**Presented at STRATA 2003, Rome, Italy, September 8, 2003**

# Developing User Strategies in PVS: A Tutorial

Myla Archer[1], Ben Di Vito[2], and César Muñoz[3*]

[1] Naval Research Laboratory, Washington, DC 20375, USA
archer@itd.nrl.navy.mil
[2] NASA Langley Research Center, Hampton, VA 23681, USA
b.l.divito@larc.nasa.gov
[3] National Institute of Aerospace, Hampton, VA 23666, USA
munoz@nianet.org
http://research.nianet.org/~munoz

**Abstract.** This tutorial provides an overview of the PVS strategy language, and explains how to define new PVS strategies and load them into PVS, and how to create a strategy package. It then discusses several useful techniques that can be used in developing user strategies, and provides examples that illustrate many of these techniques.

## 1 Introduction

Why use strategies in PVS? There are several compelling reasons for doing so. We offer a few scenarios below that illustrate productive uses for strategies.

PVS provides a core set of inference rules supplemented by decision procedures and other simplification heuristics. Continuing enhancements to the theorem prover gradually increase the automation available to interactive users. Nevertheless, the level of automation perceived by users is still much lower than desired. This is not a problem peculiar to PVS; similar provers suffer the same limitations. In fact, PVS is among the most automatic of provers in its class.

Strategies provide an accessible means of increasing the automation available to users of the PVS prover. This can be done in generic form, suitable for a wide range of proving tasks, or in specific problem domains, yielding specialized tools suitable only in narrow contexts. Development of strategies can be performed by end users or specialists whose role is to create strategies for use by others. Over time, strategy development can lead to a reusable body of "deductive middleware." An effective division of labor in the overall conduct of mechanical theorem proving is a possible outcome of this process.

In the following, we provide several examples of strategies that are likely to be beneficial to PVS users.

- *Modest strategies to streamline prover use.* This is the simplest category of strategies, typically involving rules with just a few lines of definition. An example would be introducing rules to invoke frequently occurring sequences of proof commands. Consider the sequence `(LIFT-IF)`, `(SPLIT)`, and `(ASSERT)`. One could introduce a strategy named `IF-SPLIT` to carry out this sequence. Such strategies are easy to create, although their benefit is limited to saving the effort of repetitive typing.
- *Extended forms of predefined rules.* A slightly more advanced approach is to identify commonly needed inferences that are guided by user input. By writing strategies that accept arguments, it is possible to create enhanced versions or combinations of rules that already exist in the predefined set provided by PVS. In fact, many of the higher level predefined rules were created using the strategy mechanism. Consider, for example, a rule to claim that the lefthand sides of two formulae are equal, then invoke the appropriate `CASE` command. We might apply such a strategy using `(CLAIM-EQ -1 -3)` where `CLAIM-EQ` is the new proof rule and `-1` and `-3` are the numbers of the sequent formulae to be considered.
- *Algebraic manipulation and arithmetic simplification.* The PVS decision procedures handle linear arithmetic well, but have more difficulty with nonlinear expressions. In such cases, users must apply lemmas from the prelude or other sources. Strategies can be effective at manipulating arithmetic expressions when guided by user input. The package Manip [5], for instance, provides strategies for conducting user-directed manipulations of real-valued expressions. Similarly, the package Field [6] carries out higher level arithmetic reduction with considerable automation.

[*] Funded under NASA Cooperative Agreement NCC-1-02043.

# Report Documentation Page

*Form Approved*
*OMB No. 0704-0188*

1. REPORT DATE
**SEP 2003**

2. REPORT TYPE

3. DATES COVERED
**00-00-2003 to 00-00-2003**

4. TITLE AND SUBTITLE
**Developing User Strategies in PVS: A Tutorial**

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
**Naval Research Laboratory,4555 Overlook Avenue,**
**SW,Washington,DC,20375**

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
**Approved for public release; distribution unlimited**

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:

| a. REPORT | b. ABSTRACT | c. THIS PAGE |
|---|---|---|
| **unclassified** | **unclassified** | **unclassified** |

