Manip User's Guide, Version 1.3

Ben L. Di Vito
NASA Langley Research Center
b.divito@nasa.gov

October 19, 2012

Abstract

Sequent manipulations for an interactive prover such as PVS can be labor intensive. We provide an approach to tactic-based proving for improved interactive deduction in specialized domains. This approach has been mechanized by the Manip package of strategies (tactics) and support functions. Although it was designed originally to reduce the tedium of low-level arithmetic manipulation, many of its newer features are suitable as general-purpose prover utilities. Besides strategies aimed at algebraic simplification of real-valued expressions, Manip includes term-access techniques applicable in arbitrary settings.

1 Introduction

Proving theorems involving nonlinear arithmetic can try a user's patience. While the automated deduction features of PVS can often find a proof, reasoning in the domain of nonlinear, real-valued expressions is currently limited to specialized sub-domains. Fortunately, SRI continues to increase the amount of automation in PVS and companion tools such as the SMT solver Yices. Third-party contributors such as NASA Langley have introduced new capabilities for deduction over polynomials.

We look forward to these and similar improvements. Nevertheless, there will always be a point where the automation runs out. The tools described here are designed to help when that point is reached. They are not designed to supplant anything already provided in PVS, only to augment the capabilities so that commonly needed operations can be done more conveniently.

Three types of extensions are included. First is a set of prover strategies that allows commonly occurring manipulations to be performed using fewer steps. Second is a notation and supporting implementation for an extended method of specifying input expressions when invoking prover commands. Third is a set of Emacs functions to improve the user interface for invoking these strategies and ordinary prover rules as well.

Section 7 explains the major enhancements introduced in version 1.2. Section 8 explains the major enhancements introduced in version 1.3. Section 11 lists a number of caveats and limitations of the current version.

Manip is now fairly mature. It has been exercised successfully for over ten years by users at NASA Langley and also at a few other sites. Further development in the near future is not anticipated, although enhancements are possible if the need arises. Manip was originally designed to be a separate package invoked using the PVS prelude extension feature. As of PVS version 5.0, it is included as a built-in component of PVS.

2 Emacs Extensions

This set of extensions introduces PVS prover shortcuts that help when manipulating sequents. The package streamlines interactive strategy invocation by assisting with certain types of argument entry. It adds features similar to those of the PVS prover helps package originally developed by C. Michael Holloway of NASA Langley and now distributed with PVS. Also provided are miscellaneous Emacs features to help with proof maintenance and other assorted tasks.

2.1 Prover Command Invocation

[Note: The features in this section were introduced in the first version of Manip. They have been partially superseded by newer features in later versions.]

Two specific interface features are incorporated. One is a means of invoking strategies that prompts the user through the argument list so it is unnecessary to memorize the formal argument lists of strategies. This works for all the built-in prover rules and strategies as well. The other feature allows the user to streamline cut-and-paste operations by supporting argument entry via mouse-dragging selections. This is helpful when it is necessary to include PVS expressions clipped from the current sequent. (Newer Manip features of other kinds, such as the syntax matching capability introduced in version 1.2, have reduced considerably the need for literal term extraction.) Both of these features are incorporated into a single TAB-command invocation sequence.

The basic usage pattern is as follows.

- TAB-z initiates the command entry sequence. The user is prompted for the name of a strategy (or rule) to invoke. The user will be prompted for inputs according to the formal argument list of the chosen rule or strategy.
- To supply a value for an argument, the user has the choice of either entering text in the minibuffer, or selecting a region of text in the prover's Emacs buffer, either by a mouse selection or any other means that sets *point* and *mark*.
- A typed minibuffer text argument is terminated by a CR (Return/Entry key) in the usual way. For a text region selection, TAB-, (TAB key followed by comma) causes the text region to be grabbed and added to the list of strategy arguments.
- Quotes are added automatically to selected text but not to typed text because it might contain numbers or other constants. The user repeats the text entries or region selections until all required arguments have been supplied.
- If there are `&optional` arguments, the user is prompted for these as well and may enter them using the same methods. Optional argument keywords are not typed. Entering a null string in the minibuffer for an optional argument selects its default value.
- Entering a ";" for any optional argument causes the remaining optionals to be skipped and will proceed to the `&rest` argument phase if such an argument exists. Otherwise, ";" terminates argument entry. A TAB-; (TAB-semicolon) typed after a region selection has the same effect of moving to the next phase.
- Rest argument entry proceeds for as many values as the user wishes to supply.

- Entering the string "\" (single backslash character) discards the last argument and rolls back to the previous one.
- Argument entry may be terminated at any time in the `&optional` or `&rest` phases by supplying the value "." in the minibuffer. Typing TAB-. (TAB-period) after a region selection has the same effect.
- After the desired sequence of arguments has been gathered, the completed rule or strategy command is sent to the prover.

This sequence may be abandoned at any point before completion using `C-g` and the partially constructed command will be deleted from the end of the Emacs buffer.

2.2 Proof Maintenance Utilities

Several functions are available to assist with proof maintenance activities.

- Maintaining PVS proofs sometimes requires replaying previous proofs after changing one or more theories, then editing failed steps embedded deep within the tree structure of commands. TAB-y is a utility to assist in finding the correct proof node in the Emacs buffer `Proof`, which is created by various commands such as `M-x edit-proof`. Position the cursor at the beginning of a proof label such as `tan_increasing_imp.3.2.2` in the prover buffer `*pvs*`. The label will be parsed and the cursor moved to the buffer `Proof` at the first step of the branch determined by the label. It is also possible to use labels found in prover messages such as:

  ```
  This completes the proof of tan_increasing_imp.3.2.2.
  ```

  The period at the end of the line will be recognized as punctuation rather than a part of the label and thus discarded by the label parser.

  This feature also works with the `*Proof*` buffer created by the `show-current-proof` command, which typically is used during ongoing proof development rather than proof maintenance. Whichever buffer is currently displayed, `Proof` or `*Proof*`, will be searched for a proof label. With `*Proof*` the cursor is placed on the last step of a branch rather than the first.

- The interactive Emacs Lisp function `M-x expand-strategy-steps` allows a user to "expand" the strategy steps of a proof file, provided no proof is in progress. The user will be prompted for a proof file name. Each rule in the proof file is checked against a list of base rules found in the core PVS distribution. Any strategy name not found there is appended with a '$' character so that it becomes a nonatomic command, causing the next proof attempt to expand it into steps found only in the core rule base. A backup file of the original proofs is saved in a `.sprf` version of the proof file. Finally, the revised proof file is installed to make it current.

  This feature can be used for several purposes:

  – It allows proofs to be developed using domain-specific strategies for increased productivity then converted to a portable form using only core proof rules.

  - It allows proofs to be rerun without strategies to confirm that no unsoundness has been introduced by the strategies.
  - It allows users to create personal strategies for proof development, even highly speculative ones, knowing that proofs can be easily purged of nonstandard commands should the strategies be later discarded or abandoned.

- The interactive Emacs Lisp function `M-x restore-strategy-steps` allows a user to restore the strategy steps found in a previously saved `.sprf` file. The current `.prf` file is simply replaced by the `.sprf` file. This function may not be invoked while a proof is in progress. The restored proof file is installed to make it current.

2.3 Other Emacs Extensions

Other Emacs features and TAB key assignments are provided for miscellaneous purposes.

- Several commands described later require the user to embed parameters in control strings using the percent (%) character. This causes a problem when installing edited proofs because of the well-formedness checking performed by the PVS `install-proof` function. In particular, % characters are interpreted as PVS comment characters, which can cause some expressions to fail the balance checks.

  To avoid this problem by suppressing the string balance checks, we have added an alternative function called `install-proof!`. After editing a proof, a user may invoke `install-proof!` using the (modified) key binding `C-x C-s`, while the regular version of `install-proof` is still available using `C-c C-i` or `C-c C-c`.

3 Strategy Preliminaries

Before describing the actual strategies, we sketch a few noteworthy features and capabilities that apply throughout the package. Most are concerned with the formulation of arguments supplied during strategy invocation.

- In PVS nomenclature, a *rule* is an atomic prover command and a *strategy* is a command that expands into one or more atomic steps. A *defined rule* is a command defined as a strategy but invoked as an atomic step. What we loosely call strategies in this package are defined rules when invoked in the normal manner. The "$" forms are the nonatomic strategy forms, which can be used to improve diagnostic information by showing the expansion into core PVS rules. For instance, using `cancel$` instead of `cancel` spawns an expanded proof.

- Many of the lemmas in prelude theory `real_props` are used by Manip's arithmetic strategies. Additional lemmas are needed to implement certain operations. The prelude theory `extra_real_props` provides the additional real number properties used by Manip.

- As is true for the built-in prover commands, wherever a formula number is called for, a formula label (symbol) may be supplied instead of a number. The special symbols $+$, $-$ and $*$ are also available with their usual meanings as lists of formula numbers. A special form is provided to construct the complement of a set of formula numbers. Using the form

`(^ n1 ... nk)` wherever formula numbers are required indicates the list of all formulas minus $n_1, \ldots, n_k$. Two additional forms, `(-^ ...)` and `(+^ ...)`, yield the complements of antecedent and consequent formula lists.

- Many of the arithmetic strategies accept *term numbers* as arguments, which are specified in a manner analogous to formula numbers. A term's position within its enclosing expression determines its term number value. Rules for counting terms are based on the arithmetic operators involved. Term numbers are expressed as integers, with term 1 designating the first (left-most) term. The special symbol $*$ is also available to denote the list of all terms in an expression. Negative term numbers allow indexing from right to left, that is, -1 selects term n, -2 selects term $n-1$, etc. The special form `(^ n1 ... nk)` indicates all term numbers except $n_1, \ldots, n_k$.

- The prover has many commands that allow a user to specify PVS expressions as arguments. Such expressions take the form of a literal string constant such as "`2 * PI * a!1`". We have found it useful to extend this capability and allow richer forms of expressions. Collectively these are called *extended expression specifications*. Section 5 describes these features in detail. For now we note that all strategies in this package that call for arguments in the form of terms or expressions may be supplied an extended expression as well as the familiar text string form. Extended expressions provide the means to specify terms by location reference as well as textual pattern matching, both of which offer new ways of selecting and synthesizing one or more PVS terms. A third method based on syntax matching was added in version 1.2 (Section 7).

- Extended expressions also collect formula number data whenever possible. Package strategies accept extended expressions wherever formula numbers are required. Numeric values are extracted from expression descriptors instead of text string components.

- Several strategies are provided in two variants to accommodate different argument types. One variant, which typically accepts arguments based on formula numbers, suffices for simple but common uses. The other variant, which accepts arguments based on a subset of extended expressions called *location references* (Section 5.2), can support more complicated uses. The second variant is distinguished by the `!` character at the end of its name.

- Proof branching is generated by some strategies where justification cases arise. Justification proofs may be attempted by supplying a non-nil value for the optional argument *try-just*, which may be either a proof step or the value `t` to indicate the step `(grind)`. In either case, if the justification branch is not proved completely, the proof state for that branch will be rolled back so the user can make a fresh attempt.

- When we speak of inequalities in the strategy descriptions, we refer to relations from the set $\{<, \leq, >, \geq\}$. The $\neq$ operator (`/=`) is not included because PVS normally eliminates such formulas by negating and moving them to the other side of the turnstile. Any occurrences of `/=` may be removed by using low-level prover commands such as `(prop)` and `(ground)`.

- Most user inputs are checked for well-formedness. Some are passed on to the prover for parsing where errors should be caught. Some bivalent argument types are drawn from a

set of symbols such as {L,R}. In such cases, typing any input value different from the default (e.g., L) will be assumed to represent the default's opposite value (e.g., R).

- Traditionally, Lisp symbols could be written in either upper or lower case. In this document and in online documentation, we often show symbols in upper case for emphasis ("L" is more clear than "l", which can look like the numeral "1"). PVS built with Allegro Common Lisp has recently become case sensitive, meaning that "L' and "l" are different symbols, while the CMU Lisp version of PVS remains case insensitive. We now strongly suggest that symbols be typed in lower case for both versions. With Allegro, upper case symbols will give incorrect results.

4 Algebraic Manipulation Strategies

This section describes a set of PVS prover strategies for manipulating arithmetic expressions and performing other detailed proving steps. It includes strategies helpful for proving formulas containing nonlinear arithmetic and similar expressions where PVS has limited automation. It is important to emphasize that these strategies might not be suitable as first-choice tools. Often it is preferable to try the more automatic prover features first, such as automatic rewriting using theories of rewrite rules or César Muñoz's Field strategies, then consider using these manipulations only if the other features fail. When proving with highly complex expressions, however, the automatic prover commands might take too long to be useful. In such cases, the more deliberate steps available from these strategies might be preferable.

For those manipulation strategies that require explicit terms as arguments, the terms can be specified using either text strings in the normal manner, or the extended expressions of Section 5, or the extensions to the Emacs PVS prover helps (TAB shortcuts) described in Section 2. Tables 1 and 2 list the various manipulation strategies provided along with their formal argument lists.

4.1 Simple Arithmetic Strategies

This group of strategies performs common algebraic manipulations that normally are not needed if your formulas fall within the domain of the linear arithmetic decision procedures or the rewrite rules of prelude theory `real_props`. Auto-rewriting with `real_props` is often powerful enough to prove many goals when combined with `grind`. Sometimes, however, it leads to excessive or unbounded rewriting. In such cases, more deliberate steps need to be taken.

Following are descriptions of the strategies and their signatures (formal argument lists). Invoking a strategy from the prover command line requires surrounding it in parentheses when typed, as is usually done. Invoking one using the TAB-z method of Section 2 will cause you to be prompted for each argument using the formal argument names shown.

`swap` *lhs operator rhs* `&optional` (*infix?* `t`) [*Strategy*]
`swap!` *expr-loc* [*Strategy*]

The `swap` strategy tries exchanging terms in commutative expressions. It replaces each applicable expression according to the scheme $x \circ y \Longrightarrow y \circ x$. All occurrences of $x \circ y$ in the sequent are replaced. Infix operators are normally expected, but prefix function application is

Table 1: Summary of manipulation strategies.

Syntax	Function
`(swap lhs operator rhs &opt (infix? t))`	$x \circ y \Longrightarrow y \circ x$
`(swap! expr-loc)`	
`(group term1 operator term2 term3`	L: $x \circ (y \circ z) \Longrightarrow (x \circ y) \circ z$
`  &opt (side L) (infix? t))`	R: $(x \circ y) \circ z \Longrightarrow x \circ (y \circ z)$
`(group! expr-loc &opt (side L))`	
`(swap-group term1 operator term2 term3`	L: $x \circ (y \circ z) \Longrightarrow y \circ (x \circ z)$
`  &opt (side L) (infix? t))`	R: $(x \circ y) \circ z \Longrightarrow (x \circ z) \circ y$
`(swap-group! expr-loc &opt (side L))`	
`(swap-rel &rest fnums)`	Swap sides and reverse relations
`(equate lhs rhs &opt (try-just nil))`	$\ldots lhs \ldots \Longrightarrow \ldots rhs \ldots$
`(has-sign term &opt`	Claims term has sign indicated
`  (sign +) (try-just nil))`	
`(mult-by fnums term &opt (sign +))`	Multiply both sides by term
`(div-by  fnums term &opt (sign +))`	Divide both sides by term
`(split-ineq fnum &opt (replace? nil))`	Split $\leq$ ($\geq$) into $<$ ($>$) and $=$ cases
`(flip-ineq fnums &opt (hide? t))`	Negate and move inequalities
`(show-parens &opt (fnums *))`	Show fully parenthesized formulas
`(move-terms fnum side`	Move additive terms to other side
`  &opt (term-nums *))`	
`(permute-terms fnum side`	Permute additive terms on one side
`  &opt (term-nums 1) (end R))`	
`(permute-terms! expr-loc`	Permute terms within an expression
`  &opt (term-nums 1) (end R))`	
`(elim-unary fnum &opt (side *))`	Converts $x \pm -y$ to $x \mp y$
`(elim-unary! expr-loc)`	and $-x + y$ to $y - x$
`(isolate fnum side term-num)`	Move all but one term
`(isolate-replace fnum side term-num`	Isolate then replace with equation
`  &opt (targets *))`	
`(cancel &opt (fnums *) (sign nil))`	Cancel terms from both sides
`(cancel-terms &opt (fnums *) (end L)`	Cancel speculatively & defer proof
`  (sign nil) (try-just nil))`	
`(cancel-add &opt (fnums *))`	Cancel additive terms in formulas
`(cancel-add! expr-loc)`	
`(op-ident fnum &opt`	Apply operator identity to rewrite
`  (side L) (operation *1))`	expression
`(op-ident! expr-loc &opt (operation *1))`	
`(cross-mult &opt (fnums *))`	Multiply both sides by denom.
`(cross-add &opt (fnums *))`	Add subtrahend to both sides
`(transform-both fnum transform`	Apply transform to both
`  &opt (swap nil) (try-just nil))`	sides of formula

Table 2: Summary of manipulation strategies (continued).

