands.

Freeze LP's state often. This makes it easier to try different alternatives when looking for a proof.

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.

7.2 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 `\A x e(x)`. In the absence of other assertions involving `c`, the second only implies `\E x e(x)`. If you don't know whether an identifier is a variable or a constant, type display symbols to find out.

Be careful about the use of free variables in formulas. The formula `x = {} => subset(x, y)` correctly (albeit awkwardly) expresses the fact that the empty set is a subset of any set. However, its converse, `subset(x, y) => x = {}`, does not express the fact that any set that is a subset of all sets must be the empty set. That fact is expressed by the equivalent formulas `\A x (\A y subset(x, y) => x = {})` and `\A x \E y (subset(x, y) => x = {})`.

An axiom or conjecture of the form `when A yield B` has the same logical content as one of the form `A => B`, but different operational content. Given the axiomization

```
declare variable x: Bool
declare operators
  a: -> Bool
  f, g, h: Bool -> Bool
  ..
assert
  when f(x) yield g(x);
  g(x) => h(x);
  f(a)
  ..
```

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 formulas that are matched by `A` and `B`, LP will apply the multiple-hypothesis rule but not the single-hypothesis rule.

7.3 Orienting formulas into rewrite rules

If you put some well-selected ordering constraints in the registry, LP will orient formulas 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 defining formulas, such as `P(x) = P1(x) /\ P2(x)`, with the term being defined on the left side of the equality sign.

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

- Set automatic-registry `off`, instruct LP to order only the offending formula, and choose one of the presented suggestions that order the formula 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 formulas to be ordered as desired.
- Alternatively, rerun the proof with automatic-registry `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 `on`. This last step is important because proof scripts with automatic-registry `off` are not usually robust.

Occasionally, LP will fail to orient a set of formulas 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., the dsmpos ordering 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.

7.4 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, the => method, the <=> method, or the `if` method.

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 `true`), make them immune. Typing either of the commands

```
set immunity on                     prove ... by explicit-commands
prove ... by explicit-commands      make immune conjecture
set immunity off                    resume by ...
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, implication, or logical equivalence, try the corresponding proof method.
- Try computing critical pairs between hypotheses and other rewrite rules, for example, by typing the command `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 (by typing `display *Hyp`) and check whether any that you expected to see are missing or are not oriented 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.

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* displays 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 commands.

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 a formula, will often be enlightening in such cases. Setting the trace-level up to 6 will show which rewrite rules are applied in the normalization.

7.5 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 are 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 associative-commutative 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-pairs command. Also, try to avoid computing critical pairs between rewrite rules that contain subterms such as `t1 /\ t2 /\ ... /\ t5`, with multiple occurrences of the same associative-commutative operator.

7.6 Unix command line options

The following options can be specified on the Unix command line when invoking LP:

`-c` Enables experimental features for conditional rewriting.

`-d` *fileName* Specifies the location of LP's run-time library, to which `~lp` in the lp-path setting refers. The default is `/usr/local/lib/LP`.

`-e` Enables undocumented experimental features in LP.

-max_heap <number> Sets an upper bound, in megabytes, on the size of LP's heap. This bound should be large enough for LP top handle a proof without running its collector too often (for example, 10 meg on a 32-bit machine and 20-meg on a 64-bit machine). It should be small enough for LP to run without paging (for example, half of the total amount of memory on a single-user machine).

-min_gc <number> Sets the minimum threshhold, in megabytes, beneath which LP's collector will not run automatically.

-t Prevents LP from aborting execution of `.lp` files when an error occurs; useful for testing LP.

7.7 Obtaining LP

To use LP, you need to retrieve two files from `larch.lcs.mit.edu`:

- LP's run-time library lp.lib.tar.Z
- An executable version appropriate for your hardware:

```
lp.alpha.Z          DEC Alpha AXP running OSF/1
lp.decmips.Z        DECstation running Ultrix
lp.sparc.Z          Sparcstation running SunOS 4.1 or Solaris 5.3
```

These files are available both over the WWW and by anonymous ftp. To obtain them by anonymous ftp, type the following commands (with `platform` replaced by one of `alpha`, `decmips`, or `sparc`). Then uncompress the files you obtained, for example, by typing `uncompress *.Z`.

