A PVS Prover Strategy Package for Common Manipulations
Version 0.9

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

October 31, 2001

Abstract

Low-level sequent manipulations are sometimes needed when using the PVS prover. A preliminary package of strategies and Emacs functions is provided to make such manipulations less tedious. Included are strategies aimed at real number theorem proving as well as techniques applicable in a more general setting.

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 rather limited. Fortunately, SRI continues to increase the amount of automation in PVS. In particular, decision procedures for real arithmetic are a planned future enhancement. We look forward to these 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 8 describes package installation and setup. Section 9 lists a number of caveats and limitations of the current version. We emphasize that this package is being made available on an experimental basis. Considerable redundancy exists in the feature set. After a period of user experimentation, we expect to remove those features of marginal utility and elaborate those that have been found to be useful.

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

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. 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) 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.

- 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.

- A feature of interest to both strategy writers and users is a quick way to restore the prover's state after a Lisp error is detected. TAB-] (right bracket) enters the Lisp command `(restore)` to return the prover to its previous state.

- 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.

- Many of the lemmas in prelude theory `real_props` are used by the arithmetic strategies. Additional lemmas are needed to implement certain operations. The PVS prelude-extension feature is the mechanism for integrating these new lemmas without requiring any action on the user's part. Extension theories and their lemmas are automatically visible within the prover.

  A theory called `extra_real_props` extends the prelude with a set of real number properties. These lemmas are applied as needed by the strategies, but also may be applied directly by the user if desired. As strategies are added to the package, this theory will be extended further to support new proof tactics.

- 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.

- 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 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.

- 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.

- 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 "$" forms are the nonatomic strategy forms, which can be used to improve diagnostic information by showing the expansion into core PVS rules. For instance, use `cancel$` instead of `cancel` to obtain an expanded proof.

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, 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.

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` `&opt (side L) (infix? T))`	L: $x \circ (y \circ z) \Longrightarrow (x \circ y) \circ z$ R: $(x \circ y) \circ z \Longrightarrow x \circ (y \circ z)$
`(group! expr-loc &opt (side L))`	
`(swap-group term1 operator term2 term3` `&opt (side L) (infix? T))`	L: $x \circ (y \circ z) \Longrightarrow y \circ (x \circ z)$ 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` `(sign +) (try-just nil))`	Claims term has sign indicated
`(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` `&opt (term-nums *))`	Move additive terms to other side
`(isolate fnum side term-num)`	Move all but one term
`(isolate-replace fnum side term-num` `&opt (targets *))`	Isolate then replace with equation
`(cancel &opt (fnums *) (sign nil))`	Cancel terms from both sides
`(cancel-terms &opt (fnums *) (end L)` `(sign nil) (try-just nil))`	Cancel speculatively & defer proof
`(op-ident fnum &opt` `(side L) (operation *1))`	Apply operator identity to rewrite 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
`(factor fnums &opt (side *)` `(term-nums *) (id? nil))`	Factor common multiplicative terms from additive terms given
`(factor! expr-loc &opt` `(term-nums *) (id? nil))`	
`(transform-both fnum transform` `&opt (swap nil) (try-just nil))`	Apply transform to both sides of formula

Table 2: Summary of manipulation strategies (continued).

Syntax	Function
`(permute-mult fnums &opt (side R) (term-nums 2))`	Rearrange factors in a product
`(permute-mult! expr-loc &opt (term-nums 2))`	
`(name-mult name fnum side &opt (term-nums *))`	Select factors, assign name to their product, then replace
`(name-mult! name expr-loc &opt (term-nums *))`	
`(recip-mult fnums side)`	$x/d \Longrightarrow x * (1/d)$
`(recip-mult! expr-loc)`	
`(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))`	Generate case analyses for products
`(mult-extract name fnum &opt (side *) (term-nums *))`	Extract selected terms, name replace them, then simplify
`(mult-extract! name expr-loc &opt (term-nums *))`	

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 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 - 2)" "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 12). 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 disjunctions 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. PVS 3.0 is supposed to include a way to display parenthesization, so this strategy is only a stopgap measure until 3.0 arrives.

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.

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 be able to cancel nonidentical terms when they are believed to be equal. This capability is offered through `cancel-terms`, which case splits on the assumption that two terms are equal. In the special case of a multiplicative expression, the user can specify at which *end* (L or R) the allegedly common term is to be found. Unlike `cancel`, either end of a chain of multiplications may be specified and a preparatory rearrangement will be carried out before canceling. Associative groupings are ignored when identifying the *end* term. For inequalities, the *sign* argument can be used to indicate term polarity as in `cancel`. Finally, 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`) [*Strategy*]
`permute-mult!` *expr-loc* `&optional` (*term-nums* `2`) [*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.