17. LIMITATION OF ABSTRACT

18. NUMBER OF PAGES
**27**

19a. NAME OF RESPONSIBLE PERSON

- *Deduction support for specialized models or specifications.* Verification or analysis tasks based on theorem proving often take place in the context of a specialized model of computation, such as state machines, hybrid automata, etc. Proofs in such contexts often have a stylized character that lends itself to automated proof. By capturing the proof steps and decision processes in the form of strategies, it is possible to provide a great deal of targeted automation to the proof effort. TAME is an example of such an approach within the domain of timed automata.
- *Interfaces to external proof support tools.* Occasionally it is desirable to make use of additional tools that support the prover in the construction of large or difficult proofs. Strategies in this role can be used as a means of accessing the current proof state and exporting information to an external tool. After computing its result, the external tool can supply information to be acted on in some way, such as submitting prover commands. An example would be a tool that performs database searches, then returns the names of suitable lemmas for possible invocation. PVS's `musimp`, `model-check`, and `abstract-and-model-check` strategies are also examples of this approach.
- *Interfaces to support external components through proving.* The support relationship can work in the other direction as well. Under some arrangements, the prover can be used to provide support to an external process. For example, a computer algebra system might wish to consult a theorem prover to confirm that a transformation it needs to perform is valid under certain conditions. This request could be posed as a set of conjectures sent to the prover, where a strategy-guided proof process would attempt to settle the question and return a result.

These suggested uses of PVS are by no means exhaustive. They are realistic, however. Each of these uses has either been implemented or is currently under development. No doubt other applications will be discovered. It is our hope that this tutorial might lead others to investigate new possibilities.

The remainder of this tutorial is organized as follows. Section 2 provides the basic information needed for defining your own strategies and making them available in PVS. Section 3 describes and illustrates a set of techniques that can be used in the development of user strategies. Section 4 provides examples that demonstrate how to use various techniques to develop both strategies that facilitate user interaction with PVS and automatic strategies. Finally, Section 5 discusses some additional support that would be useful in to developers of PVS user strategies.

## 2 The basics

### 2.1 PVS commands.

PVS commands can be either *rules* or *strategies.* A *rule* is a command that can be invoked by name and (if appropriate) applied to arguments. Rules execute as atomic steps in the PVS prover. A *strategy* is a command created by using zero or more PVS strategy-building commands to combine rule applications and other strategies. Thus, every rule application is also a (degenerate) strategy. Executing a strategy in the PVS prover causes execution of the sequence of atomic steps needed by the strategy for the current subgoal. On the syntactic level, the heart of a strategy definition is a *strategy expression* built by using strategy-building command names to combine rule names (applied to arguments, which may involve variable names) and other strategy expressions.

A representative set of PVS strategy-building commands is listed in Table 1. For short, we will refer to these commands as *strategicals*, in analogy to the *tacticals* in Coq, HOL, and other theorem provers that are used to combine simpler tactics into more complex ones.

A simple example strategy that is sometimes useful is:

```
(THEN (LIFT-IF) (PROP) (ASSERT) (FAIL))                    (1)
```

Strategy (1) is useful in determining whether straightforward simplification combined with the PVS decision procedures will achieve a goal; if it does not, then the intended behavior of this strategy is to return to the proof subgoal in which it is invoked, without generating any new subgoals. Most simple sequential strategies do not use `(FAIL)`; because it does so, Strategy (1) can behave badly. In particular, it causes full or partial proof failure if none of `(LIFT-IF)`, `(PROP)`, and `(ASSERT)` has an effect. One way to ensure the intended