```
csh>       ftp larch.lcs.mit.edu
Name:      anonymous
Password:  your_email_address
ftp>       cd pub/Larch/LP
ftp>       bin
ftp>       get lp.lib.tar.Z
ftp>       get lp.platform.Z
ftp>       quit
```

See also

- Installation instructions
- Reporting problems
- News about new features, releases

7.8 Installing LP

To install LP, proceed as follows.

- Install the executable version of LP in some directory that occurs on your Unix search path before `/usr/bin` (which contains a Unix line printer utility also called `lp`). Remove the platform name as you do this, for example, by typing the command `mv lp.sparc /usr/local/bin/lp`. If necessary, use the `chmod` command to make this file executable.
- Unpack LP's run-time library by typing `tar xf lp.lib.tar`. This will create a directory named "`lp`*release*", where *release* identifies the version of LP supported by the library.

- If possible, make `/usr/local/lib/LP` a symbolic link to this directory, or copy the contents of this directory to `/usr/local/lib/LP`. If you cannot do this, alias the command `lp` to "`lp -d` *dir*", where *dir* is the name of the directory you created.
- If possible, copy the man page for LP, `LP/docs/lp.1`, to `/usr/man/man1`. This makes it possible for users to see the man page for LP by typing `man 1 lp`.

7.9 Reporting problems

Please report any bugs that you find in LP so that we can fix them. To do this, complete the following form and mail it to `larch@lcs.mit.edu`.

```
Name:     ______________________________      Date: __________________

E-mail address: _________________________

Version:  ___         [Type lp to find out.]

Hardware: ___ Sparc      ___ DECmips     ___ Alpha
          ___ Other: __________________________

Operating System and Version: _____________________________

Which of the following symptoms occurred (check all that apply):
    __ system level error (OS crashes or hangs, unrelated data is lost)
    __ program crashes (segmentation fault, core dump, etc.)
    __ program reports a fatal error
    __ program hangs
    __ program exhibits __ incorrect, __ mysterious, or __ unfriendly behavior
    __ program destroys data
    __ poor or incorrect error messages
    __ incorrect or missing documentation
    __ missing feature (please elaborate)
    __ other (please elaborate)

Problem description (please append a stand-alone command file that we can
execute to reproduce the bug):
```

7.10 News

April 27, 1995

An updated release, Version 3.1a, of LP is now available. This release fixes some bugs in Version 3.1.

December 30, 1994

A major new release, Version 3.1, of LP is now available. For more information, see

- Features added in Release 3.1
- Changes required in old proof scripts for use with Release 3.1

- LP user manual
- Getting LP

7.10.1 Features added in Release 3.1

The following features have been added to LP since Release 2.4.

- Support for full first-order logic, not just the universal-existential subset supported by Release 2.4. See quantifiers.
- A simple sort system for describing polymorphic abstractions.
- New inference mechanisms
  - Proofs by well founded induction.
  - Proofs of conjectures of the form `t1 <=> t2`, with `t1 => t2` and `t2 => t1` as subgoals.
  - Proofs that use deduction rules for backward inference. See apply.
- Richer syntactic conventions, such as `x[n]` and `n!`, for operators and terms.
- Additional user amenities, for example, enhanced facilities for naming sets of statements.
- New rewrite rule for the boolean operators, namely, `~p <=> ~q -> p <=> q`.

The following features behave differently in Release 3.1 than they did in Release 2.4.

- Some settings (e.g., the default name prefix) are now local to proof contexts.
- Names such as `thmCaseHyp1.1` are reused within non-overlapping proof contexts.
- LP now attempts to preserve the order of operands in formulas such as `init => c = 1`, so that preference is given to ordering equalities in the direction the user may have intended when these formulas normalize to equations.

7.10.2 Changes required in old proof scripts

Users must observe the following new syntactic conventions when using Release 3.1 instead of Release 2.4.

- Case is significant now in identifiers (but not in names).

```
Old: declare sort Nat                New: declare sort Nat
     declare operator 0: -> nat           declare operator 0: -> Nat
```

- There are new symbols for the boolean operators.

```
Old: not, &, |                       New: ~, /\, \/
```

(Fewer parentheses are needed now because the boolean operators /\ and \/ bind more tightly than =>, which binds more tightly than <=>.)

- There is now a single equality operator; i.e., "double equals" is gone.

```
Old: x + 0 == x                      New: x + 0 = x
     init(x) == x = 0                     init(x) <=> x = 0
     init(x) == x = 0                     init(x) = (x = 0)
```

- Declarations for infix operators need to be decorated with markers. (Now users can also declare prefix and postfix operators.)

```
Old: declare op +: N, N -> N        New: declare op __+__: N, N -> N
```

- Assertions must be separated by semicolons. (Now users can mix arbitrary assertions in the same `assert` command.)

```
Old: assert ac +                    New: assert
     assert                                ac +;
       1 = s(0)                            1 = s(0);
       x + 0 = x                           x + 0 = x
       ..                                  ..
```

- Induction rules and partitioned-by's must begin with the keyword `sort`.