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.

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`) [*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.

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. 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 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. Any indices after the 0 have no effect.

- 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.

- 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.

- 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.

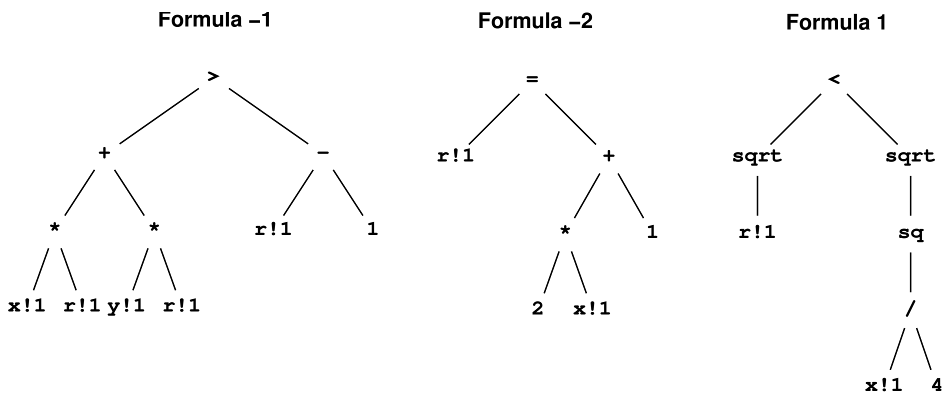
- A form `(->* g1 ... gk)` that behaves the same as `(-> ...)` except that all eligible paths are found and returned as values.

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. Also note that repeated function names in a go-to form, such as `(! 2 (->* "sq" "sq"))`, will not descend to the lower expression(s). The second occurrence of the guard `"sq"` will be immediately matched by the expression reached by the first occurrence. To reach the lower `"sq"` requires a specification such as `(! 2 (->* "sq") 1 (->* "sq"))`.

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.

Figure 1: Expression trees for formulas in Table 3.



- 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.

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 expressions. For instance, `(! * R "+")` finds all formulas whose right-hand sides have 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.

5.3 Pattern Matching

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 in terms of the built-in regular expression package bundled with PVS's Allegro Common Lisp environment. This module provides only basic regular expression features, much less sophisticated than Perl-style regular expressions. Nevertheless, it appears to be adequate for the purpose at hand, and runs faster than more elaborate matching engines.

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.

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`
`(! -2 R -*)`	`2`, `x!1`, `1`
`(! 1 R -*)`	`x!1`, `4`
`(! -2 R **)`	`2 * x!1 + 1`, `2 * x!1`, `2`, `x!1`, `1`
`(! 1 R 1 **)`	`sq(x!1 / 4)`, `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))
```

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).

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

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))
```

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

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 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. We will extend the matching later to incorporate other elements of the PVS grammar.

Bottom-up expression matching (post-order traversal) may be requested as well as top-down

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 *

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 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 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

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))
```

```
(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.

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

- 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.

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 formula numbers (no duplicates)
`$*n`	List of all formula numbers (includes duplicates)
`$1j, $2j, ...`	CLOS object for nth expression
`$*j`	List of all CLOS objects

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 is 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.

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 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 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.

6.4 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.

6.5 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.

7 Support Functions

Several Common Lisp functions defined in this package might be of use to strategy writers. They are used to access PVS data structures and perform other routine but frequently needed chores.

`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.

`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 to be passed to the Common Lisp function `excl:match-regexp` (under the Allegro implementation) 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>`. Some descriptors will not have meaningful values for each component. The value `nil` is supplied in such cases.

Table 10: Package file summary.

File name	Purpose
README	Summary of distribution
manip.ps	User's manual (this document)
pvs-prover-manip.el	Emacs Lisp functions
pvs-strategies	Top-level strategies file (sample)
manip-strategies	Prover strategies for manipulation
extend	Subdirectory to hold extra_* files
.pvscontext	Context file for extension theory
extra_real_props.pvs	Additional real number lemmas
extra_real_props.prf	Proofs of additional lemmas
rulebase-signatures	Needed by Emacs utilities; leave intact
test.pvs	Theory of test cases
test.prf	Proof steps for test theory

`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.