Syntax	Function

`(factor fnums &opt (side *)`
`(term-nums *) (id? nil))`
`(factor! expr-loc &opt`
`(term-nums *) (id? nil))`
	Extract common multiplicative factors from additive terms given

`(distrib fnums &opt (side *)` `(distrib! expr-loc)`	Distribute multiplication over additive terms

`(permute-mult fnums &opt (side R)`
`(term-nums 2) (end L))`
`(permute-mult! expr-loc &opt`
`(term-nums 2) (end L))`
	Rearrange factors in a product

`(name-mult name fnum side`
`&opt (term-nums *))`
`(name-mult! name expr-loc`
`&opt (term-nums *))`
	Select factors, assign name to their product, then replace

`(recip-mult fnums side)` `(recip-mult! expr-loc)`	$x/d \Longrightarrow x * (1/d)$
`(isolate-mult fnum &opt (side L)` `(term-num 1) (sign +))`	Select a factor and divide both both sides to isolate factor
`(mult-eq rel-fnum eq-fnum` `&opt (sign +))`	Multiply sides of relation by sides of equality
`(mult-ineq fnum1 fnum2` `&opt (signs (+ +)))`	Multiply sides of inequality by sides of another inequality
`(mult-cases fnum` `&opt (abs? nil) (mult-op *1))`	Generate case analyses for products

`(mult-extract name fnum &opt`
`(side *) (term-nums *))`
`(mult-extract! name expr-loc`
`&opt (term-nums *))`
	Extract selected terms, name replace them, then simplify

also accommodated by setting the *infix?* argument to `nil`. In fact, any binary function may be used provided its commutativity can be established.

In the `!` variant, a subset of extended expressions called *location references* (Section 5.2) may be used to supply the *expr-loc* argument. The referenced expression must be an application of a commutative function or operator. The strategy determines whether it is an infix or a prefix application. Multiple expression locations may result from a single *expr-loc* argument. Only the first will be processed.

Usage: `(swap "a!1" * "(x!1 - 2)")` commutes the two factors in the multiplicative expression `a!1 * (x!1 - 2)`. If formula 3 is `min(a!1, x!1) > 0`, then `(swap! (! 3 L))` rewrites the formula to `min(x!1, a!1) > 0`.

`group` *term1 operator term2 term3* `&optional` (*side* `L`) (*infix?* `t`) [*Strategy*]
`group!` *expr-loc* `&optional` (*side* `L`) [*Strategy*]

`group` tries rearranging terms in associative expressions. It replaces each applicable expression according to one of two schemes:

$$\mathrm{L} : x \circ (y \circ z) \Longrightarrow (x \circ y) \circ z, \qquad \mathrm{R} : (x \circ y) \circ z \Longrightarrow x \circ (y \circ z)$$

The `!` variant allows suitable expressions to be indicated by location references.

Usage: `(group "a!1" * "x!1" "u!1" R)` changes the expression `(a!1 * x!1) * u!1` so it associates to the right.

`swap-group` *term1 operator term2 term3* `&optional` (*side* `L`) (*infix?* `t`) [*Strategy*]
`swap-group!` *expr-loc* `&optional` (*side* `L`) [*Strategy*]

The previous two strategies are combined to replace according to the schemes:

$$\mathrm{L} : x \circ (y \circ z) \Longrightarrow y \circ (x \circ z), \qquad \mathrm{R} : (x \circ y) \circ z \Longrightarrow (x \circ z) \circ y$$

This might be used to "lift" a middle term out where it can be more accessible to lemmas and rewrite rules. The `!` variant allows suitable expressions to be indicated by location references.

Usage: `(swap-group "a!1" * "x!1" "sq(u!1)")` moves the middle term to the left.

`swap-rel` `&rest` *fnums* [*Strategy*]

Relational formulas may have their two sides swapped using this strategy, with the direction of any inequality operators being reversed. Normally this type of transformation is unnecessary. It might be useful in writing higher level strategies, however, where it can simplify matters to assume that the relation is always less-than, for example. In other situations it may be used to move preferred terms left so that rewrites are tried first on the chosen side.

`equate` *lhs rhs* `&optional` (*try-just* `nil`) [*Strategy*]

With `equate` a user can claim an equality between expressions and have *rhs* replace *lhs*. If the optional argument *try-just* is non-nil, it will be interpreted as a prover command to invoke for proving the justification, i.e., for proving *lhs* = *rhs*. As a special case, the value "`t`" may be given to apply `(grind)` in the justification step. Although the effect of `equate` can be achieved using the prover rule `case-replace`, `equate` obviates the explicit construction of an equality expression and offers more convenience when used with the Emacs TAB-z feature or the extended expression feature.

Usage: `(equate "(x!1 - 2)" "a!1" (assert))` replaces `(x!1 - 2)` by `a!1`, then applies `(assert)` to try to prove the equality holds.

`has-sign` *term* `&optional` (*sign* `+`) (*try-just* `nil`) [*Strategy*]

Often it is desirable to claim that a term has a certain sign or other relationship to zero. `has-sign` allows the user to claim that *term* has a designated property, where *sign* can be one of six symbols with meaning as follows:

`+`	`-`	`0`	`0+`	`0-`	`+-`
$x > 0$	$x < 0$	$x = 0$	$x \geq 0$	$x \leq 0$	$x \neq 0$

Proof of the justification step can be tried or deferred as indicated by *try-just*.

Usage: `(has-sign "sin(phi!1 + 2*PI) - sin(phi!1)" 0 t)` claims an expression has value zero and tries to prove it using `(grind)`.

`mult-by` *fnums term* `&optional` (*sign* `+`) [*Strategy*]

Both sides of a relational formula may be multiplied by a common factor using `mult-by`. For inequality relations, when the factor is known to be positive or negative, use + or − as the sign argument. Otherwise, use *, which introduces a conditional expression to handle the two cases in the same manner as `cross-mult` (see page 13). No sign argument is needed for equalities.

The built-in prover command `both-sides` offers a way to achieve similar effects. `mult-by`, however, provides the means to specify a term's polarity and perform a case split accordingly, which usually proves the justification branch automatically. With `both-sides`, it is often necessary to prove the justification explicitly. Moreover, when multiplying an inequality by a negative term, it will not formulate the desired proposition.

Usage: `(mult-by 2 "y!1" -)` multiplies both sides of formula 2 by `y!1`, which is declared to be negative, causing the two sides of the formula to be swapped.

`div-by` *fnums term* `&optional` (*sign* `+`) [*Strategy*]

Both sides of a relational formula may be divided by a common divisor using `div-by`. For inequality relations, when the divisor is known to be positive or negative, use + or − as the sign argument. Otherwise, use *, which introduces a conditional expression to handle the two cases in the same manner as `mult-by`. No sign argument is needed for equalities.

Usage: `(div-by 2 "sq(y!1)")` divides both sides of formula 2 by `sq(y!1)`, which is assumed to be positive.

`split-ineq` *fnum* `&optional` (*replace?* `nil`) [*Strategy*]

Given that *fnum* is a nonstrict, antecedent inequality (`<=` or `>=`), `split-ineq` forces the sequent to split into two cases, e.g., an equal-to and a less-than case. It also works if *fnum* is a strict consequent inequality. Simplification using `(assert)` is applied after splitting. The equality may be optionally used for replacement by supplying the direction `LR` or `RL` for the *replace?* argument.

Usage: If formula 2 is `x!1 > y!1`, then `(split-ineq 2 RL)` causes a case split on the expression "`x!1 = y!1`" and performs the replacement of `x!1` for `y!1` in the equality branch.

`flip-ineq` *fnums* `&optional` (*hide?* `t`) [*Strategy*]

One property of the prover's sequent representation is that a sequent having antecedent (consequent) formula P is equivalent to one having $\neg P$ as a consequent (antecedent) formula. The prover automatically makes use of this equivalence, even though $\neg P$ does not explicitly appear in the sequent. In the case of an inequality relation, its negation is itself another inequality. Occasionally a user might prefer one inequality form and location over another. Adjustments along these lines may be accomplished using `flip-ineq`.

For *fnums* that are inequality relations, `flip-ineq` negates the inequalities and moves the negated formulas by exchanging between antecedents and consequents. Conjunctions and dis-

junctions of inequalities are also accepted, causing each conjunct or disjunct to be negated in an application of De Morgan's law. If *hide?* is set to `nil`, the original formulas are left intact; otherwise, they are hidden.

Usage: If formula 2 is "`x!1 > y!1`", then `(flip-ineq 2)` causes "`x!1 <= y!1`" to be added as a new formula -1. If formula -3 is the disjunction "`x!1 > 9 OR y!1 < 6 OR z!1 >= 3`", then `(flip-ineq -3 nil)` adds "`x!1 <= 9 AND y!1 >= 6 AND z!1 < 3`" as a new formula 1 and retains the original formula -3.

`show-parens &optional` (*fnums* `*`) [*Strategy*]

Occasionally it is useful to see how terms are associated in a complex expression. The full parenthesization of a formula's term structure may be displayed using `show-parens`. Its behavior is incomplete; it does not handle all features of PVS syntax, only the common ones such as infix and prefix function application.

As of version 3.0, PVS includes the `M-x pvs-set-proof-parens` function, which is also available from the PVS menu. Turning this feature on causes sequents to be displayed with full parenthesization. Thus, `show-parens` is partly obsolete, but we retain it because it offers finer control, allowing parenthesis display only when needed and selectable by formula.

4.2 Intermediate Arithmetic Strategies

This second group of arithmetic strategies tries to carry out common manipulations without specifying the actual terms from the sequent. This is generally desirable to prevent detailed expressions from being saved with the proof step. Avoiding such cases can lead to more robust proofs that require less updating when lemmas or theories are changed.

`move-terms` *fnum side* `&optional` (*term-nums* `*`) [*Strategy*]

With `move-terms` a user can move a set of additive terms numbered *term-nums* in relational formula *fnum* from *side* (L or R) to the other side, adding or subtracting individual terms from both sides as needed. *term-nums* can be specified in a manner similar to the way formula numbers are presented to the prover. Either a list or a single number may be provided, as well as the symbol "`*`" to denote all terms on the chosen side. Note that parentheses and associative grouping are ignored for purposes of assigning term numbers, e.g., term 2 in `"x + (y + z)"` is `y`, not `y + z`.

Usage: `(move-terms 3 L (2 4))` moves terms 2 and 4 from the left to the right side of formula 3.

`isolate` *fnum side term-num* [*Strategy*]

A special case of `move-terms` is offered by `isolate`, which moves all additive terms except that numbered *term-num* from *side* (L or R) to the other side. If *fnum* is an equality, the effect is the same as solving for an additive term. Isolate is equivalent to the form `(move-terms fnum side (^ term-num))` (refer to term-number discussion on page 5).

Usage: `(isolate 1 R 3)` moves all right-side terms except number 3 to the left.

`isolate-replace` *fnum side term-num* `&optional` (*targets* `*`) [*Strategy*]

A further special case is when `isolate` is applied to an antecedent equality. The resulting equality may be used to replace the isolated term in *targets*, after which the equality is hidden.

Usage: `(isolate-replace 1 L 3 +)` solves for left-side term 3 and uses the resulting equality for replacement in the consequent formulas.

`cancel &optional` (*fnums* `*`) (*sign* `nil`) [*Strategy*]

Cancellation is available through the automatic rewrites of prelude theory `real_props`. Often this rewriting does more than desired, however, and at other times misses opportunities for cancellation. For these reasons, we provide a more focused operation in `cancel`. When the top-level operator on both sides of a relation in *fnums* is the same operator drawn from the set $\{+, -, *, /\}$, `cancel` tries to eliminate common terms using a small set of rewrite rules and possible case splitting. No other simplification is attempted.

Cancellation is possible when *fnum* has one of two forms:

$$x \circ y \; R \; x \circ z, \qquad y \circ x \; R \; z \circ x$$

The types allowed for x, y, z depend on the relation and arithmetic operator involved. In the default case, when *sign* is `NIL`, x is assumed to be (non)positive or (non)negative as needed for the appropriate rewrite rules to apply. Otherwise, an explicit *sign* can be supplied to force a case split so the rules will apply. If *sign* is `+` or `-`, x is claimed to be strictly positive or negative. If *sign* is `0+` or `0-`, x is claimed to be nonnegative or nonpositive. If *sign* is `*`, x is assumed to be an arbitrary real and a three-way case split is used. No sign argument is needed for equality relations.

At times, unproved cases requiring user attention are split off. Such cases can result when the canceled term does not match the *sign* argument or when cancellation is invalid for other reasons. A further caveat is that `cancel` only works with top-level operations. This means that `(x * y) * z = (x * a) * b` will not yield to `cancel`, nor will it be simplified through `real_props` automatic rewriting. In such cases, use the `cancel-terms` strategy (immediately following) or do some rearranging of the formula before attempting `cancel`.

Usage: `(cancel 3 0-)` tries to cancel from both sides of formula 3 after first splitting on the assumption that the common term is nonpositive.

`cancel-terms &optional` (*fnums* `*`) (*end* `L`) (*sign* `nil`) (*try-just* `nil`) [*Strategy*]

Often it is desirable to cancel nonidentical terms speculatively. This capability is offered through `cancel-terms`, which splits into cases on the assumption that both left-most or right-most terms in a relational formula are equal. The user can specify at which *end* (L or R) of a chain of similar infix applications to look for the allegedly common term. Associative groupings are ignored when identifying the *end* term. The '-' operator is considered equivalent to '+' for this purpose. On the other hand, only the outer-most application in a chain of '/'-separated terms is recognized.

As an example, suppose that formula 2 is `x * y * z > a * b * c`. `(cancel-terms 2)` will first introduce a case split on the condition `x = a`. Then it will use this equality to reduce formula 2 to `y * z > b * c`. On the other proof branch, the user will have to establish `x = a`.

For inequalities, the *sign* argument can be used to indicate term polarity as in `cancel`. In addition, an automatic proof attempt of the terms' equality can be triggered using *try-just*.

Usage: `(cancel-terms 3 L + t)` tries to cancel the left-most term from both sides of formula 3 after first splitting on the assumption that the positive terms are equal. An automatic attempt to prove their equality using `(grind)` is performed.

`op-ident` *fnum* `&optional` (*side* `L`) (*operation* `*1`) [*Strategy*]
`op-ident!` *expr-loc* `&optional` (*operation* `*1`) [*Strategy*]

The cancellation strategies do not handle any "one-sided" cases, e.g., a relation of the form $x\ R\ x * y$. Rewriting with `real_props` likewise offers no benefit. We provide `op-ident` to perform the setup for such cancellations and similar operations. The operator identity given by *operation* is used to rewrite the expression found on *side* of formula *fnum*.

In the `!` variant, a subset of extended expressions called *location references* is provided for supplying the *expr-loc* argument (Section 5.2). Multiple expression locations may result from a single *expr-loc* argument. Each will be processed separately.

Currently, the following operations are available using these designated symbols:

`z+`	`+z`	`-z`	`1*`	`*1`	`/1`
$0 + x$	$x + 0$	$x - 0$	$1 * x$	$x * 1$	$x/1$

Note that symbols using 'z' rather than '0' are used because `+0` and `-0` are treated by Lisp as the number 0 rather than as symbols.

Usage: `(op-ident -2 L 1*)` rewrites formula -2 from `b!1 < a!1 * b!1` to the equivalent formula `1 * b!1 < a!1 * b!1`. The form `(op-ident! (! -2 L) 1*)` achieves the same result, although both occurrences of `b!1` will be replaced.