| Strategical | Description |
|---|---|
| `(APPLY` *step*`)` | Turns *step* into a defined rule. |
| `(THEN` $step_1$ `...` $step_n$`)` | Applies $step_1$ to $step_n$ in order down all branches. |
| `(THEN@` $step_1$ `...` $step_n$`)` | Applies $step_1$ to $step_n$ in order down the main proof branch. |
| `(IF` *lisp-expr* $step_1$ $step_2$`)` | If *lisp-expr* evaluates to *true* then applies $step_1$.<br>Otherwise, applies $step_2$. |
| `(TRY` $step_1$ $step_2$ $step_3$`)` | Tries $step_1$; if it modifies the proof state then applies $step_2$.<br>Otherwise, applies $step_3$. |
| `(ELSE` $step_1$ $step_2$`)` | Behaves as `(TRY` $step_1$ `(SKIP)` $step_2$`)`. |
| `(SPREAD` *step* `(`$step_1$ `...` $step_n$`))` | Applies *step* and spreads $step_1$ to $step_n$ over the new subgoals. |
| `(BRANCH` *step* `(`$step_1$ `...` $step_n$`))` | Like `SPREAD` but reuses $step_n$ on any extra subgoals. |
| `(REPEAT` *step*`)` | Iterates *step* until it does nothing down the main proof branch. |
| `(REPEAT*` *step*`)` | Iterates *step* until it does nothing down all branches. |
| `(WITH-LABELS` *step* `(`$labs_1$ `...` $labs_n$`))` | Applies *step*; then labels all new formulae in the new subgoals with $labs_1$ to $labs_n$. |
| `(LET ((`$v_1$ $lisp\text{-}expr_1$`) ...`<br>`(`$v_n$ $lisp\text{-}expr_n$`))` *step*`)` | Applies a new command that is just like `step`, but where<br>$v_i$ has been replaced by the evaluation of $lisp\text{-}expr_i$ for $1 \leq i \leq n$. |

**Table 1.** PVS strategicals

behavior of Strategy (1) is to use the strategy expression in (1) as the body of a defined rule, as described in Section 2.2. Another way is to "wrap" it with the command `APPLY`, as in:

```
(APPLY (THEN (LIFT-IF) (PROP) (ASSERT) (FAIL)))                (2)
```

Finally, one may catch the action of `FAIL` with the command `TRY`. For more on both `TRY` and the use of wrappers, see Section 3.

Note that the two strategicals `IF` and `LET` allow the introduction of Lisp code into a strategy. Strategies that incorporate Lisp code are more sophisticated than Strategies (1) and (2). The Lisp code generally uses information about the current proof state, though a few useful things can be done by using Lisp code to set and observe global variables. Strategies that use information about the proof state are discussed later in Section 3.

### 2.2 Defined rules and strategies.

PVS proof rules are of two kinds: *primitive* rules and *defined* rules. Both primitive and defined rules behave like atomic steps when applied to appropriate arguments, but, unlike a primitive rule, a defined rule is derived from a strategy expression. The strategy expression corresponding to a defined rule can be observed in PVS by typing:

```
M-x help-pvs-prover-strategy
```

Also, the documentation string for a strategy can be viewed within the prover via the command `HELP`.

A defined rule is created by applying the PVS macro `defstep`. Paraphrased from the PVS Prover Guide [10, 11], the format for `defstep` is:

```
(defstep name
         parameter-list
         strategy-expression                    (3)
         documentation-string
         format-string )
```

The *parameter-list*, whose precise description can be found in [10, 11], can contain required arguments plus `&optional` and `&rest` parts, rather like the parameter list in a Lisp function definition. The *documentation-string* is generally used to describe the effect of applying the strategy; it is printed interactively as part of the documentation of proof steps that is printed by the "help" facilities of PVS, e.g., when one types `(HELP` *name*`)` during a proof, or `M-x help-pvs-prover` or `M-x help-pvs-prover-strategy` followed by *name* at any time when using PVS. The *format-string* is printed interactively when the defined rule *name succeeds*, i.e., completes the proof of the current goal, or when it returns one or more subgoals. In addition to creating a new defined rule *name*, the macro `defstep` also creates a named strategy *name*\$. Variants of `defstep` include `defhelper`, which does not require the *documentation-string* or *format-string* arguments, and `defstrat`, which does not require the *format-string* argument. The macro `defhelper` is intended for defining "internal" auxiliary steps that can be used in other strategies, while `defstrat` defines a strategy without a corresponding (atomic) defined rule.

Strategy (1) can be turned into the defined rule `PROP_PROBE` using the definition:

```
(defstep PROP_PROBE ()
         (THEN (LIFT-IF) (PROP) (ASSERT) (FAIL))                    (4)
         "Checks for a trivial proof" "By simple reasoning")
```

Once the definition of `PROP_PROBE` has been loaded into PVS, the desired effect of Strategy (1) can be accomplished by just typing `(PROP_PROBE)` when prompted by PVS for a proof rule. Because Strategy (1) does not refer to any unbound parameter names, the effect of `(PROP_PROBE)` is equivalent to that of Strategy (1) wrapped in `(APPLY ...)`. The exact effect of Strategy (1), in which one sees all the steps in the reasoning, can be duplicated by typing `(PROP_PROBE$)` when prompted for a rule.