```
Old: assert Nat generated by 0, s   New: assert sort Nat generated by 0, s
     assert Set paritioned by /in        assert sort Set partitioned by \in
```

- Some qualifications added to disambiguate terms must be rewritten.

```
Old: assert a.b:S = 2               New: assert a.(b:S) = 2
     assert a:S[n]                       assert (a:S)[n]
```

- Cases must be separated by commas.

```
Old: resume by case a b not(a | b)  New: resume by case a, b, ~(a \/ b)
```

- Deduction rule hypotheses and conclusions must be separated by commas.

```
Old: when h1 h2 yield c1 c2         New: when h1, h2 yield c1, c2
```

- Deduction rules now use the new notation for universal quantifiers.

```
Old: when forall x p(x) q(x)        New: when \A x p(x), \A x q(x) yield c
          yield c
```

- Commas and/or slashes are used now in name lists for the operations of union and intersection.

```
Old: display d-r nat set            New: display d-r / (nat, set)
```

- The annotations for box checking have been changed. (It is best to let LP regenerate them in a script file.)

```
Old: prove P(x) by case x = 0       New: prove P(x) by case x = 0
       <> 2 subgoals for proof                <> case xc = 0
         ... commands ...                     ... commands ...
         [] case 0 = xc                       [] case xc = 0
         ... commands ...                     <> case ~(xc = 0)
         [] case not(0 = xc)                  ... commands ...
       [] conjecture                          [] case ~(xc = 0)
                                            [] conjecture
```

Users may have to make additional changes in their scripts because LP reacts differently to them in the following ways.

- LP may orient some equations (e.g., case hypotheses) in the opposite direction. Many times, the new direction will be the one the user desired. But the new direction may cause LP to compute different critical pairs.
- LP flattens terms differently, because new spellings for (the boolean) operators collate differently. This may result in different reduction sequences during normalization, which may cause some proofs to succeed faster, some to succeed slower, and some to fail entirely.
- LP may generate different variable identifiers, e.g., when translating `partitioned by`.
- LP formulates the proof obligation for a deduction rule as an implication. This enables users to prove deduction rules using induction. However, box-checking now supplies/requires different annotations.

7.11 Accessible quantifiers

The fix and instantiate commands, together with the generalization and specialization proof methods, enable users to eliminate quantifiers from facts and conjectures. For a quantifier to be eliminable, it must be *accessible*, that is, it must not be in the scope of another (explicit or implicit) quantifier over the same variable.

7.12 Confluent rewriting systems

A set of rewrite rules is called *confluent* (or *Church-Rosser*) if, whenever a term `t` can be rewritten in two ways to terms `u` and `v`, there is another term `w` such that `u` and `v` both rewrite to `w`. A set of rewrite rules is confluent if it is both terminating and *locally confluent*, i.e., if whenever a term `t` can be rewritten two ways, each in a single step, to terms `u` and `v`, there is another term `w` such that `u` and `v` both rewrite to `w` (Newman, 1942).

7.13 Conjecture

A *conjecture* is a formula, deduction rule, induction rule, or operator theory that has been entered by a prove command, but whose proof is not yet complete. See also subgoal.

7.14 Conservative extension

One system is a *conservative extension* of another if any consequence of that system is also a consequence of the other or contains a function symbol that does not occur in the other. Thus, conservative extensions can be used to define properties of new symbols, but do not introduce inconsistencies or additional properties of existing symbols.

7.15 Convergent rewriting systems

A set of rewrite rules is called *convergent* if all terms can be rewritten to a terminal form in a finite number of steps, and all terms have a unique normal form.

See also

- Confluence
- Termination
- The complete command

7.16 Equational term-rewriting

Equational term-rewriting differs from conventional term-rewriting in that matching, unification, and completion are performed module a set `E` of equations.

A set `E` of equations defines an *equational theory*, which is the set of equations that can be derived from `E` by substituting equals for equals. More generally, a set of equations and rewrite rules defines an equational theory, which is that obtained by considering the rewrite rules as equations. A major purpose of the Knuth-Bendix completion procedure is to provide a decision procedure (reduction to normal form) for the question of whether equations are in the equational theory of a given system of equations and rewrite rules.

7.17 Flattened representations for terms

LP facilitates matching terms involving associative-commutative (ac) operators by using a special *flattened* representation for those terms.

When an argument of an ac operator has that operator as its principal operator, LP replaces it by its arguments. For example, when `+` is ac, `(a + b) + (c + d)` flattens to `a + b + c + d`.

Furthermore, when a flattened term involving an ac operator has more than two arguments, LP sorts the arguments lexicographically; for example, when `+` is ac, `(c + b) + a` flattens to `a + b + c`.

Two-argument terms such as `q <=> p`, where <=> is the hardwired ac operator for boolean equivalence, are not flattened to `p <=> q`. This enables LP to orient them into rewrite rules in the direction the user (may have) intended.

7.18 Matching

A term `t1` is said to *match* a term `t2` if it can be transformed into `t2` by some substitution of terms for its variables. When operators with ac or commutative operator theories occur in `t1` and `t2`, or when `t1` and `t2` contain quantifiers, `t1` matches `t2` in LP if there is a substitution that transforms `t1` into a term equivalent to `t2` modulo these theories and changes of bound variables.

7.19 Normalization

A term is said to be *normalized*, or in *normal form*, with respect to a set of rewrite rules if none of those rules can be applied to the term. The command show normal-form displays the normal form of a term with respect to the active rewrite rules in LP's logical system. See also the normalize command.

If the rewriting system is not guaranteed to terminate (e.g., if the user oriented an equation into a rewrite rule manually), the normal form computation may not terminate. When the rewriting system is not known to terminate, LP stops the rewriting process and issues a warning after a very large number of rewrites during a normal form computation. See the rewriting-limit setting.

If the rewriting system is not confluent, a term may have more than one normal form.

LP uses normalization as a method of forward and backward inference.

7.20 Reduction by rewrite rules

Terms can be *reduced* or *rewritten* by the application of a rewrite rule, as follows. First, the left side of the rewrite rule is matched against the term or one of its subterms. (Matching is the process of finding a substitution for the variables of a term that makes it equivalent to another term.) If matching is successful, the resulting substitution is applied to the right side of the rewrite rule and the matched term is replaced with this new term.

7.21 Skolem constants and functions

A *Skolem constant* is a new constant that is substituted for a variable when eliminating an existential quantifier from a fact or a universal quantifier from a conjecture. For example, the fact `\A x (f(x) = c)` can be obtained by eliminating the existential quantifier from `\E y \A x (f(x) = y)` and replacing `y` by the Skolem constant `c`. As long as `c` does not appear in any other fact or in the current conjecture, this Skolemization represents a conservative extension of LP's logical system.

For another example, the conjecture `\A x (f(x) = 0)` can be proved from the subgoal `f(c) = 0`, where `c` is a Skolem constant. As long as `c` is new, the proof is sound.

When an existential quantifier is being eliminated from a fact that contains free variables, or when the existential quantifier occurs in the scope of a universal quantifier, the quantified variable must be replaced by a term involving a *Skolem function* applied to the free and universally quantified variables. For example, the fact `\A x (x < bigger(x))` can be obtained by eliminating the existential quantifier from `\A x \E y (x < y)` and replacing `y` by `bigger(x)`.

See the fix command and the generalization proof method.

7.22 Substitutions

A *substitution* is a mapping from variables to terms (of the appropriate sorts). A substitution can be applied to a term as well as to a variable; its effect is to replace (simultaneously) each variable in the term by the substitution applied to that variable. More precisely, if `sigma` is a substitution defined on variables, then `sigma` extends to terms by setting