`cross-mult` `&optional` (*fnums* `*`) [*Strategy*]

When the various rewrite rules fail to produce the desired effect in eliminating divisions, `cross-mult` may be used to explicitly perform "cross multiplication" on one or more relational formulas. For example, $a/b < c/d$ will be transformed to $ad < cb$. The strategy determines which lemmas to apply based on the relational operator and whether negative divisors are involved. Cross multiplication is applied recursively until all outermost division operators are gone.

`cross-mult` also tries to do something reasonable in case the denominators are not known to be strictly positive or negative. Lemmas provided in theory `extra_real_props`, such as

```
div_mult_pos_neg_lt1: LEMMA
  z/n0y < x IFF IF n0y > 0 THEN z < x * n0y ELSE x * n0y < z ENDIF
```

are used to carry out cross multiplication using conditional expressions. If the denominators are of type `posreal` or `negreal`, however, these lemmas are not required.

`cross-add` `&optional` (*fnums* `*`) [*Strategy*]

Performing "cross addition" is handled in most cases by `move-terms`. There are times, however, when it is desirable to find subtractions automatically and add the subtrahends to both sides. This type of cross addition is performed by `cross-add`, applying the procedure recursively until all outermost subtraction operators on either side of the relational formulas are gone.

`factor` *fnums* `&optional` (*side* `*`) (*term-nums* `*`) (*id?* `nil`) [*Strategy*]
`factor!` *expr-loc* `&optional` (*term-nums* `*`) (*id?* `nil`) [*Strategy*]

If the expression on *side* of each formula in *fnums* has multiple additive terms, `factor` may be used to extract common multiplicative factors and rearrange the expression. The additive terms indicated by *term-nums* are regarded as bags of factors to be intersected for common factors. Terms not found in *term-nums* are excluded from this process. If *side* is `*`, both sides will be factored separately using *term-nums*, which might not be useful unless *term-nums* is also `*`. The default case of "`(factor <fnums>)`" tries to factor both sides (separately) using all the terms of each side. Currently, there is no attempt to handle divisions; only multiplications within additive terms are recognized by the factoring process.

In the `!` variant, the *expr-loc* argument supplies a location reference to identify the target expression(s). Multiple expression locations may result from a single *expr-loc* argument. Each will be processed separately.

If the optional argument *id?* is set to `t`, then the additive terms are wrapped in an application of the identify function `id` after factoring. This prevents later distribution of the multiplication operators by subsequent prover commands, which might undo the work of `factor` before the factored expressions can be used.

As an example, suppose formula 4 has the form

```
f(x) = 2 * a * b + c * d - 2 * b
```

and the command "`(factor 4 R (1 3) t)`" is issued. Then the strategy will rearrange formula 4 to:

```
f(x) = 2 * b * id(a - 1) + c * d
```

For a more complicated example, `(factor! (! 4 R (->* "cos") 1))` factors the argument of each instance of the `cos` function on the right side of formula 4.

`transform-both` *fnum transform* `&optional` (*swap* `nil`) (*try-just* `nil`) [*Strategy*]

A generalized "both sides" command is offered by `transform-both`, although it is unable to select suitable lemmas and therefore leaves that work for the user. The idea is to apply an arbitrary transform to both sides of a relational formula, where the transform is written as a parameterized PVS expression. This mechanism is described in full in Section 6.1; a special case is used here. The strategy itself may be viewed as a special case of the strategy invocations described in Section 6.2.

The transform expression uses the string "`%1`" to represent the left- and right-hand side expressions in the relation. Hence the transform can be regarded as a macro or template expression with "`%1`" serving as an implicit macro or template parameter. As an example, suppose formula −3 is "`a/b = c/d`." Invoking the command

```
(transform-both -3 "2 * sqrt(%1)")
```

takes the square root of both sides of formula −3 then multiplies by 2. A case split is introduced based on the formula

```
2 * sqrt(a/b) = 2 * sqrt(c/d)
```

Proof of the justification step can be tried or deferred until later. The flag *swap* is used to indicate when the sides should be swapped (e.g., when multiplying by a negative number).

Usage: `(transform-both 3 "-PI * %1" T (ground))` multiplies both sides by $-\pi$, swapping the two sides in the process, and tries to prove the transformation is valid using `ground`.

4.3 Strategies for Manipulating Products

To enhance reasoning capabilities for nonlinear arithmetic, we provide several strategies for manipulating products or generating new products. This supports an overall approach of first converting divisions into multiplications where necessary, then using a broad array of tools for reasoning about multiplication. Many of these manipulations apply lemmas already present in the prelude. Use of the strategies allows proof construction without detailed knowledge of these lemmas or the need to remember their names.

`permute-mult` *fnums* `&optional` (*side* `R`) (*term-nums* `2`) (*end* `L`) [*Strategy*]
`permute-mult!` *expr-loc* `&optional` (*term-nums* `2`) (*end* `L`) [*Strategy*]

When there are three or more multiplicative terms in a product, it is sometimes difficult to make progress because the terms appear in an undesirable order or the association of terms gets in the way of applying lemmas. This can impede the application of various simplifications such as cancellation. To remedy the situation, a user can apply `permute-mult` to reorder terms in a product.

To perform this task, as well as several others in this group of strategies, the user needs to refer to individual terms in a product. This is done using the same method as earlier strategies. After identifying the expression to draw terms from, the argument *term-nums* is used to supply a single term number or list of term numbers. Terms in a product are numbered left-to-right starting with number 1. Parentheses are ignored for the purpose of numbering terms.

For *end* = `L`, the action of `permute-mult` is as follows. Let the expression on *side* of a formula in *fnums* be a product of terms, $P = t_1 * \ldots * t_n$. Identify a list of indices I (*term-nums*) drawn from $\{1, \ldots, n\}$. Construct the product $t_{i_1} * \ldots * t_{i_l}$ where $i_k \in I$. Construct the product $t_{j_1} * \ldots * t_{j_m}$ where $j_k \in \{1, \ldots, n\} - I$. Then rewrite the original product P to the new product $t_{i_1} * \ldots * t_{i_l} * t_{j_1} * \ldots * t_{j_m}$. Thus the new product is a permutation of the original set of factors with the selected terms brought to the left in the order requested. For *end* = `R`, the selected terms are placed on the right.

In the `!` variant, the *expr-loc* argument supplies a location reference to identify the target expression(s). Multiple expression locations may result from a single *expr-loc* argument. Each will be processed separately.

Usage: `(permute-mult 3 L (4 2))` rearranges the product on the left side of formula 3 to be `t4 * t2 * t1 * t3`, with the default association rules making it internally represented as `((t4 * t2) * t1) * t3`.

`name-mult` *name fnum side* `&optional` (*term-nums* `*`) [*Strategy*]
`name-mult!` *name expr-loc* `&optional` (*term-nums* `*`) [*Strategy*]

With `name-mult` a user can take the action of `permute-mult` one step further. After selecting and extracting a product P of subterms to place on the left of the new product, P is assigned a name and a `name-replace` operation is carried out so that $P =$ *name* is added as a new

antecedent formula. In the `!` variant, if multiple locations result from *expr-loc*, only the first one is processed.

Usage: `(name-mult "prod1" 3 L (4 2))` rearranges the product on the left side of formula 3 to be `PROD1 * t1 * t3` and adds the equality `t4 * t2 = PROD1` to the antecedents.

`recip-mult` *fnums side* [*Strategy*]
`recip-mult!` *expr-loc* [*Strategy*]

With `recip-mult` a user can convert an expression from a division to a multiplication by the reciprocal of the divisor. This presents an alternative way to deal with divisions from that offered by the `cross-mult` strategy. Reciprocals might be preferable when it is necessary to maintain a formula in the form of an equation such as `x = y * (1/z)` because substitution for `x` is anticipated shortly. Reciprocals also help when applying lemmas that assume expressions are in product form. In the `!` variant, if multiple locations result from *expr-loc*, each is processed separately.

Usage: `(recip-mult 2 R)` turns the (top-level) division on the right side of formula 2 into reciprocal multiplication.

`isolate-mult` *fnum* `&optional` (*side* `L`) (*term-num* `1`) (*sign* `+`) [*Strategy*]

`isolate-mult` is used to migrate factors from a product to a division on the other side of a relation. Generally this is undesirable, but there are circumstances where solving for a term found within a product is necessary to enable later replacement actions. Given that formula *fnum* has the form $t_1 * \ldots * t_n$ R e (*side* is L), selecting term i for isolation produces the new formula t_i R $e/(t_1 * \ldots * t_{i-1} * t_{i+1} * \ldots * t_n)$. For inequalities, the *sign* argument may be used to indicate when this divisor is a negative quantity. A case split is introduced to establish that the divisor is positive or negative as claimed.

Usage: `(isolate-mult 4 L 3 +)` divides both sides by all of the left-side terms of formula 4 except number 3, which collectively forms a positive product.

`mult-eq` *rel-fnum eq-fnum* `&optional` (*sign* `+`) [*Strategy*]

Sometimes it is helpful to generate a new relation based on the products of terms from two existing formulas. Given a relational formula a R b and an antecedent equality $x = y$, `mult-eq` forms a new antecedent or consequent relating their products, $a*x$ R $b*y$. If R is an inequality, the *sign* argument can be set to one of the symbols in {`+, -, 0+, 0-`} to indicate the polarity of x and y (positive, negative, nonnegative, nonpositive). A *sign* of `*` is not supported (yet).

Usage: `(mult-eq -3 -2 -)` multiplies the sides of formula -3 by the sides of equality -2, which are assumed to be negative. `(mult-eq -2 -2 -)` would square both sides of -2.

`mult-ineq` *fnum1 fnum2* `&optional` (*signs* `(+ +)`) [*Strategy*]

In certain cases, the terms of two inequalities can be used to generate a new inequality. Given two relational formulas *fnum1* and *fnum2* having the forms a R_1 b and x R_2 y, `mult-ineq` forms a new antecedent relating their products, $a*x$ R_3 $b*y$. If R_2 is an inequality having the opposite direction as R_1, `mult-ineq` proceeds as if it had been y R_2' x instead, where R_2' is the reverse of R_2. The choice of R_3 is inferred automatically based on R_1, R_2, and the declared

signs of the terms. R_3 is chosen to be a strict inequality if either R_1 or R_2 is. If either formula appears as a consequent, its relation is negated before carrying out the multiplication.

Not all combinations of term polarities can produce useful results with `mult-ineq`. Therefore, the terms of each formula are required to have the same sign, designated by the symbols `+` and `-`. Inequalities on terms of different polarities are not supported, largely because the truth of whether an inequality holds on the products depends on the relative magnitudes of the products rather than just the polarity of their factors. The formulas are allowed to have different signs, however, relative to each other. For example, *fnum1* could be an inequality on positive terms while *fnum2* is on negative terms. The *signs* argument must be a list of two signs denoting the polarities of *fnum1* terms and *fnum2* terms.

Usage: `(mult-ineq -3 -2 (- +))` multiplies the sides of inequality formula -3 by the sides of inequality -2, which are assumed to relate negative and positive values, respectively. `(mult-ineq -2 -2)` would square both sides of -2.

`mult-cases` *fnum* `&optional` (*abs?* `nil`) (*mult-op* `*1`) [*Strategy*]

Case analyses for relational formulas containing products are generated by `mult-cases`. Two types of relations are accommodated. If *fnum* has the form $x*y\ R\ 0$ (or $0\ R\ x*y$), `mult-cases` will rewrite *fnum* to two cases relating x and y to 0, as appropriate. Some flattening and simplification will be attempted after rewriting.

If *fnum* is a consequent inequality of the form $a*b\ R\ c*d$, `mult-cases` will generate sufficient conditions to establish the inequality by considering relations between a and c, and between b and d. Likewise, for an antecedent inequality of this form, `mult-cases` will generate necessary conditions for *fnum*. The lemmas used by `mult-cases` contain instances of the `abs` function, which are normally expanded. To suppress this expansion, set *abs?* to `t` and the applications of `abs` will be retained.

Some branching of the sequent is likely with this second relational form. Moreover, when the terms are unconstrained real values, the conditions generated are complex. Much better simplification occurs if the terms are known to be (non)positive or (non)negative. All combinations of term polarities should produce meaningful results.

IF *fnum* is an inequality of the form $a*b\ R\ c$ or $a\ R\ c*d$, *fnum* is first transformed into the form $a*b\ R\ c*d$ by multiplying c or a by 1. *mult-op* may be set to `*1` (`1*`) to multiply on the right (left). Case analysis then proceeds as in the general case described above.

Usage: `(mult-cases 2)` generates conditions for the products found in formula 2.

`mult-extract` *name fnum* `&optional` (*side* `*`) (*term-nums* `*`) [*Strategy*]
`mult-extract!` *name expr-loc* `&optional` (*term-nums* `*`) [*Strategy*]

Operating at a somewhat higher level, `mult-extract` performs a series of steps to simplify sums of products and put them into a form amenable to further manipulation. First, it extracts the additive terms specified by *term-nums* from the expression found on *side* of formula *fnum*. Each additive term is treated as a product of factors, some of which may contain divisions. Each product term thus selected is extracted using `name-replace` to form a new antecedent equality. A name for each product is constructed by appending an index to the argument *name*. After each equality is established, the divisors are multiplied out to remove top-level division operations (similar to the action of `cross-mult`). Then common factors on both sides of each equality are identified and canceled. In the `!` variant, if multiple locations result from *expr-loc*,

only the first one is processed.

Usage: `(mult-extract 2 L (1 3))` applies the prescribed sequence of manipulations to additive terms 1 and 3 on the left side of formula 2.

5 Extended Expression Specifications

To enhance the effectiveness of prover strategies, we provide a means for specifying *extended expressions* as strategy arguments. Two major types of extensions are included: location references and textual pattern matching. Location references allow a user to indicate a precise subexpression within a formula by giving a path of indices to follow when descending through the formula's expression tree. Pattern matching allows strings to be found and extracted using a specialized pattern language that is based on, but much less elaborate than, regular expressions. As of version 1.2, a third type based on matching PVS expressions syntactically is available (Section 7). Together the extensions offer much more flexibility for entering PVS expressions than simple text strings.

5.1 General Syntax

Extended expressions are specified using a combination of string literals and Lisp-oriented notation. Evaluation of extended expressions takes place during strategy execution, yielding sets of values that are used to form arguments to built-in prover commands. The results of this evaluation usually denote expressions in the PVS language but need not do so. Expression strings can be arbitrary text that will be combined later with other text to form more meaningful strings. The substitution mechanism presented in Section 6.1 enables this type of recombination.

An extended expression is recursively defined to have one of the following forms:

- A literal text string (characters in double quotes).
- An integer denoting a formula number. The string value of such an expression is the textual representation of the PVS formula.
- A symbol denoting either a formula label or one of the special symbols $+$, $-$, $*$, with their usual meanings as sets of formulas.
- A *location reference* having the form `(! <ext-expr> i1 ... in)`, where $i_1, \ldots, i_n$ are index values.
- A *pattern match* having the form `(? <ext-expr> p1 ... pn)`, where $p_1, \ldots, p_n$ are pattern strings.
- A *syntax match* having the form `(~ s1 ... sn)`, where $s_1, \ldots, s_n$ are specification items as described in Section 7.7.
- A list `(e1 ... en)`, where $e_1, \ldots, e_n$ are extended expression specifications. After evaluation, the lists generated by $e_1, \ldots, e_n$ will be concatenated into a single list.

Note that numbers when used as extended expressions do not denote numbers in the PVS language as they usually do at the prover interface. Numbers denote formulas; string literals such as `"4"` must be used to indicate PVS numbers. Similarly, formula labels must be entered as

symbols rather than strings, e.g., `sq_fmla` rather than `"sq_fmla"`. Also note that the location reference and pattern match forms take another extended expression as their first "argument." In practice, this is almost always a number or symbol. Nesting of extended expressions is possible, although some combinations do not yield useful results.

Evaluation of an extended expression can result in zero or more separate strings or objects being generated. Internally, evaluation produces a list of descriptors, each of which contains a text string, the number of the formula of origination, and the Common Lisp CLOS object that represents a PVS expression. Only the string component exists in all cases. For example, pattern matches generally do not produce a CLOS object because matches return arbitrary strings that need not correspond to PVS expressions.