By allowing the possibility of parameters, the macro `defstep` allows a strategy (as well as its corresponding defined rule) to be applied in an environment where the parameter names are bound to specific values. The *format-string* in the definition of a rule with parameters can refer to these parameters: any inclusion of ∼`a` in the format string is replaced by the value of an actual parameter, with successive ∼`a`'s picking up successive parameters.

A simple example of a new rule with all these features is the rule `suppose`, whose definition is in Figure 1.[4] The rule `suppose` incorporates formula labeling and comments into the simplest version of the PVS command

```
(defstep suppose (x)
  (let ((suppstring (format nil "Suppose ~a" x))
        (nsuppstring
          (format nil "Suppose not [~a]" x)))
  (branch (with-labels (case x) (("Suppose")("Suppose not")))
          ((comment suppstring) (comment nsuppstring))))
  "For doing a simple case split and tracking the cases"
  "First supposing ~a true and then supposing it false")
```

**Fig. 1.** Definition of a rule with a parameter and a *format-string* that refers to it.

`CASE`. The strategy expression body of `suppose` uses the strategicals `LET`, `WITH-LABELS`, and `BRANCH`. With `LET`, it incorporates Lisp code that computes two comment strings. Using `WITH-LABELS`, it applies the labels from the first list `("Suppose")` to new formulae in the first new subgoal, and the labels from the second list `("Suppose not")` to new formulae in the second new subgoal. Since each of the first and second new subgoals have just one new formula, and these new formulae represent, respectively, the meanings of `x` and `(NOT x)`, they are labeled appropriately. The second argument of `BRANCH` is a list of two commands, which will be applied respectively to the new subgoals. Each of these commands adds its argument as a comment in the subgoal to which it is applied; this comment will appear above the sequent when the subgoal is displayed. Each comment will also be recorded in the saved proof at the beginning of the new proof branch starting at its associated proof goal. The use of labels and comments will be discussed further in Section 3.

[4] Though we use a mixture of upper and lower case versions of names in this tutorial, it is safest to use only lower case in actual strategy files; see the PVS release notes at http://pvs.csl.sri.com.

### 2.3 Adding new rules and strategies to PVS.

Once you have defined one or more new rules using `defstep`, `defstrat`, or `defhelper`, you can make your new rule(s) available in PVS by saving the definition(s) in a file named `pvs-strategies` and putting it in the PVS context where you wish to use the new rules. The file `pvs-strategies` does not need to be a physical file, it can be a link to a file containing your definitions. This way, you can keep a set of definitions consistent across several contexts.

The file `pvs-strategies` is loaded when the first proof in a session is being started, or when a new proof is being started after the content of `pvs-strategies` has been changed. Because `pvs-strategies` is loaded into Lisp, it can contain arbitrary Lisp code—not only rule definitions, but function definitions, global variable initializations, load commands, etc. One use of a load command (that is in fact employed by TAME) is to load a set of strategies specific to one context that can be generated from some theory in that context. Further, if `common_strat` is a file containing a set of strategies that you use in all your developments, you can load those strategies by putting the line

```
(load "<PATH>/common_strat")
```

in the file `pvs-strategies`, where *<PATH>* is the path where the file `common_strat` is found. Section 3 describes some possible uses of functions and global variables.

For testing purposes, one can introduce strategy definitions directly from the command line:

```
(LISP (DEFSTEP strat-name ...))
```

To redefine one later, recall the previous command input using `M-s` or `M-r`, then edit the definition and resubmit it. This technique allows for quick tests or explorations of small strategies.