```
sigma(f(t1, ..., tn)) = f(sigma(t1), ..., sigma(tn))
```

A substitution `sigma` is an *instance* of a substitution `sigma1` if there is a substitution `sigma2` such that `sigma(t) = sigma2(sigma1(t))`.

See also captured.

7.23 Subgoal

A *subgoal* is a conjecture introduced by a method of backward inference in an attempt to prove another conjecture. Often additional hypotheses may be used in the proof of the subgoal.

7.24 Terminating rewriting systems

A set of rewrite rules is called *terminating* if each term can be reduced only a finite number of times. Generally some well-founded ordering on terms is used to establish the termination of a rewriting system.

7.25 Unification

Two terms are *unifiable* if they have a common instance, i.e., if there is a substitution (called a *unifier* of the terms) that transforms them both into the same term. When two terms contain operators with ac or commutative operator theories, or when they contain quantifiers, they are unifiable if there is a substitution that transforms them into terms that are equivalent modulo these theories and changes of bound variables.

A set of unifiers for two terms is *complete* if every unifier of the two terms is a substitution instance of some unifier in the set. A unifier of two terms is a *most general unifier* if, whenever it is (equivalent to) a substitution instance of another unifier, that other unifier is also (equivalent to) a substitution instance of it. For any two unifiable terms containing ac and/or commutative operators, there is a finite set of most general unifiers that is complete.