For the strategies of Section 6.2, multiple expression specifications may be supplied as arguments. What happens in such cases is that each specification gives rise to an arbitrary number of descriptors. All the descriptor lists are then concatenated to build a single descriptor list before substitutions are performed.

5.2 Location References

A location reference has the form `(! <ext-expr> i1 ... in)`. The starting point `<ext-expr>` must describe the location of a valid PVS expression. The index values $\{i_j\}$ are used to descend the parse tree to arrive at a subexpression, which becomes the final value of the overall reference. Actually, the final value is a list of expressions, which allows for wild-card indices to traverse multiple paths through the tree. Moreover, the index values may include various other forms and indicators used to control path generation.

Location references are so named because they specify sites within the current sequent. This property allows them to be used as arguments for certain strategies where a mere text string is inadequate. For example, the `factor!` strategy can factor an expression in place using this feature even if the target terms appear in the argument to a function. Thus, location references can be regarded as somewhat analogous to array or structure references in a procedural programming language.

An example of a simple location reference is `(! -3 2)`, which evaluates to the right-hand side (argument 2) of formula -3. If this formula is "`x!1 = cos(a!1)`," then the string form of the location reference is "`cos(a!1)`." Adding index values reaches deeper into the formula, e.g., `(! -3 2 1)` evaluates to "`a!1`." Breadth can be achieved as well as depth; `(! -3 *)` evaluates to a list having one element for each side of the formula.

Strictly speaking, formula numbers and symbols are also location references, albeit in shorthand form. In fact, the extended expression `4` is equivalent to `(! 4)`. This establishes the base case for the definition. Indices determine which paths will emanate from this base expression.

Index values and directives $\{i_j\}$ may assume one of the following forms:

- An integer i in the range $1, \ldots, k$, where k is the arity of the operator or function at the current point in the expression tree. Paths follow the i^{th} branch or argument. If i is the last index (i_n), the value returned for the location reference is the i^{th} argument of the current subexpression. Negative integers allow indexing from right to left, that is, -1 selects argument k, -2 selects $k-1$, etc.

- One of the symbols `L` or `R`, which denote the index values 1 and 2, leading paths through the first or second branch accordingly.

- The index value 0, which returns the function symbol of the current expression, provided it is a function application. In a higher-order function application, the function itself can be an expression, as in $f(x)(y)$. Indices after the 0 can be supplied to retrieve components of the function expression. Consistent with the convention for negative indices, index value $-k$ for the application of a function with arity k is equivalent to index 0.

- The *wild-card* symbol `*`, which indicates that this path should be replicated n times, one for each argument expression. The values returned are those generated by all n of the paths.

- A list `(j1 ... jm)` of integers indicating which argument paths should be included for replication, i.e., a subset of the `*` case.

- A complement form `(^ j1 ... jm)` that indicates all argument paths should be followed except those in $\{j_k\}$.

- One of the *deep wild-card* symbols {`-*`, `*-`, `**`}, which indicates that this path should be replicated as many times as needed to visit all nodes in the current subtree. The values returned are the leaf objects (terminal nodes) for `-*`, the nonterminal nodes for `*-`, and all nodes (subexpressions) for `**`.

- A text string serving as a *guard* to enable continuation of the current path(s). If the function or operator symbol of the current subexpression is equal to the string, path elaboration continues. Otherwise, the path is terminated and an empty list is returned. Guards act to select desired paths from multiple candidates.

- A list `(s1 ... sk)` of strings that serves as a guard in the form of patterns to be matched in the manner of Section 5.3.

- An extended expression form `(<symb> ...)` serving as a path guard. The `<ext-expr>` or `<fnums>` starting point should be omitted from the form. Instead, the current subexpression is implicitly supplied as the starting point. Example: `(? "=")`.

- A form `(-> g1 ... gk)` that serves as a *go-to* operator to specify a systematic search down and across the subtree until the first path is found having intermediate points satisfying all the guards $\{g_i\}$ in sequence. The selected path generates the final value. Each guard g_i may be either a string or list of strings, with meanings as described above.

- A form `(->* g1 ... gk)` that behaves the same as `(-> ...)` except that all eligible paths are found and returned as values.

We note a few fine points about these features. Infix and prefix function applications are considered equivalent for indexing purposes; in both "`x!1 * y!1`" and "`atan(x!1, y!1)`," `x!1` is argument 1 and `y!1` is argument 2. Out of range index values cause termination of a path and an empty return value. The same is true of index lists that "fall off the end" of a path by supplying too many indices. The deep wild-cards {`-*`, `*-`, `**`} may be followed by other indicators, which use the various subexpressions as their starting points. During a tree search, backtracking is performed as needed so that the go-to operators `->` and `->*` find any (upper level) paths that meet the indexing specification. If there are nested applications of a function, for example, only the upper-most subexpression will be returned.

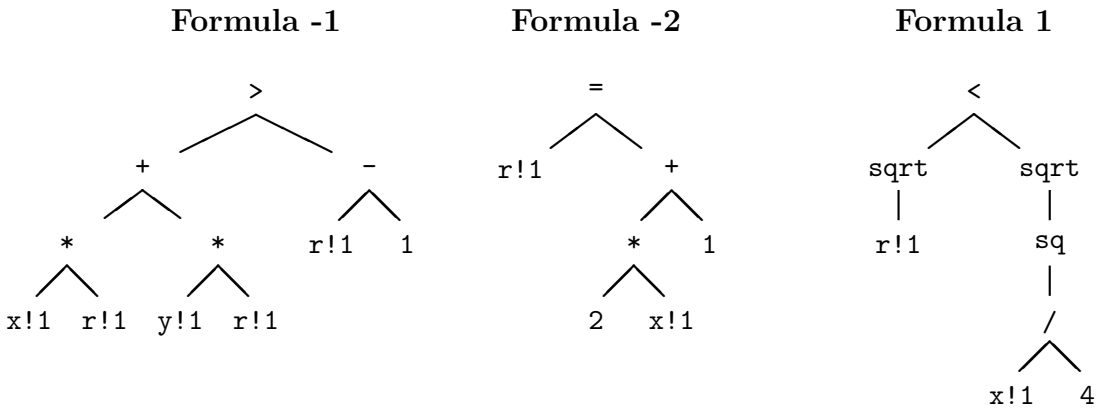

Figure 1: Expression trees for formulas in Table 3.

A few special indexing cases exist for arithmetic expressions. They result in some apparent "flattening" of the parse tree during traversal. The conventions make indexing more convenient for arithmetic terms and correspond more closely to our usual algebraic intuition for numbering terms. The conventions are as follows.

- Additive terms, i.e., terms that are arguments of a + or − operator, are counted left to right irrespective of the associative groupings that may be in effect. They are treated as if they were all arguments of a single addition/subtraction operator of arbitrary arity.
- Multiplicative terms, i.e., terms that are arguments of a ∗ operator, are counted left to right irrespective of the associative groupings that may be in effect. They are treated as if they were all arguments of a single multiplication operator of arbitrary arity.

Parentheses for these associative operators are effectively ignored during the flattening process, e.g., for the three expressions "`x * (y * z)`", "`x * y * z`", and "`(x * y) * z`", term 2 is `y` in each case.

Manip 1.2 introduced a few small changes to location references. One is the addition of a `!!`-variant to suppress index flattening as described above. In the form `(!! <ee> i1 ... in)`, index values $i_1, \ldots, i_n$ follow the unmodified branching of parse trees. Another change is that repeated function names in a go-to form, such as `(! 2 (->* "sq" "sq"))`, will descend to lower expression(s). The second occurrence of the guard `"sq"` can only match a subexpression below the first occurrence. Also, deep wild-cards now visit and include function names/expressions (in addition to argument expressions) during node traversal.

We illustrate the formulation of location references using the notation just described. Table 3 gives the result of evaluating location references with respect to the formulas shown beneath the table. Figure 1 depicts the expression trees for these formulas.

Combinations of indexing directives offer useful ways to find multiple expressions. For instance, `(! * R "+")` finds all right-hand sides having the form of an addition. Similarly, `(! + (->* "cos") 1)` finds all arguments of the `cos` function in the consequent formulas. Another example is `(! - "=" L)`, which finds the left-hand sides of antecedent equalities.

Table 3: Examples of location reference expressions.

Location reference	Expression strings
`(! -2)`	`r!1 = 2 * x!1 + 1`
`(! -2 L)`	`r!1`
`(! -2 R)`	`2 * x!1 + 1`
`(! -2 R 1)`	`2 * x!1`
`(! -2 R 2)`	`1`
`(! -2 R 1 2)`	`x!1`
`(! -1 L 2 1)`	`y!1`
`(! 1 R 1)`	`sq(x!1 / 4)`
`(! 1 R 1 1)`	`x!1 / 4`
`(! 1 R 1 1 2)`	`4`
`(! -2 *)`	`r!1, 2 * x!1 + 1`
`(! -1 L 2 *)`	`y!1, r!1`
`(! -1 L * 1)`	`x!1, y!1`
`(! -1 L * *)`	`x!1, r!1, y!1, r!1`
`(! -1 L (^ 1))`	`y!1 * r!1`
`(! -2 R -*)`	`*, *, 2, x!1, 1`
`(! 1 R -*)`	`sqrt, sq, x!1, 4`
`(! -2 R **)`	`2 * x!1 + 1, +, 2 * x!1, *, 2, x!1, 1`
`(! 1 R 1 **)`	`sq(x!1 / 4), sq, x!1 / 4, /, x!1, 4`
`(! - "=")`	`r!1 = 2 * x!1 + 1`
`(! -2 * "+")`	`2 * x!1 + 1`
`(! 1 (-> "sqrt"))`	`sqrt(r!1)`
`(! 1 (->* "sqrt"))`	`sqrt(r!1), sqrt(sq(x!1 / 4))`
`(! 1 (-> "sq"))`	`sq(x!1 / 4)`
`(! 1 (-> "sq") 1)`	`x!1 / 4`
`(! -1 (-> "+") *)`	`x!1 * r!1, y!1 * r!1`
`(! -1 (->* "+" "*") *)`	`x!1, r!1, y!1, r!1`
`(! 1 *- 0)`	`<, sqrt, sqrt, sq, /`

where the formulas are as follows:

```
{-1}  x!1 * r!1 + y!1 * r!1 > r!1 - 1
[-2]  r!1 = 2 * x!1 + 1
  |-------
[1]   sqrt(r!1) < sqrt(sq(x!1 / 4))
```

5.3 Pattern Matching

[Note: The features in this section were introduced in the first version of Manip. They have been partially superseded by newer features, in particular, the syntax matching capabilities added in version 1.2 (Section 7).]

Recall that a pattern match is specified using the form `(? <ext-expr> p1 ... pn)`. Each pattern p_j is expressed as a text string using a specialized pattern language. Unlike location references, pattern matches usually produce only a text string and lack a corresponding CLOS object for a PVS expression. The patterns $p_1, \ldots, p_n$ are applied in order to the textual representation of each member of the base expression list. In each case, matching stops after the first successful match among the $\{p_j\}$ is obtained. All resulting output strings are collected and concatenated into a single list of output strings.

5.3.1 Pattern Language

The pattern language was designed to meet the anticipated needs of prover users in describing PVS expressions. Pattern matching features are implemented using a modest regular expression package, which provides limited regular expression features, generally less sophisticated than Perl-style regular expressions. Nevertheless, it appears to be adequate for the purpose at hand.

A pattern string may denote either a *simple* or a *rich* pattern. Simple patterns are easier to express and are expected to suffice for many everyday matching applications. When more precision is required, rich patterns may be used for more expressive power.

No alternation is provided in the pattern language itself. To achieve the effect of alternation, multiple pattern strings may be supplied instead of a single pattern. Each pattern in the list is tried in sequence until a match is obtained. Thus the output strings issue from the first pattern to produce a nonempty result.

An empty list of patterns will match no strings. A null pattern (`""`), however, matches any string but returns no useful values. Typically, various substrings are extracted and returned as the result of the matching process. Successful matches that return no output strings result in the default value of a single empty string.

5.3.2 Simple Patterns

Simple patterns allow matching against literal characters, whitespace fields, and arbitrary substrings. Pattern strings comprise a mixture of literal characters and meta-strings for designating text fields. Each literal character must match itself in the target string. Each field designator matches a string of zero or more characters in the target string.

Meta-strings denote either whitespace fields or non-whitespace fields. A whitespace field is indicated by a space character in the pattern, which stands for a field of zero or more whitespace characters (space, tab, form feed, or newline). A non-whitespace field is a meta-string consisting of the percent (`%`) character followed by a digit character (0–9). Such a field matches zero or more arbitrary characters in the target string. Both capturing and non-capturing fields are provided. A capturing field causes the matching substring to be returned as an output string.

The meta-string `%0` denotes a noncapturing field, while those with nonzero digits are capturing fields. If a nonzero digit d is the first occurrence of d in the pattern, a new capturing field is thereby indicated. Otherwise, it is a reference to a previously captured field whose contents must be matched. Note that the nonzero digits used must be consecutive starting with 1 (e.g., `"%1 = %3"` is improper).

Table 4: Examples of simple pattern matching.

Pattern	Matching string(s)	Captured fields
`(? 1 "%1(r!1)")`	`sqrt(r!1)`	`sqrt`
`(? 1 "sqrt(sq(%1))")`	`sqrt(sq(x!1 / 4))`	`x!1 / 4`
`(? -2 "r!1 = %1")`	`r!1 = 2 * x!1 + 1`	`2 * x!1 + 1`
`(? 1 "%1(%0) <")`	`sqrt(r!1) <`	`sqrt`
`(? -1 "> %1 - %0")`	`> r!1 - 1`	`r!1`
`(? 1 "%1(%0) < %1(%2)")`	All of formula 1	`sqrt, sq(x!1 / 4)`
`(? (! (-2 1) R) "%1x!1")`	`2 * x!1`, `sqrt(sq(x!1`	`2 * `, `sqrt(sq(`

where the formulas are as follows:

```
{-1}  x!1 * r!1 + y!1 * r!1 > r!1 - 1
[-2]  r!1 = 2 * x!1 + 1
  |-------
[1]   sqrt(r!1) < sqrt(sq(x!1 / 4))
```

We illustrate the formulation of simple patterns using the notation just described. Table 4 shows the result of matching various patterns against the sample formulas.

5.3.3 Rich Patterns

Rich patterns follow the same basic approach as simple patterns, but add extensions for multiple matching types and multiple text field types. To be distinguished from simple patterns, rich patterns must begin with the character '`@`'. To specify the type of matching requested, the second character of the pattern encodes the user's choice. Thus a rich pattern has the form `@<match type><pattern string>`.

Table 5 shows the match types currently offered. Note that the default (partial string match) can be obtained by omitting the match type code, in which case the second character is interpreted as part of the pattern string. Obviously, this will not work if the first character of the pattern string is one of the match type encodings.

In a full string match, the pattern must match against the entire text string under consideration. A partial string match is less strict, admitting any substring that satisfies the pattern. Generally, the left-most substring with the largest extent is chosen for a partial match.

The type-`s` match allows a partial match to determine a boolean outcome, then returns the full input string as the result if successful. In effect, this lets matching be used as a filter to allow all or none of the string to pass. None of the strings captured via `%1`, `%2`, etc., will be included in the result. Also returned is the CLOS object for the input string, where applicable. This allows the result of a type-`s` match to be used as input to a location reference.

Expression-oriented matching is also provided, which allows matching to proceed with respect to the parse tree of an expression. In top-down matching, a pre-order traversal of the tree is performed where matching is attempted at each visited node. If the textual representation of the expression denoted by the node matches the pattern, traversal stops and returns the match

Table 5: Character encoding for match types.

Character	Match type
`f`	Full string match
`p`	Partial string match (first substring to match)
`s`	Partial match returning full string if successful
`t`	Top-down expression matching
`b`	Bottom-up expression matching
`<digit>`	Top-down expression matching, skipping top-most `<digit>` levels
Other	Partial string match

Table 6: Character encoding for text field types.