### 2.4 Creating a strategy package.

If a set of definitions is general enough to be used in several developments or to be used by other PVS users, you may want to pack them as a prelude library extension. The basic functionality of prelude library extensions has been available in older versions of PVS. However, it became fully operational and simple to use in PVS 3.1. A prelude library extension is a set of PVS theories, strategies, and Lisp code that are available to the user as if they were part of the PVS prelude context. As the developer of a prelude library extension, make a directory *MyPackage* and put the following files in it:

- Files `*.pvs` containing PVS theories that your development requires. These theories become part of the PVS prelude theories; therefore, be careful not to introduce inconsistencies.
- A file *my-strat* containing the new strategies.
- A file `pvs-lib.lisp` containing

  ```
  (in-package :pvs)
  ;; If your development requires other prelude libraries, then
  ;; uncomment the following line and modify it as appropriate.
  ;; (load-prelude-library "OtherPackage")
  (libload "my-strat")
  ```

- A file `pvs-lib.el` containing Emacs Lisp code that is part of your development.

Once you have put all these files together, instruct the users of your prelude extension to

1. Set the variable `PVS_LIBRARY_PATH` to point to *<PATH>*, where

   *<PATH>/MyPackage*

   is the actual location of your package.
2. Invoke the Emacs command `M-x load-prelude-library` *MyPackage* the first time *MyPackage* is going to be used in a context. Next time that PVS is restarted in the same context, the prelude extension will be automatically reloaded in the environment.

## 3 Some useful techniques for strategy writing

This section describes a set of techniques that can be used by a strategy developer to create sophisticated PVS strategies. These techniques include:

1. Incorporating backtracking with TRY.
2. Controlling standard PVS steps with appropriate arguments.
3. Observing the proof state.
4. Probing the CLOS structure of the proof state.
5. Defining helper functions in Lisp.
6. Carefully using global variables.
7. Computing a command in Lisp, and then invoking it.
8. Using auxiliary lemmas for rewriting and forward chaining.
9. Using labels and comments.
10. Using functions from PVS.
11. Applying wrappers.
12. Naming subexpressions of complex expressions.
13. Using templates.
14. Comparing proof step definitions using PVS's multiple proof feature.

The TAME [1] strategies and the strategy packages Manip [5] and Field [6] all employ many or all of these techniques. Below, we illustrate how each individual technique can be used to advantage.

### 3.1 Using TRY for backtracking.

Backtracking is a powerful technique for automatic proof search. It enables the restoring of an original proof state after an unsuccessful proof attempt. In PVS, backtracking is achieved by a careful crafting of TRY, FAIL, and atomic proof rules.

The TRY command in PVS combines a conditional and a backtracking control structure. As a conditional control structure, TRY performs an action based on the progress made by a proof command on the current proof state. For instance, the strategy expression

```
(TRY (THEN (LIFT-IF) (PROP) (ASSERT))
     (COMMENT "Progressing ...")
     (SKIP))
```

applies the proof command (THEN (LIFT-IF) (PROP) (ASSERT)). If it does something, i.e., it modifies the current proof state, the comment "Progressing ..." is added to the new proof state. Otherwise, the strategy expression performs the proof command (SKIP) and does nothing else.

On the other hand, the third argument of TRY is a backtracking alternative to failures signaled in its first argument. Failures in TRY's second and third arguments are propagated out of the command. The following semantics, based on an informal set of rules provided by N. Shankar, exposes some technicalities of the behavior of TRY.

We assume that any proof command evaluates to one of the following states:

- *skip*: If the proof states remains unchanged.
- *failure*: If a failure is signaled.
- *success*: If the current goal is discharged.
- *subgoals*: If new subgoals are generated.
- *backtracking*: If backtracking is required.

The evaluation of SKIP, FAIL, and TRY is given by the function $|.|$ as follows

- $|$(SKIP)$| = \textit{skip}$.
- $|$(FAIL)$| = \textit{failure}$.

- $|\texttt{(TRY A B C)}| = \begin{cases} |\texttt{C}| & \text{if } |\texttt{A}| \in \{\mathit{skip}, \mathit{backtracking}\} \\ |\texttt{A}| & \text{if } |\texttt{A}| \in \{\mathit{failure}, \mathit{success}\} \\ \mathit{backtracking} & \text{if } |\texttt{A}| = \mathit{subgoals}, |\texttt{B}| \in \{\mathit{failure}, \mathit{backtracking}\} \\ \mathit{subgoals} & \text{if } |\texttt{A}| = \mathit{subgoals}, |\texttt{B}| \in \{\mathit{skip}, \mathit{subgoals}\} \\ \mathit{success} & \text{if } |\texttt{A}| = \mathit{subgoals}, |\texttt{B}| = \mathit{success} \end{cases}$

To complete the description of `TRY`'s behavior, it is necessary to consider that

- The states *failure* and *backtracking* do not propagate out of atomic proof rules, i.e., if the strategy expression of the atomic proof rule `S` evaluates to either *failure* or *backtracking*, then $|\texttt{S}| = \mathit{skip}$.
- At the top-level, the state *failure* forces the theorem prover to exit, while the state *backtracking* evaluates to *skip*.