8 Files

There are several files included with this package. Their functions are summarized in Table 10.

To set up PVS to use the package, first unpack the `tar` file to create the subdirectory `manip-package` in a suitable place. Then create, replace or augment the `pvs-strategies` file. If this file does not already exist, adapt the sample provided in this package, or create a new one and add the following line:

```
(load "<path>/manip-strategies")
```

where `<path>` points to `manip-package` or wherever you chose to put the package files. If the file does exist, simply add this line to the end or rearrange the strategy files so that `pvs-strategies` is just a top-level shell that loads all the individual strategy files to be used. You may follow this with other file loading operations, for example, loading the Field strategies developed by César Muñoz.

It is best to place the `pvs-strategies` file in your home directory. PVS will check for it there as well as the current directory. Then place all the other package files in a separate directory or leave them in the `manip-package` directory that resulted from unpacking. An environment variable called `PVSMANIP` is now supported, which should be set to the path for the directory where the package files are stored.

The `extend` subdirectory holds the `extra_real_props` files, which are included automatically using the PVS prelude extension feature. This means the lemmas of that theory will be found automatically without having to explicitly import the theory. For this to work, the `extend` subdirectory needs to be found in the directory named by `PVSMANIP`, or in the current directory if this environment variable is not set. It might happen that the context file included in `extend` fails to show all of the lemmas in `extra_real_props` as being proved. If this occurs, the theory may be reproved from the `.prf` file found in `extend`.

Finally, set up the Emacs environment. If you are already using a `.pvsemacs` file, add a line to the end:

```
(load-file "<path>/pvs-prover-manip.el")
```

making sure to replace `<path>` by the directory path needed to access the Emacs Lisp file `pvs-prover-manip.el` (usually the same as the path above). If you have no `.pvsemacs` file, you are urged to create one and add the line above. Otherwise, you can always manually load `pvs-prover-manip.el` within PVS using the command `M-x load-file`.

A small theory used for regression testing is also included with the package to illustrate the various features. Formulas found in the theory are not theorems and their "proofs" will not lead anywhere. They are provided as a means to exercise assorted strategies and easily view the results. Stepping through the proofs will give the user a demonstration of the major capabilities of the manipulation package.

9 Caveats

- This version of the package has been developed for and tested on PVS version 2.3 (patch level 1.2.2.36).

- Because the package is still experimental, error checking is somewhat limited. User inputs are not thoroughly checked, so applying a strategy could result in a Lisp error. If this happens, restore the prover's state (TAB-`]` is convenient for this purpose), review the strategy's input expressions for problems, and retry using modified inputs. Over time, refinements of the package will improve error checking.

- Several strategies make use of additional real number lemmas provided in prelude extension theory `extra_real_props`. As described in the previous section, these lemmas should be found automatically by PVS. If they are not found, check to see whether the file setup is correct.

- There are cases in which `extra_real_props` will not be found that appears to be due to a PVS bug. (It has been reported.) In particular, if your theory has no importings whatsoever, the prelude extension feature seems to fail. To get around this problem, make your theories import something, which is usually the case in practice. You can always add a line such as

  ```
  IMPORTING extra_real_props
  ```

  which should take care of the problem without changing the semantics of your theories.

- Despite efforts to make the package work under both Emacs and XEmacs, there may be residual problems getting the package to run under GNU Emacs. If this happens, the best approach is to run PVS under XEmacs when using this package. If GNU Emacs is used and problems occur with `extra_real_props`, try issuing the command `M-x load-prelude-library` to force PVS to load the theory.

- Due to a limitation of the regular expression module being used, the pattern matching implementation does not recognize the length-one case for `%d&` text fields. How to remedy this is still under study.

- 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.

- 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.

10 Conclusion

It is our belief that tactic-based theorem proving holds much promise for automating domain-specific reasoning. To date, this area has received only modest attention from the formal methods community. Much more effort has gone into developing decision procedures and rewrite rule capabilities, for example. 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 new set of tools offers a practical contribution to the area of tactic-based proving.

Be advised that this manipulation package is still under development. Further experimentation is needed to refine it and make it more useful to a broad array of PVS users. Feedback of any kind is encouraged so that it might be improved. We hope that this package can serve as a starting point that can be easily extended and enhanced by the user community as the need arises. We also welcome users to build on these features when developing their own personal strategies, bearing in mind that future modifications are likely.

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 ICASE. We appreciate their insightful input and feedback.

Index