Character	Field type
`*`	Zero or more arbitrary characters
`+`	One or more arbitrary characters
`&`	One or more arbitrary characters, where the first and last are non-whitespace characters
`i`	PVS identifier (allows `!` for prover variables)
`#`	Numeric field (digits only)
Other	Same as `*`

result. Otherwise, matching is attempted on each subexpression in left-to-right order. Currently, the most common syntactic features of PVS expressions are accommodated, e.g., infix and prefix function applications, but not all language features are included. If a pattern does not match as expected, it might be due to this incompleteness in the current implementation. See the syntax matching features in Section 7 for a more capable way to match elements of the PVS grammar.

Bottom-up expression matching (post-order traversal) may be requested as well as top-down matching. In addition, restricted top-down matching may be performed by skipping the top few levels of the expression tree. This is useful to avoid undesired matches caused by greedy matching of parenthesized expressions. Naturally, complex formulas can give rise to expensive searches when these expression-oriented forms of matching are used.

Capturing and non-capturing text fields are extended in rich patterns to allow multiple field types. The basic field designator is extended to a three-character sequence of the form `%<digit><field type>`. Table 6 shows the field types currently offered. If the field type character is omitted, the default type is `*`, which is the same as the field type for simple patterns. Field types in rich patterns enable more discriminating searches than those of simple patterns.

To illustrate the use of these extended pattern features, Table 7 shows the result of matching

Table 7: Examples of rich pattern matching.

F#	Pattern	Matching string	Captured fields
-2	`@pr!1 = %1&`	`r!1 = 2 * x!1 + 1`	`2 * x!1 + 1`
-2	`@p%1# * %2i`	`2 * x!1`	`2, x!1`
-2	`@s%1# * %2i`	`2 * x!1`	All of formula -2
-2	`@f%1# * %2i`	None	
-2	`@p%1# * %1`	None	
1	`@p%1i(r!1)`	`sqrt(r!1)`	`sqrt`
1	`@p%1i(%0*)`	`sqrt(r!1)`	`sqrt`
1	`@s%1i(%0*)`	`sqrt(r!1)`	All of formula 1
1	`@psqrt(sq(%1*))`	`sqrt(sq(x!1 / 4))`	`x!1 / 4`
-1	`@p%1i - %0*`	`r!1 - 1`	`r!1`
1	`@tsq(%1*)`	`sq(x!1 / 4))`	`x!1 / 4)`
1	`@1sq(%1*)`	`sq(x!1 / 4))`	`x!1 / 4)`
1	`@2sq(%1*)`	`sq(x!1 / 4)`	`x!1 / 4`
1	`@bsq(%1*)`	`sq(x!1 / 4)`	`x!1 / 4`
-1	`@1%1& * %2&`	`x!1 * r!1 + y!1 * r!1`	`x!1 * r!1 + y!1, r!1`
-1	`@2%1& * %2&`	`x!1 * r!1`	`x!1, r!1`
-1	`@b%1& * %2&`	`x!1 * r!1`	`x!1, r!1`
1	`@f%1i(%0*) < %1(%2*)`	All of formula 1	`sqrt, sq(x!1 / 4)`
-2	`@p%1&=%2&`	None	

where the formulas are as follows:

```
{-1}  x!1 * r!1 + y!1 * r!1 > r!1 - 1
[-2]  r!1 = 2 * x!1 + 1
  |-------
[1]   sqrt(r!1) < sqrt(sq(x!1 / 4))
```

various rich patterns against the sample formulas.

6 General Purpose Strategies

This section describes a set of general purpose PVS prover strategies for manipulating arbitrary sequents. They are not specialized for arithmetic. Some offer generic capabilities useful in implementing other strategies for specific purposes. Table 8 lists the strategies provided along with their formal argument lists.

Often prover users would like ways to capture expressions from the current sequent and use them to build arguments to prover commands such as `case`. We have provided extended expressions to achieve this capture. Next we add a parameter substitution technique to yield a major new way to formulate prover commands. To complete the suite, we add a family of higher-order strategies that substitute strings and formula numbers into a parameterized

Table 8: Summary of general purpose strategies.

Syntax	Function
`(invoke command &rest expr-specs)`	Invoke command by instantiating from expressions and patterns
`(for-each command &rest expr-specs)`	Instantiate and invoke separately for each expression
`(for-each-rev command &rest expr-specs)`	Invoke in reverse order
`(show-subst command &rest expr-specs)`	Show but don't invoke the instantiated command
`(claim cond &opt (try-just nil) &rest expr-specs)`	Claims condition on terms
`(name-extract name &rest expr-specs)`	Extract & name expr, then replace
`(move-to-front &rest fnums)`	Reorder sequent formulas
`(rotate--)`	Rotate antecedent list
`(rotate++)`	Rotate consequent list
`(use-with lemma &rest fnums)`	Use a lemma with formula preferences for instantiation
`(apply-lemma lemma &rest expr-specs)`	Use lemma with expressions
`(apply-rewrite lemma &rest expr-specs)`	Rewrite with expressions
`(unwind-protect main-step cleanup-step)`	Invoke step, then cleanup actions, even if main goal was proved

command (rule or strategy). The command can be regarded as a template expression (actually, a Lisp *form*) in which embedded text strings and special symbols can serve as formal parameters for substitution.

Consider a simple example. Suppose formula 2 is

```
sin(2 * PI * omega!1 + delta!1) >= 0
```

and we wish to claim that the `sin` argument is nonnegative. The command

```
(invoke (case "%1 >= 0") (! 2 L 1))
```

accomplishes this task by invoking the prover command

```
(case "2 * PI * omega!1 + delta!1 >= 0")
```

as if it had been typed in this form.

The reasons for having such a capability, along with the extended expressions of Section 5, are to:

- Reduce the amount of typing or other data entry required, such as that described in Section 2.

- Make it easy to construct new expressions using (possibly large) fragments of the current sequent as subexpressions.
- Reduce the brittleness of proofs by avoiding the inclusion of complete expressions from the current sequent in stored proof steps.
- Make it easy to construct specialized strategies that are instances of more general ones.

We envision the higher-order strategies as being more useful during later stages of proof development, especially when finalizing a proof to make the permanent version more robust. During preliminary stages of a proof, it is easier to work directly with the actual expressions. Once the outline of a proof is firm, pattern matching and location reference features can be introduced to abstract away excessive detail.

6.1 Parameter Substitution

The outcome of evaluating an extended expression can be used to carry out textual and symbolic substitutions within a parameterized command. Such a command is assumed to be a Lisp form:

```
(<rule/strategy> <argument 1> ... <argument n>)
```

The argument expressions can be numeric, textual, or symbolic values as well as parenthesized expressions. This can lead to nesting of arbitrary depth. As usual at the prover interface, neither the top-level parenthesized expression nor its arguments are evaluated as normal Lisp expressions. The interpretation of arguments is left for the command to carry out when it is finally invoked.

Input data for the substitution process is a list of expression descriptors computed during the evaluation of one or more extended expression specifications. As described in Section 5, each descriptor contains a text string and, optionally, a formula number and CLOS object. The descriptor list is the source of substitution data while the parameterized command is its target.

Within this framework, we allow two classes of substitutable data: literal text strings and Lisp symbols. The top-level parenthesized expression is traversed down to its leaves. Wherever a string or symbol is encountered, a substitution may be performed. The final command thus produced will be invoked as a prover command in the manner defined for the chosen higher-order strategy. (Lisp programmers can think of this process as evaluating a backquote expression with specialized implicit unquoting.)

Parametric variables for substitution are allowed as follows. Within literal text strings, the substrings `%1`, ..., `%9` serve as implicit text variables. The substring `%1` will be replaced by the string component of the first expression descriptor. The other `%`-variables will be replaced in order by the corresponding strings of the remaining descriptors.

Aggregations may be obtained using the string `%*` and its variants. `%*` will be replaced by a concatenation of all expression strings. `%,` behaves the same except that it separates the strings using the delimiter "`,␣`".

Certain reserved symbols beginning with the `$` character are provided to serve as substitutable symbolic parameters. Such symbols are not embedded within string constants as are the `%`-variables; they appear as stand-alone symbols within the list structure of the parameterized command. The symbols `$1`, `$2`, etc., represent the first, second, etc., expression descriptors from the list of available descriptors. These symbols should be used as arguments to strategies

Table 9: Special symbols for command substitution.

Symbol	Value
`$1, $2, ...`	nth expression descriptor
`$*`	List of all expression descriptors
`$1s, $2s, ...`	nth expression string
`$*s`	List of all expression strings
`$1n, $2n, ...`	Formula number for nth expression
`$*n`	List of all formula numbers
`$1j, $2j, ...`	CLOS object for nth expression
`$*j`	List of all CLOS objects
`$+, $+s, $+n, $+j`	Duplicate-free versions of `$*, $*s, $*n, $*j`

that require a location-reference type of extended expression. They may be used as arguments for strategies in this package whenever terms or formula numbers are called for.

Variants of these symbols exist to retrieve the text string, formula number, and CLOS object components of a descriptor. These are needed to supply arguments for built-in prover commands, which are not cognizant of extended expressions. The symbols `$1s`, `$1n` and `$1j` serve this purpose. Note that CLOS object values have no use when entering prover commands from the keyboard. They are provided for the convenience of strategy writers.

Aggregations may be obtained using the symbol `$*` and it variants. A list of all source expression descriptors is given by `$*` while the list of strings and formula numbers is given by `$*s` and `$*n`. Because one formula might be associated with multiple expressions, the descriptor list can contain duplicate formula numbers. A list without duplicates is available from the symbol `$+n`. Table 9 summarizes the special symbols usable in substitutions.

We note that when using the list-valued symbols, their values are "spliced" into the surrounding Lisp expression. If they are used in a context that requires parentheses, they need to be supplied by the user. For example, if `$+n` has the value `(1 3 5)`, then `(hide $+n)` will be expanded to `(hide 1 3 5)`. Conversely, `(hide-all-but ($+n))` will be expanded to `(hide-all-but (1 3 5))`. In the following sections we present more examples of how the `%`-variable and `$`-variable substitutions are applied to produce a final instantiated command.

6.2 Invocation Strategies

The following higher-order strategies make use of the parameter instantiation features to construct and invoke prover command instances.

`invoke` *command* `&rest` *expr-specs* [*Strategy*]

This strategy is used to invoke *command* (a rule or strategy) after applying substitutions extracted by evaluating the expression specifications *expr-specs*. All expression descriptor lists are appended to form a single list before substitution occurs. Note that there is not a one-to-one correspondence between descriptors and expression specifications. Each specification can produce zero or more descriptors.

As an example, suppose formula 3 is

```
f(x!1 + y!1) <= f(a!1 * (z!1 + 1))
```

Then the command

```
(invoke (case "%1 <= %2") (? 3 "f(%1) <= f(%2)"))
```

would apply pattern matching to formula 3 and create the bindings `%1` = `"x!1 + y!1"` and `%2` = `"a!1 * (z!1 + 1)"`, which would result in the prover command

```
(case "x!1 + y!1 <= a!1 * (z!1 + 1)")
```

being invoked. An alternative way to achieve the same effect using location referencing is the following:

```
(invoke (case "%1 <= %2") (! 3 * 1))
```

String substitutions are not limited to command arguments that accept PVS language expressions. They may also be used to construct function, lemma and theory names.

As another example, suppose we wish to hide most of the formulas in the current sequent, retaining only those that mention the `sqrt` function. We could search for all formulas containing a reference to `sqrt` using a simple pattern, then collect all the formula numbers and use them to invoke the `hide-all-but` rule. Applying `invoke` as follows

```
(invoke (hide-all-but ($+n)) (? * "sqrt"))
```

would hide all formulas except those containing the string `sqrt`.

`for-each` *command* `&rest` *expr-specs* [*Strategy*]

This strategy is used to invoke *command* repeatedly, once for each expression generated by *expr-specs*. The effect is equivalent to applying `(invoke command <expr i>)` n times.

As an example, suppose we wish to expand every function in the consequent formulas that has the expression "`n!1 + 1`" as its argument. The following command carries this out, assuming there is only one such expression per formula.

```
(for-each (expand "%1") (? + "@p%1i(n!1 + 1)"))
```

`for-each-rev` *command* `&rest` *expr-specs* [*Strategy*]

This strategy is identical to `for-each` except that the expressions are taken in reverse order.

As an example, suppose we wish to find all the antecedent equalities and use them for replacement, hiding each one as we go. This needs to be done in reverse order because formula numbers will change after each replacement.

```
(for-each-rev (replace $1n :hide? t) (! - "="))
```

`show-subst` *command* `&rest` *expr-specs* [*Strategy*]

This strategy does not invoke any commands, but applies the matching and substitutions as the strategy `invoke` would. The instantiated command is displayed so the user can see the result of substitutions without actually attempting any proof commands. The idiom

```
(show-subst ($*) <ext expr 1> ... <ext expr n>)
```

allows a convenient display of the descriptors produced by evaluating extended expressions. Tweaking the expressions and iterating enables the user to converge on a correct formulation before invoking an actual prover command.

`claim` *cond* `&optional` (*try-just* `nil`) `&rest` *expr-specs* [*Strategy*]

The `claim` strategy is basically the same as the primitive rule `case`, except that the formula expression is derived using the parameterization technique described in Section 5. It also differs by being limited to only two-way case splitting. The condition presented in argument *cond* is a parameterized string expression of the kind described in Section 6.1. It may be instantiated by the terms found in the `&rest` argument *expr-specs*. For example, to claim that a numerical expression lies between two others, we could use something like

```
(claim "%1 <= %2 & %2 <= %3" nil "a/b" "x+y" "c/d")
```

to generate a case split on the formula:

```
a/b <= x+y & x+y <= c/d
```

Argument `try-just` allows the user to try proving the justification step (the second case resulting from the case split).

Usage: `(claim "%1 + PI = %2" T "phi!1" "theta!1")` introduces a claim and tries to prove it using `grind`.

`name-extract` *name* `&rest` *expr-specs* [*Strategy*]

Rather than invoking a command, this strategy is used to compute a list of expressions, then extract each expression string from it, assign a name to the expression, and finally, replace the expression by the name. If *expr-specs* evaluates to multiple expressions, unique names are formed by appending an index to *name*. The equality formulas generated by the internal `name-replace` commands are not hidden. This strategy is useful for removing embedded expressions and lifting them to one side of an equality formula, where the various arithmetic manipulation strategies can be applied thereafter.

Usage: `(name-extract angle (? 3 "2 * sin(%1)"))` applies the pattern to find the argument to the `sin` function, giving it the name ANGLE, then replaces it throughout the sequent.

6.3 Substitution Shortcuts

To streamline user input for simple cases, we provide the following shortcuts usable during the substitution process.

- *Embedding extended expressions in strings.* Commands such as

  ```
  (claim "%1 < %2" nil (! 1 2) (! 3 4))
  ```

  may be rewritten to a form that embeds the extended expressions in the target string:

  ```
  (claim "%! 1 2% < %! 3 4%")
  ```

Location references may be embedded by replacing the outer-most parentheses with percent characters. After evaluation, the first expression string generated by each location reference will replace the corresponding `%!...%` substring. Concurrent use of `%`-variables in the same string is possible. Embedding pattern match expressions is also possible but not recommended because of the need to escape quote characters.

- *Embedding extended expressions in list structures.* Commands such as

```
(invoke (hide $*n) (? + "cos"))
```

may be rewritten to a form that embeds the extended expressions in the target list:

```
(invoke (hide (? + "cos")))
```

Either location references or pattern match expressions may be embedded this way. The effect is to extract the formula numbers yielded by the evaluation and substitute them for the (! ...) or (? ...) sublist. If the results need to be contained in a single list argument to a rule, add an extra set of parentheses, as in:

```
(invoke (hide-all-but ((? + "cos"))))
```

These shortcuts also support the newer syntax-matching form `(~ ...)` introduced in version 1.2 (Section 7.7). Bear in mind that the shortcuts apply only in limited contexts. For greater flexibility, use the general substitution mechanism as described in Sections 6.1 and 6.2.

6.4 Defining New Strategies

Invocation strategies are useful as building blocks for more specialized strategies that users might need for particular circumstances. Extended expressions can support an alternative to the more code-intensive strategy-writing style that requires accessing the data structures (CLOS objects) representing PVS expressions. The invocation strategies can make writing lightweight strategies more accessible to users without a deep background in Lisp programming.

Consider a simple example. We wish to automate a specialized type of backward chaining process. Suppose a consequent formula exists having the form `f(e1) <= f(e2)` for two expressions e_1 and e_2. If f is monotonic and we know we can prove that $e_1 \leq e_2$, this would suffice to establish the consequent formula. So we would like to back-chain on this goal to produce the new goal `e1 <= e2`. The following strategy definition accomplishes this task by applying the pattern matching features.

```
(defstep backchain-leq (fnum)
  (let ((case-step
         `(invoke$ (case "%2 <= %3") (? ,fnum "@f%1i(%2*) <= %1(%3*)"))))
    (branch case-step ((assert) (skip))))
  "Backchain on inequality in FNUM for monotonic function."
  "~%Backchaining on inequality in formula ~A")
```

The pattern recognizes the desired inequality form for an arbitrary function and extracts the embedded arguments. A `case` rule invocation is constructed using these expressions. Of the two goals produced by the `case` rule, one is simplified using `assert`, while the other is the main branch left for the user to continue.

6.5 Formula Reordering Strategies

The next group of strategies includes several for manipulating the order of formulas within a sequent. Formula reordering can be helpful before instantiating quantifiers using `inst?` or applying lemmas via the `use` rule. It also can be helpful as a component of higher level strategies where uniform placement of formulas is needed.

`move-to-front &rest` *fnums* [*Strategy*]

Invoking `move-to-front` on a list of formulas causes them to be pulled to the front of their respective lists (antecedents or consequents). They remain in the same relative order that they appeared initially regardless of the order in which they are listed in argument *fnums*. Example: `(move-to-front -4 3 -2 2)` causes the new order to become -2, -4, -1, -3 $\vdash$ 2, 3, 1.

`rotate--` [*Strategy*]
`rotate++` [*Strategy*]

These strategies cause the antecedent (`--`) or consequent (`++`) formulas to be "rotated," i.e., the first formula is moved to the end and all the others move up by one.

6.6 Lemma Invocation Strategies

This final group of strategies is used to invoke lemmas in various ways not already provided by the built-in prover commands.

`use-with` *lemma* `&rest` *fnums* [*Strategy*]

The `use` command for importing and instantiating lemmas sometimes chooses wrong or useless variable instantiations. We could improve the chances for correct selection in some cases by reordering the formulas so that preferred terms are tried earlier in the instantiation process. The `use-with` strategy implements this heuristic by creating a temporary copy of the terms in *fnums* and placing it at the front of the sequent (formula -1). Then a `use` command for *lemma* is invoked so that the search for instantiable terms begins with the temporary formula. The effect is to consider terms from the user's preferred formulas (in the order given) before looking elsewhere in the sequent. Instantiation heuristics apply various criteria for suitability so this tactic might not achieve the desired effect.

Usage: `(use-with "sin_gt_0" 3 -2)` tries to instantiate the variables of `sin_gt_0` by first examining the terms of formulas 3 and -2.

`apply-lemma` *lemma* `&rest` *expr-specs* [*Strategy*]
`apply-rewrite` *lemma* `&rest` *expr-specs* [*Strategy*]

Occasionally is it necessary to provide explicit instantiations when applying lemmas or rewrite rules. This happens when the prover's automatic instantiation heuristics fail to pick out the desired expressions. In such cases, these two strategies provide an abbreviated way to force the binding of expressions to lemma variables. It is necessary to know the lemma variable names so that the expressions can be supplied in the correct order. PVS lists lemma variables in alphabetical order when the `inst` command is invoked. This is the order in which expressions

should be supplied in the strategy command. `apply-lemma` has an effect similar to the `use` command. `apply-rewrite` is similar to `rewrite`, although only equality rules are currently handled.

6.7 Utility Strategies

Common Lisp includes the function `unwind-protect` so programmers can mitigate some consequences of run-time failures. In particular, this function arranges for exceptions during expression evaluation to trigger cleanup actions so that important state restorations can be performed in all cases.

A similar need arises during PVS proofs, except the need is to protect oneself from success rather than failure. Sometimes strategies executing on a user's behalf find it necessary to take actions that temporarily change the prover's state. An example is the need to turn on certain rewrites before beginning a proof sequence. Because of the way PVS works, if the current subgoal is proved during this strategy, its execution is terminated and any additional steps it wished to perform, such as turning off rewrites after the proof attempt, will not be carried out, possibly to the detriment of parallel subgoals. To compensate for this behavior, we provide the following strategy.

`unwind-protect` *main-step cleanup-step* [*Strategy*]

The proof step *main-step* is attempted. After this step is finished, whether it proved its goal or not, the *cleanup-step* will be performed. State-changing actions should be invoked within *main-step*, and their complementary actions should be invoked within *cleanup-step*.

7 Syntax Matching

A new type of matching and rule invocation was introduced in version 1.2 that allows users to match against syntactic structures within PVS expressions. Unlike string-based pattern matching (Section 5.3), syntax matching works with parse trees that are implicit in PVS's object representation of formulas. A syntax pattern P is (usually) a string containing a PVS-language expression with optional pattern variables. P will be parsed and its syntax tree forms the pattern to match against formulas' parse trees and their subtrees. Information derived from this process can then be used to initiate proof steps. A new type of extended expression specification based on this feature was added to complement those of Section 5.

For many uses, these concepts offer more powerful and convenient ways to accomplish the tasks described in Sections 5 and 6. A full description of supported features is presented below. Readers wishing to see examples may skip ahead to Section 7.6.

7.1 A Syntax Matching Strategy

A single top-level matching strategy is provided to conduct syntax-based expression search and proof-step generation. This strategy comes in several variants that support a rich variety of user options. Its interface is therefore more complicated than typical prover strategies. Instead of the usual fixed list of arguments, we provide a more flexible way to specify inputs.

`match &rest` *item-specs* [*Strategy*]

This strategy is used to match one or more patterns against the formulas of the current sequent, then instantiate the matching expressions in a parameterized proof rule, and finally, invoke the resulting rule with optional steps for proving the branches. The list *item-specs* is a free-form sequence of symbols, strings and parenthesized Lisp expressions for specifying patterns and intended actions. Parsing of these items is performed internally within the `match` strategy.

The general form is

`(match` [`?`] [*fnums*] $P_1 \ldots P_n$ [*onums*] [*action*] [$T_1 \ldots T_k$] [`!` $S_1 \ldots S_{k+2}$] `)`

where the various items have meaning as follows:

- The optional `?` symbol, which also may be written `show`, indicates that the action should not be performed. Instead, information about matching instances and generated expressions will be displayed, showing what proof step would be executed.
- The *fnums* field is used to specify which formula numbers should be searched by the pattern matcher. Formula numbers may be specified in the usual way, i.e., as explicit numbers or lists of same or one of the symbols {`+`, `-`, `*`}. The entire sequent (`*`) is the default search range.
- The list $P_1, \ldots, P_n$ contains one or more syntax patterns. Details of pattern specification are presented below (Section 7.4).
- To indicate which of several possible matches should be considered the chosen instance(s), the field *onums* allows the stipulation of occurrence numbers. The default value is 1, meaning the first matching occurrence should be picked. A number, a list of numbers, or the symbol `*` may be supplied for *onums*.
- If supplied, the *action* value should be either: one of the symbols {`rep`, `case`, `step`}; one of the PVS function symbols used as binary relational or boolean operators; or the name of a prover rule or strategy.
- The template expressions $T_1, \ldots, T_k$ should be either strings or parenthesized s-expressions. Typically they contain embedded template variables that will receive substitutions from the pattern matches. Details are provided below (Section 7.5).
- After the delimiter symbol `!` there can appear proof steps $S_1, \ldots, S_m$, where $m \leq k + 2$. These can be either parenthesized s-expressions, optionally containing embedded template variables, or numbers used as abbreviations for proof rules (see Table 10). Numeric values provide a coarse scale to indicate how strenuously a proof should be tried.

An optional comment may be inserted at the end of a `match` invocation. The symbol "`--`" serves as the beginning of a comment, as in the following example:

```
(match lambda * -- "find all lambda expressions")
```

Any Lisp forms after the `--` delimiter will be ignored, except that they remain with the proof step and are retained in the proof file.

Table 10: Numeric codes or abbreviations for branch steps.

Code	Proof Rule	Code	Proof Rule
0	`(skip)`	5	`(ground)`
1	`(prop)`	6	`(smash)`
2	`(bddsimp)`	7	`(bash)`
3	`(simplify)`	8	`(reduce)`
4	`(assert)`	9	`(grind)`

7.2 Principal Forms

Depending on the value of the *action* field in the item specification list, there are several possible variants in the use of the `match` strategy.

- `(match` [*fnums*] $P_1 \ldots P_n$ [*onums*] `)`
  In the simplest variant, no action is present, only patterns. The effect is to conduct matching and display the results. Figure 2 shows an example. No change to the proof state occurs as this case is equivalent to the proof command `(skip)`. This `match` form is useful for collecting and examining similar subexpressions from a complex sequent.

- `(match` [`?`] [*fnums*] $P_1 \ldots P_n$ [*onums*] `rep` [T] [`!` $S_1 \ldots S_3$] `)`
  Equality replacement is performed when *action* has value `rep`. The first expression matched by pattern P_1 is replaced by the expression derived from template T. If T is absent, the second matching pattern instance P_2 is used instead. Introducing optional proof steps S_i is discussed in Section 7.3.

- `(match` [`?`] [*fnums*] $P_1 \ldots P_n$ [*onums*] *op* [T] [`!` $S_1 \ldots S_3$] `)`
  If *op* is a PVS relational operator or binary boolean operator, this variant will perform a case split on LHS *op* RHS, where LHS and RHS are derived as for `rep`.

- `(match` [`?`] [*fnums*] $P_1 \ldots P_n$ [*onums*] `case` $T_1 \ldots T_k$ [`!` $S_1 \ldots S_{k+2}$] `)`
  An arbitrary `case` rule may be formulated by introducing multiple templates $T_1 \ldots T_k$, one for each case expression. Each may have a separate proof step in $S_1, \ldots, S_k$.

- `(match` [`?`] [*fnums*] $P_1 \ldots P_n$ [*onums*] *rule* [`!` $S_1\, S_2$] `)`
  An existing proof rule or strategy may be invoked on the expression instances $E_1, \ldots, E_m$ that match patterns $P_1, \ldots, P_n$. The step `(`*rule* $E_1 \ldots E_m$`)` will be constructed, which is not usable with many built-in prover rules, but could be of benefit with strategies.

- `(match` [`?`] [*fnums*] $P_1 \ldots P_n$ [*onums*] `step` T `)`
  Finally, the `step` form is the most general, allowing template T to be instantiated as desired to create an arbitrary proof step to submit to the prover.

```
Rule? (match -2 "deriv(%)" *)
[Manip.match]

Pattern Matches:
 {-2} %1 : deriv(LAMBDA (x: real_abs_lt1):
                   deriv(atanhN(n!1))(x) -
                    x * x * deriv(atanhN(n!1))(x))
 {-2} %2 : deriv(atanhN(n!1))
 {-2} %3 : deriv(atanhN(n!1))
 {-2} %4 : deriv(deriv(atanhN(n!1)))
 {-2} %5 : deriv(atanhN(n!1))
 {-2} %6 : deriv(deriv(atanhN(n!1)))
 {-2} %7 : deriv(atanhN(n!1))
 {-2} %8 : deriv(atanhN(n!1))

No change on: (match -2 "deriv(%)" *)
atanh_taylors_prep6.1.1.1.1.1.1 :
...

[-2]  deriv(LAMBDA (x: real_abs_lt1):
               deriv(atanhN(n!1))(x) - x * x * deriv(atanhN(n!1))(x))
       =
       (LAMBDA (x: real_abs_lt1):
          deriv(deriv(atanhN(n!1)))(x) -
           x * x * deriv(deriv(atanhN(n!1)))(x))
        + (LAMBDA (x: real_abs_lt1): -2 * (x * deriv(atanhN(n!1))(x)))
```

Figure 2: Example of a display-only invocation of `match`.

7.3 Control of Proof Steps

In most forms of Section 7.2, optional proof steps may be specified to advance the proof along the branches created by the main action. Control is provided by the Manip strategy `branch-back`, which was added in version 1.2 to support `match`.

`branch-back` *step steplist* [*Strategy*]

The purpose of `branch-back` is to invoke a branching *step* and initiate subproofs for the branches with automatic backtracking for any steps that fail to completely prove their goals. It is similar to the built-in strategy `branch`, except that the elements $S_1, \ldots, S_n$ of *steplist* are attempted in an all-or-nothing fashion. If step S_i for a branch fails to prove its goal completely, the proof state for branch i is rolled back to the point before S_i was invoked.

Branch steps are typically used to prove the justification for a main action or any TCCs that get generated as a by-product of the main action. It is helpful to apply aggressive strategies such as `grind` in such cases. When they finish, however, they can leave the proof state for a branch unrecognizable to the user. It often better to have an atomic, transactional type of behavior that rolls back the state on failure.

In the forms of Section 7.2, if there are k template expressions, up to $k + 2$ steps may be supplied. The first k steps are used for proofs that arise from the k main branches, and default

to `(skip)`. Step $k+1$ is used for the "else" branch of case splits, and defaults to `(grind)`. Step $k+2$ is used for any additional branches spawned by TCCs, and defaults to `(assert)`.

7.4 Pattern Specifications

Syntax patterns are written using one of the following forms:

- *A symbol denoting a syntax class.* The pattern matches any PVS expression of the designated class. Type expressions are allowed as well. Tables 11 and 12 list the syntax classes supported, which currently include every top-level expression category other than `TABLE`. Partial support is provided for `COND`. Example: `lambda`.

- *A PVS expression in string form with optional pattern variables.* The expression must be a syntactically well-formed PVS expression, except for the pattern variables, which begin with "`%`" and can appear wherever an identifier would be allowed. The expression needs to be parseable after pattern variables are substituted by valid identifiers. Example: `"sin(%a)"`.

- *One of the symbols* `&0`, `&1`, *..., denoting repetition of the previous pattern.* A symbol `&`k is expanded into a full pattern that is a copy of the previous pattern with its occurrence count incremented by k. Thus, `&1` denotes the second occurrence, `&2`, the third, etc.

- *A fully specified form:* `(` [*fnums*] *pattern* [*onums*] `)`. The *fnums* and *onums* fields have the same meaning as described in Section 7.1, except that these are local bindings effective only for a single pattern, and they override the corresponding global bindings. Example: `(+ tuple *)`.

Internally, all patterns are mapped into fully specified forms by inserting global formula and occurrence number specifications where needed.

A string pattern also may take the "anchored" form `"^<pattern>$"`, where the meaning is similar to corresponding notation in regular expressions. In this case, anchoring is relative to formulas. Thus, `"^% = %$"` matches formulas where the top-level operator is "="; it will not match interior occurrences of "=". Where they apply, anchored searches are both faster and more precise. Currently, one-sided anchoring (`"^..."` or `"...$"`) is not supported; if used, such patterns will invoke two-sided, fully anchored matching.

In addition to repetition using `&`k, another form of pattern repetition is provided. If pattern specifications P_i and P_{i+1} have identical expression and formula number fields, and P_{i+1} does not contain an occurrence number field, then P_{i+1} will acquire an occurrence number that is one greater than P_i's. An example is `(match lambda lambda)`, where this is expanded into:

```
(match (* lambda 1) (* lambda 2))
```

Pattern variables represent arbitrary PVS expressions or other syntactic elements that may appear in the variable's context. Pattern variables may have several forms:

- *A nonbinding expression variable indicated by* `%`. This subpattern matches any PVS expression or syntax element. The subexpression matched will not be remembered.

- *A binding expression variable of the form* `%v`. This subpattern matches any PVS expression or syntax element, and the subexpression matched will be remembered and associated with `%v`. Any string forming a valid PVS identifier may be used in place of `v`.

Table 11: Elements of PVS expression syntax supported for matching.

Symbol[a]	Syntax Class	Example Instance(s)
number	Numeric constant	123, 456
name_	Name expression	z, airborne?, a!1, max_size, empty?[real]
infix	Infix operator application	x + y, P OR Q
unary	Unary operator application	-a, NOT P
applic	Function application	f(x, y), z/2
lambda	Lambda expression	LAMBDA (n: nat): 2 * n
forall	Universal quantification	FORALL x: P(x)
exists	Existential quantification	EXISTS n: odd?(n)
quant	Arbitrary quantification	FORALL i: P(i), EXISTS j: Q(j)
bind	Binding expression	LAMBDA x,y: x * y, EXISTS j: Q(j)
if_	IF expression	IF P(x) THEN x ELSE 1 ENDIF
cond	COND expression	COND x > 0 -> 1,..., ELSE -> 0 ENDCOND
proj	Projection application	t`1, (a,b)`2
field_	Field application	r`f, state`time
tuple	Tuple expression	(1, 2, 3), (a, b)
record_	Record expression	(# time := 0, day := 1 #)
set	Set expression	{ x | 0 < x AND x < 8 }
null	Empty list	(: :)
list	List expression	(: a, b, c :)
let_	LET expression	LET a = sqrt(b), c = cos(d) IN a * c
where	WHERE expression	a * a WHERE a = sqrt(b)
assign	Assignment	time := 0, day := 1
update	Override expression	f WITH [(0) := 9]
select	Selection	null: 0, cons(a,b): len(b)+1
cases	CASES expression	CASES s OF null: 0,... ENDCASES
coerce	Coercion	n :: posnat, x/2 :: fraction
bracket	Bracket expression	[| 1, 2, 3 |]
paren_vbar	Parenthesis-vbar expression	(| 1, 2, 3 |)
brace_vbar	Brace-vbar expression	{| 1, 2, 3 |}
string	String expression	"terminal area"

[a]Trailing underscores are used to distinguish certain classes from identically named prover rules.

Table 12: Elements of type-expression syntax supported for matching.

Symbol	Syntax Class	Example Instance(s)
type_name	Type name expression	nat, set[real], functions[int, real]
subtype	Subtype expression	{x | x > 0}, {n: nat | n < 6}, (prime?)
type_applic	Type application	below(8), upto(limit)
funtype	Function type	[real -> real], [nat, int -> time]
tupletype	Tuple type	[nat, nat], [nat, int, real]
recordtype	Record type	[# time: real, day: int #]

- *A nonbinding expression-list variable indicated by* `%%`. This is similar to `%` except that it matches a nonempty, comma-separated list of PVS expressions such as the argument list in a function application. Not all syntactic lists can be matched in this way because `%%` can only appear where an identifier is allowed by the PVS grammar.

- *A binding expression-list variable indicated by* `%%v`. This is the binding variable form of the previous case.

- *A nonbinding expression variable with syntax-class restriction indicated by* `%{class}`. Eligible names for *class* are the symbols from Tables 11 and 12. Any expression of the requested type will be matched.

- *A binding expression variable with syntax-class restriction indicated by* `%v{class}`. This is the binding variable form of the previous case.

Binding of pattern variables causes an association list of variable names and expressions to be collected. These are used for display purposes and to support template substitutions as described in the next section. Moreover, the full subexpressions matched by patterns are also collected and associated with names `%1, %2,...`, which likewise are used for display and substitution.

One further consideration concerns the treatment of binding variables that are repeated within a pattern or list of patterns. At times one would like multiple instances of the same variable to mean that they can only match identical subexpressions. At other times, such as when `match` is used for information display and multiple instances occur implicitly, one would prefer the matching to be unconstrained and allow different subexpressions to match variables of the same name. Implicit binding-variable duplication happens when multiple match occurrences are requested, as in `(match "sqrt(%x)" *)`.

Manip uses a convention based on variable names to allow both types of binding to be specified. For pattern variables that begin with an uppercase letter, all matching subexpressions for that variable are constrained to be "type-check equal." For all other pattern variables, subexpression matching is unconstrained by the value of the expression. Here are several examples to illustrate patterns and what they could match:

```
"%a * %a"            matches  x!1 * x!1, x!1 * y!1, 4 * x!1
"%A * %A"            matches  x!1 * x!1
"sin(%a)" "cos(%a)"  matches  sin(2*x!1), cos(2*x!1); sin(y!1), cos(z!1)
"sin(%A)" "cos(%A)"  matches  sin(2*x!1), cos(2*x!1)
```

7.5 Template Substitution

Having collected pattern variable bindings in the search phase, the `match` strategy applies this binding information to perform substitutions into the template expressions $T_1, \ldots, T_k$ as well as the proof steps $S_1, \ldots, S_m$. Template expressions need only include template variables for this to occur. Templates also may be devoid of such substitutable variables. Proof steps may lack them as well, where it is more common not to use them.

Templates are written in one of the following forms:

- *A string optionally containing* `%`*-variables.* The string is typically a PVS expression with template variables inserted where substitutions are desired. Template variables refer to

the string values of pattern variables with corresponding names, e.g., `%1`,...,`%9` and `%a`, `%v`, etc.

- *A symbol denoting a PVS expression or formula number.* These symbols begin with the $ character and are the same as those described in Section 6.1 (see Table 9). For example, to refer to the string form of the PVS expression for the first pattern matching instance, use symbol `$1s`. To refer to its formula number, use `$1n`.

- *A parenthesized s-expression.* This expression form is used for creating proof steps. Embedded within the s-expression may be both $-symbols and %-variable substrings. Substitutions for these embedded parameters will be performed.

Because simple textual substitutions are performed on the templates, sufficiently distinct variable names should be chosen so that one is not a prefix of any other. If, for example, both `%a` and `%ab` are used, the substitution for `%a` could alter `%ab` in the template before its substitution occurs.

If a string is used where a proof step is required, the string will be "read" as a Lisp s-expression (i.e., parsed but not evaluated). Including embedded substrings requires escaping the double-quote characters. For example, the string `"(case \"x!1 = y!1\")"` becomes the s-expression `(case "x!1 = y!1")` after being "read" by the Lisp interpreter.

7.6 Use Cases

Several examples of typical use cases are presented below.

- `(match lambda *)`
  Find and display all lambda expressions in the sequent.

- `(match + "sin(%)" "% * %" *)`
  Find and display all expressions of the form $\sin(a)$ and all product terms found in the consequent formulas.

- `(match - "[%%d -> %r]" *)`
  Find and display all function type expressions in the antecedent formulas, showing range types and lists of domain types.

- `(match ? lambda &1 rep)`
  Show the first two lambda expressions and the rule generated for replacing the first by the second.

- `(match lambda &1 rep ! (bash))`
  Same as the previous case except actually carry out the step and apply `bash` afterwards.

- `(match "4*%a" rep "2*(2*%a)" ! 9)`
  Create a rewrite rule on-the-fly that replaces the first expression of the form $4a$ by $2(2a)$. Try proving the rule afterwards with `grind`.

- `(match -2 "sin(%a)" < "%a/2" ! 0 4)`
  Find the argument to the first application of `sin` in formula -2 and introduce a case split on $\sin(a) < a/2$. Use `skip` as the follow-up step for the main branch and `assert` for the else branch.

- `(match (-2 "sin(%a)") (-3 "%b/2") case "%1 < %2" "%1 = %2")`
  Find $\sin(a)$ in formula -2 and $b/2$ in formula -3 and split on the cases $\sin(a) < b/2$ and $\sin(a) = b/2$.

- `(match - "sqrt(%)" (1 2) case "%1 < %2")`
  Find the first two applications of `sqrt` in the antecedents and split on the case $\sqrt{x} < \sqrt{y}$.

- `(match - "f(%a{infix})" * typepred)`
  Apply `typepred` to every expression $f(a)$ found in the antecedent formulas, where each argument a has the form of an infix function application.

- `(match - "%f(tan(%x),tan(%y))" 2 step (mult-by -1 "%f(%x,%y)"))`
  Find the second occurrence of an expression of the form $f(\tan(x), \tan(y))$, then invoke the `mult-by` step using the factor $f(x, y)$.

7.7 Extended Expression Forms

The extended expression notation has been expanded in version 1.2 to accommodate syntax matching features. These features can be used in Manip strategies wherever extended expressions were allowed before. The general form is

`(~` [*fnums*] $P_1 \ldots P_n$ [*onums*] [`->` $T_1 \ldots T_k$] `)`

where the various items have the same meaning as for the `match` strategy. The primary difference is that an extended expression `(~ ...)` evaluates to a list of PVS expressions that will be used as needed by the context, typically another Manip strategy or one from a different package altogether.

Several variants are possible, yielding different expression results.

- `(~` [*fnums*] $P_1 \ldots P_n$ [*onums*] `)`
  In the simplest variant, only patterns are present. The effect is to conduct matching and collect a list of expressions that match patterns in their entirety (`%1`, `%2`, ...).

- `(~` [*fnums*] $P_1 \ldots P_n$ [*onums*] `)`
  In the next variant, the form is the same but the patterns contain binding pattern variables (e.g., `%a`). In this case, the expressions returned correspond to what was matched by the binding variables (`%a`, `%b`, ...).

- `(~` [*fnums*] $P_1 \ldots P_n$ [*onums*] `->` $T_1 \ldots T_k$ `)`
  Finally, the most general form overrides these conventions when explicit templates are provided. What it returns is the list `(`$E_1 \ldots E_k$`)` that results from substituting in the templates `(`$T_1 \ldots T_k$`)`.

Here are a few examples to illustrate possible uses:

- `(mult-by -2 (~ "cos(% / %)"))`

- `(factor! (~ "% + 2 * %"))`

- `(div-by (~ "^% > 1/2$") (~ "cos(%a / %b)" -> "sqrt(%a) * %b"))`

In the last example, `(~ "^% > 1/2$")` matches a full formula and makes its formula number available to the strategy `div-by`.

Expressions having form `(~` [*fnums*] P_1 ... P_n [*onums*] `)` may be used wherever location references (i.e., expressions with form `(! ...)`) are allowed, such as arguments to `factor!` and similar strategies. On the other hand, expressions of the form

`(~` [*fnums*] P_1 ... P_n [*onums*] `->` T_1 ... T_k `)`

cannot serve this purpose if the templates evaluate to strings or numbers after substitution.

7.8 Automatically Deriving Abstract Patterns

One additional feature has been included to facilitate the creation of syntax patterns. This feature allows users to select a PVS expression in the Emacs proof buffer (`*pvs*`) using a mouse dragging gesture, after which Manip will find a suitable syntax pattern for matching the expression. Patterns are synthesized so as to make them fairly abstract and eliminate much of the subexpression detail found in the sequent. In so doing, we hope to create patterns that will be robust in the face of specification changes, thereby maximizing the patterns' successful use during reproof attempts.

In practical terms, here is the procedure for using this feature:

- A user begins by moving the mouse cursor in the `*pvs*` Emacs buffer to the beginning of the desired expression. Typically this point lies somewhere in the sequent display just above the current command line (i.e., near the bottom of `*pvs*`).

- Next the user presses the left mouse button, drags it to the end of the desired expression, then releases it. Typical Emacs behavior is to add background highlighting or shading to the text region selected. (Instead of dragging, any Emacs operations that set *point* and *mark* can be used to define the region.)

- Now, without moving the mouse cusor from the region (or without changing *point* and *mark*), the user types a TAB-key sequence to invoke the preferred pattern generation procedure. TAB-`%` chooses a plain syntax pattern. TAB-`~` chooses a pattern wrapped as an extended expression, i.e., TAB-`~` gives the form `(~ <pattern>)`.

- At this point Manip does some searching and after a short delay deposits a string at the end of the `*pvs*` buffer containing the generated pattern. Presumably this is in the midst of a partially typed `match` strategy invocation (for the TAB-`%` form), or another rule/strategy (for the TAB-`~` form).

- The pattern is merely text in an Emacs buffer so the user is free to embellish it, move it, delete it, etc., before invoking the proof command under construction.

- If the chosen text region is not a valid PVS expression of the current sequent, an error message is displayed in the status line at the botton of the Emacs window. For a valid expression, the region's background highlighting is removed and its text color is changed to red.

- Often the right end of an expression is difficult to locate because it lies within a sequence of right parentheses. Manip compensates by allowing the user to end the selected region close

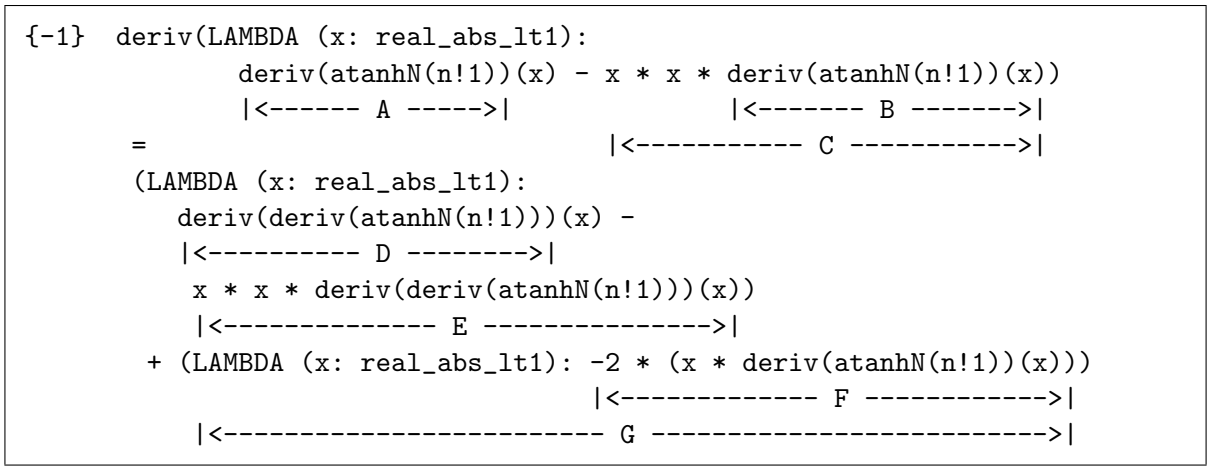

Figure 3: Example with highlighted regions for creating abstract patterns.

to the actual endpoint but not necessarily on the exact character. Nearby delimiters, white space and letters within an identifier will be examined and skipped over as appropriate to find the actual endpoint of a syntactically valid expression.

- If the expression in the user's selected region also occurs earlier in the sequent, the generated pattern will not make this distinction. It will match the first occurrence.

Besides user convenience, a goal of this pattern-generation feature is to create patterns heuristically that continue to be useful in the future as the user's theorems and proofs evolve over time. To achieve this, we strip out excess detail, trying to retain only the high level structure of an expression. Pattern variables `%` and `%%` are introduced liberally to replace subexpressions. Although a small pattern is sought, a minimum "specificity" is applied to keep from losing robustness at the other extreme, namely, with overly abstract patterns. For example, the first formula of a sequent (-1) will always match the universal pattern `"%"`. If we used this pattern, though, the proof would fail in the future should another formula come first. We strive for reasonably generic patterns but not maximally generic ones.

Similar to the case of Table 11, all valid PVS expressions except `TABLE` can be subject to pattern generation. For small expressions and those having unsupported syntax, the "pattern" returned will be the original expression itself. Moreover, those syntax elements that do not form a complete PVS expression cannot generate patterns at all. Examples of such non-expression syntax elements are assignments from a `WITH` expression and selections within a `CASES` construct. Also, some expression types will have certain subexpressions always represented by `%` or `%%`. This approach is taken to limit the combinatorial expansion that can arise from considering too many alternatives.

To better visualize pattern generation, consider the PVS formula in Figure 3, showing several of its expressions as regions A-G. Table 13 displays the patterns that would be created by TAB-`%` invocations on these regions.

An additional TAB-key generates patterns for matching entire formulas. TAB-`^` prompts for one or more formula numbers, then inserts a form `(~ "^<pattern>$" ...)` at the end of buffer `*pvs*`. With this extended expression form, users can let patterns identify formulas, obviating the need for concrete formula numbers and reducing future efforts in proof maintenance. TAB-`^`

Table 13: Abstract patterns derived for text regions in Figure 3.

Region	Derived Syntax Pattern	Notes
A	`"deriv(atanhN(%))"`	`"deriv(%)"` is considered too generic
B	`"deriv(%)(%)"`	Two applications make it specific enough
C	`"% * deriv(%)(%)"`	First `%` matches `x * x`
D	`"deriv(deriv(%))"`	Two applications again
E	`"% * deriv(deriv(%))(%)"`	Extra detail needed to distinguish from C
F	`"% * (% * %)"`	`*` is normally left-associative
G	`"LAMBDA (%): % * %"`	`%%` would be emitted for multiple bindings

can be combined with other Manip features to achieve this effect. For instance,

```
(invoke (hide (~ "^<pattern>$")))
```

will hide the first formula matching `<pattern>`. (Strategies in Manip and Field do not need `invoke` to achieve this effect because they process extended expressions automatically. For built-in prover rules, though, `invoke` would be needed.)

Finally, several Lisp variables are available to fine tune the behavior of the pattern generator. Variables can be changed within the prover using this idiom:

```
(lisp (setf <variable> <value>))
```

Normal usage should not require alteration; these variables are provided in case unusual situations arise.

A sketch of the generation algorithm will help in understanding how to use these values. Given a highlighted expression E, Manip performs a traversal of E's syntax tree to identify its subexpressions. This process is limited to a maximum depth. At each stage, Manip collects alternative subexpressions that can be represented by various combinations of syntax variables (`%`, `%%`) and actual terms. After interleaving these alternatives, a list of possible patterns results. A weight or "specificity" value is computed for each alternative pattern to indicate roughly how much of E's tree it contains. The list is then sorted by ascending specificity values. Those having a specificity below the minimum are discarded, then the next N are chosen as final candidates. These will be tried in order until the first one that successfully matches E is found.

`*max-gen-pattern-depth*` [*default:* 6] [*Variable*]

The maximum depth of tree traversal in the generation algorithm is capped by this value.

`*max-gen-pattern-try*` [*default:* 12] [*Variable*]

Match attempts during the search process are limited to at most `*max-gen-pattern-try*` candidate patterns.

`*min-pattern-specificity*` [*default:* 3] [*Variable*]

This variable stores the minimum specificity threshold for candidate patterns.

7.9 Additional Considerations

It is important to be mindful of a few limitations when using the syntax matching features.

- Syntax matching and textual substitution can take PVS subexpressions out of context, possibly inserting them into new contexts where they are invalid. For example, a subexpression containing bound variables will most likely produce semantic errors when submitted back to the prover.
- Although type expressions are supported, they are not always recognized because PVS syntax does not distinguish them in all cases. For instance, `(prime?)` and `below(8)` appearing as patterns by themselves are assumed to be value expressions rather than type expressions.
- Because of the recursive representation used by PVS for some syntax classes, a search for instances of such a class might return interior objects as well as top-level objects. Results could differ from what one would expect based on the printed form of PVS expressions.
- Objects for some syntax classes have implicit ELSE conditions that do not appear in the printed representation. Searches can match subexpressions of such implicit conditions.
- Using `%%` to match expression lists is not always possible because this feature is not fully compatible with the PVS grammar. For example, both `(%%)` for tuples and `(# %% #)` for records will fail, although for different reasons. The former will be parsed as an ordinary parenthesized expression while the latter is disallowed by the grammar.
- Large formulas having many similar subexpressions can lead to noticeable search times when performing matching. A more efficient search might be obtained by including context terms that occur infrequently.

8 Library Extension Framework

Some mathematical domains such as vectors of reals have operations that share many algebraic properties with the reals. Given that it would be highly desirable to use Manip strategies on expressions of such types, an extension framework was added to Manip in version 1.3 to support this kind of generalization.

The basic idea is to introduce an object-orientation technique for strategies. Within the Lisp code for a strategy, different methods can be invoked according to the types of the PVS objects involved. This allows a uniform interface for strategies, where each one performs a function based on the conventional algebraic properties for reals. For types that share (some of) those properties, the same strategy can be invoked in the same manner by the user. This simplifies the proof process by reducing the need for users to apply type-specific lemmas.

A division of labor has been introduced in support of this architecture, where the Manip core can be extended by functionality that resides in PVS libraries, such as the vectors library that is distributed with the NASA PVS library suite. The core Manip strategies perform the basic operations on reals as they always have. Some of these strategies have been generalized to allow their use with types other than real. In such cases they also provide the front-end processing for steps where they are operating on non-real types. The back-end processing that is more

type-specific will be contributed by extensions from the relevant library. These extensions take the form of Lisp code residing in the library's directory. A file named "`pvs-strategies`" in this directory will be loaded when the first library-related proof is started, which in turn will load additional Lisp files as needed.

CLOS classes are used to dispatch methods according to operand types. The methods for reals are built in to the Manip core. Those for library-specific types such as vectors are declared within Lisp files in the library's directory. These files will register the library extensions available to Manip users. This happens when the extension files are loaded, which occurs when the first proof using the library is started.

8.1 Extension Example: Vectors

The extensions for vector types as implemented in the NASA vectors library are summarized in Table 14. When this library is included in the user's context, a summary can be obtained by invoking this command:

```
(help manip-vectors)
```

Shown in the table are existing Manip strategies and how they can be used to operate on vectors or combinations of reals and vectors. Descriptions of the extended strategies and their operation on reals are presented in Section 4.

8.2 New Strategies

To complement the extension of existing Manip strategies, several new strategies were introduced whose actions are not normally needed for reals, but will work for reals as well as other types such as vectors. They were introduced to help make up for processing normally carried out by decision procedures in a fully automatic fashion.

`permute-terms` *fnum side* `&optional` (*term-nums* `1`) (*end* `R`) [*Strategy*]
`permute-terms!` *expr-loc* `&optional` (*term-nums* `1`) (*end* `R`) [*Strategy*]

Occasionally it is desirable to reorder the additive terms within an expression to facilitate future manipulations or the application of rewrite rules. To perform this task, `permute-terms` allows the specification of term numbers for gathering the selected terms and placing them at either *end* of the new expression. Terms in a sum are numbered left-to-right starting with number 1. Parentheses are ignored for the purpose of numbering terms.

For *end* = `L`, the action of `permute-terms` is as follows. Let the expression on *side* of formula *fnum* be a sum of terms, $S = t_1 \pm \ldots \pm t_n$. Identify a list of indices I (*term-nums*) drawn from $\{1, \ldots, n\}$. Construct the sum $t_{i_1} \pm \ldots \pm t_{i_l}$ where $i_k \in I$. Construct the sum $t_{j_1} \pm \ldots \pm t_{j_m}$ where $j_k \in \{1, \ldots, n\} - I$. Then rewrite the original sum S to the new sum $t_{i_1} \pm \ldots \pm t_{i_l} \pm t_{j_1} \pm \ldots \pm t_{j_m}$. Thus the new sum is a permutation of the original set of terms with the selected terms brought to the left in the order requested. For *end* = `R`, the selected terms are placed on the right.

In the `!`-variant, the *expr-loc* argument supplies a location reference to identify the target expression(s). Multiple expression locations may result from a single *expr-loc* argument. Each will be processed separately.

Table 14: Vector-aware strategies (for Manip 1.3).

Strategy name	Vector-specific operations
`swap`	Commutes vector sums and dot products.
`group`	Associates vector sums and scalar/dot products.
`swap-group`	Combination of these two actions.
`mult-by`	Create scalar products or dot products according to types in formula and new term.
`div-by`	Dividing by real x creates scalar product $(1/x) * v$; dividing by vector u creates scalar product $(1/\text{norm}(u)) * v$.
`move-terms`	Behaves with vector sums exactly as for reals.
`cross-mult`	Acts on divisors in the real part of scalar products.
`factor`	Factors the real and vector parts of sums of scalar products.
`isolate` `isolate-replace` `cross-add`	These strategies work for vectors because they invoke vector-aware strategies internally (specifically, `move-terms`).
`permute-mult` `name-mult`	These strategies work the same on scalar products as for real products, although their actions are limited to the real parts of scalar products.
`permute-terms` `elim-unary` `cancel-add` `distrib`	These strategies are new in version 1.3. See Section 8.2 for a description.

Usage: `(permute-terms 3 L (4 2))` rearranges the sum on the left side of formula 3 to be `t4 + t2 + t1 - t3`, with the default association rules making it internally represented as `((t4 + t2) + t1) - t3`.

`cancel-add &optional` (*fnums* `*`) [*Strategy*]
`cancel-add!` *expr-loc* [*Strategy*]

Cancellation of additive terms for reals is normally unnecessary. The decision procedures can handle such cases quite well. In other domains, however, cancellation is not so automatic. To deal with these cases in supported libraries, `cancel-add` provides a capability similar to what is available for reals.

Cancellation is possible when *fnum* has one of two forms:

$$(1)\ \ldots + x + \ldots - x + \ldots \ = \ z, \qquad (2)\ \ldots + x + \ldots \ = \ \ldots + x + \ldots$$

In other words, it can cancel terms that appear on both sides of a relation (provided they have the same sign), and it can cancel terms having opposite signs on the same side of a relation. This latter case also applies when the `cancel-add!` form is used. The positions of canceling

terms within their expressions is irrelevant; `cancel-add` will locate them and apply deductions as needed. Note that equality might be the only relation that can work for a given domain.

Usage: `(cancel-add 2)` tries to cancel additive terms from both sides of formula 2.

`elim-unary` *fnum* `&optional` (*side* `*`) [*Strategy*]
`elim-unary!` *expr-loc* [*Strategy*]

Expressions of additive terms sometimes contain terms having the unary minus operator. For some domains, and even for reals, the normal algebraic simplification one would expect does not always take place. For example, $x + -y$ might not simplify to $x - y$. This specialized type of reduction is performed by `elim-unary`. It forms a new expression by collecting all positive terms and placing them on the left, followed by all negative terms placed on the right. All unary minus operators are eliminated in favor of binary minus. Only in the case where all terms are negative will there remain any unary negations.

Usage: `(elim-unary 2)` eliminates unary minus operators from both sides of formula 2.

`distrib` *fnum* `&optional` (*side* `*`) (*term-nums* `*`) [*Strategy*]
`distrib!` *expr-loc* [*Strategy*]

The inverse operation of `factor` is `distrib`, which distributes multiplication over factors having the form of additive subexpressions. For reals, this action is performed automatically during core proof commands such as `assert`. For other domains, this action might need to be invoked explicitly, which is the reason for `distrib`. The term numbers refer to top-level additive terms in the formula or expression where distribution is desired. The other terms in the formula will remain intact.

Usage: If terms 1 and 3 on the right side of formula 2 have the form $x * (a + b)$ and $y * (c - d)$, `(distrib 2 R (1 3))` distributes multiplication in those terms to form $x * a + x * b$ and $y * c - y * d$.

9 Support Functions

Several Common Lisp functions defined in this package might be of use to strategy writers. All Lisp objects are available to any strategy layer built on top of Manip. The functions below are used to access PVS data structures and perform other frequently needed chores. In general they return `nil` when called with ill-formed arguments such as invalid formula numbers.

`get-equalities` [*Function*]

`get-equalities` returns a list of formula numbers for all antecedent equalities found in the current sequent.

`get-relations` *fnums* [*Function*]

Collect the formula numbers for all the relational formulas in the current sequent, omitting the case of the `/=` operator.

`map-fnums-arg` *fnums* [*Function*]

Use `map-fnums-arg` to map *fnums* into a list of concrete formula numbers, converting the symbols +, −, ∗ and formula labels as needed.

`extract-fnums-arg` *fnums* [*Function*]

This utility extracts a list of formula numbers from an input that could include either extended expressions or conventional formula numbers.

`map-term-nums-arg` *tnums* [*Function*]

Use `map-term-nums-arg` to map *tnums* into a list of concrete term numbers, converting the symbol ∗ and special form `(^ ...)` as needed.

`manip-get-formula` *fnum* [*Function*]

`manip-get-formula` retrieves from the current goal the PVS data object corresponding to the formula specified in *fnum*. For an antecedent formula, the unnegated form is returned. The object returned is a CLOS object instance belonging to whatever class corresponds to the top-level PVS expression.

`percent-subst` *pattern values* [*Function*]

Textual substitution of template variables $\%1, \ldots, \%n$, as discussed in Section 6.1, is performed by `percent-subst` using the list of values provided. Ideally, the number of elements in list *values* should equal n, the number of template variables.

`percent-to-regexp-pattern` *pattern* [*Function*]

This function maps a pattern written in the pattern language, i.e., strings involving text field designators, into a regular expression suitable for matching and collecting substrings. The resulting regular expression may be passed to the Common Lisp function `match-regexp` to carry out pattern matching and obtain a multiple-value outcome.

`eval-ext-expr` *expr-spec* [*Function*]

Extended expression specifications are evaluated by this function. It returns a list of expression descriptors, each of which is a structure containing the values `<expr string>, <fnum>` and `<CLOS object>`. Use the access functions `ee-string, ee-fnum` and `ee-pvs-obj` to retrieve these components from a descriptor. Some descriptors will not have meaningful values for each component; the value `nil` is supplied in such cases. Also, the function returns `nil` if the input *expr-spec* is ill-formed.

`build-instan-cmd` *cmd descriptors* [*Function*]

An instantiated command is constructed by this utility function. Substitutions for all special symbols are performed and a fully instantiated command is returned as the function's value.

Table 15: Package file summary.

File name	Purpose
`manip-guide.pdf`	User's manual (this document)
`pvs-prover-manip.el`	User interface functions and utilities
`manip-strategies.lisp`	Prover strategies for manipulation
`extended-expr.lisp`	Extended expression functions for Manip
`syntax-matching.lisp`	Functions for syntax matching feature
`manip-utilities.lisp`	Various support functions for package
`pregexp.lisp`	Generic regular expression package developed by Dorai Sitaram
`debug-utils.lisp`	Debugging utilities for strategy writers
`debug-utils.el`	Optional Emacs support for utilities

`try-justification` *name try-just* [*Function*]

Generate a step using `TRY` that tries to prove a justification branch using *try-just* and backtracks on failure.

`*suppress-manip-messages*` [*Variable*]

Setting this variable to a non-`nil` value suppresses Manip's display of errors, warnings and status messages. Returning the value to `nil` restores normal message display behavior.

10 Files

There are several files included with this package. Their functions are summarized in Table 15. These files can be found in the source code distribution of PVS (version 5.0 and later).

11 Caveats

- This version of the package has been developed for and tested on PVS version 5.0 (Apr 2011). As currently configured, it will not work with earlier versions.
- Error checking in Manip should be adequate but is not complete. Some erroneous user inputs will not be cited as such, but will result in strategies having no effect. Although applying a strategy could result in a Lisp error, this should be rare. If it happens, restore the prover's state (Ctrl-D is convenient for this purpose), review the strategy's input expressions for problems, and retry using modified inputs.
- Some strategies invoke `name-replace` as an internal step using names chosen by the strategy procedures. The naming convention followed is to use identifiers ending in a double underscore, e.g., `x1__`. Refrain from using such names to avoid conflicts.

- Due to a limitation of the regular expression module being used, the pattern matching implementation for the features in Section 5.3 does not recognize the length-one case for `%d&` text fields.
- When using the higher-order strategies (e.g., `invoke` or `for-each`) with substitutions based on the full extended expression descriptors (e.g., `$1`, `$2`, etc.), the prover might reject the command. If this happens, try changing the parameterized command to its $-form. For example, instead of `(invoke (swap-rel $1) (? - "<="))`, try the nonatomic form `(invoke (swap-rel$ $1) (? - "<="))`. Changing `$1` to `$1n` is another way to avoid this problem.

12 Conclusion

It is our belief that tactic-based theorem proving holds much promise for automating domain-specific reasoning. To date, this area has received moderate attention from the formal methods and theorem proving communities. Nevertheless, much more effort has gone into developing other capabilities, such as decision procedures and rewrite rules. While these are undoubtedly valuable, there is still ample room for other advances, particularly those that can leverage the accumulated knowledge of experienced users of deduction systems. We hope this set of tools offers a practical contribution to the area of tactic-based proving.

The Manip package has been exercised successfully for over ten years by users at NASA Langley and also at a few other sites. Further development in the near future is not anticipated, although enhancements are possible if the need arises. Feedback of any kind is still encouraged. We also welcome users to build on these features when developing their own personal strategies. Future enhancements will most likely be additions so that current features should remain intact.

Acknowledgments

The need for this package and many initial ideas on its operation were inspired by Ricky Butler of NASA Langley. Additional ideas and useful suggestions have come from César Muñoz of NASA Langley, and John Rushby and Sam Owre of SRI. We appreciate their insightful input and feedback. Special thanks go to Sam Owre for his support in adding Manip to the standard PVS distribution.

Index