For instance,

- $|\texttt{(TRY (SKIP) (ASSERT) (FAIL))}| = \mathit{failure}$.
- $|\texttt{(TRY (TRY (FAIL) A B) C D)}| = \mathit{failure}$.
- $|\texttt{(TRY (TRY A (FAIL) B) C D)}| = |\texttt{D}|$, if $|\texttt{A}| = \mathit{subgoals}$.
- $|\texttt{(TRY A (TRY B (FAIL) C) D)}| = \mathit{backtracking}$, if $|\texttt{A}| = |\texttt{B}| = \mathit{subgoals}$.

The strategy expression

```
(TRY (TRY (THEN (LIFT-IF) (PROP) (ASSERT)) (FAIL) (SKIP))
     step1
     step2)
```

applies the proof command `(THEN (LIFT-IF) (PROP) (ASSERT))`. If that command discharges the current goal, then it does nothing else. Otherwise, it backtracks to the original proof state and attempts a new proof with the command $\texttt{step}_2$. Since `FAIL` does not propagate out of atomic proof rules, i.e., it evaluates to *skip*, the logical behavior of the above strategy expression is equivalent to that of the strategy expression `(APPLY (THEN (LIFT-IF) (PROP) (ASSERT) (FAIL)))` when $\mathit{step}_2 = \texttt{(SKIP)}$.

The `TRY` command is not symmetric: failures signaled in its second argument is not handled in the same way as failures signaled in its third argument. This makes the analysis of failure propagation difficult and error prone. In particular, some PVS commands, such as `THEN`, `ELSE`, `REPEAT`, `SPREAD`, etc., are implemented with `TRY`, and their behavior with respect to failure propagation and backtracking is not easy to characterize. For instance, $|\texttt{(THEN } \mathit{step}_1 \ldots \mathit{step}_n \texttt{ (FAIL))}|$ is

- *failure*, if $n = 0$ or $|\mathit{step}_i| = \mathit{skip}$ for $1 \leq i \leq n$.
- *backtracking*, otherwise.

In general, it is a good practice to wrap as atomic proof rules the strategy expressions that can generate failures.

For the interested reader, the experimental package Practicals, available at http://research.nianet.org/fm-at-nia/Practicals, provides a redesigned set of strategicals for catching and signaling failures, as well as additional control structures for programming PVS strategies.

### 3.2 Controlling standard PVS steps.

When one needs finer control in a strategy, one sometimes needs to use variants of the standard PVS steps that do either less or more than the default actions of these steps. For example, the PVS command

(EXPAND *name*)

does not simply expand the definition of *name*, but performs some simplifications as well. This can be inconvenient; e.g., since one of these simplifications can be a `(LIFT-IF)`, it is possible for a quantified formula involving an `IF-THEN-ELSE` to become an `IF-THEN-ELSE` with two quantified formulae as branches, complicating a strategy involving skolemization or instantiation. To obtain the effect of simply expanding the definition of *name*, one should instead use the PVS command

```
(EXPAND name :ASSERT? NONE).
```

Other example PVS steps that can be made to do less for finer control are `SPLIT` and `FLATTEN`. Using the optional `:depth` argument, `SPLIT` can be prevented from producing more subgoals than one desires. One application of this technique is in the definition of the simple strategy `modus-ponens`:

```
(defstep modus-ponens (formnum)
   (spread (split formnum :depth 1) ((skip)(skip)))
   "Replaces antecedent formulae A and A => B by A and B when
    the formula A => B is labeled by formnum"
   "Performing Modus Ponens")
```

Note that while the PVS rule `ASSERT` can sometimes be used to discharge the hypothesis of an implication, `ASSERT` may cause further changes, and it does not discharge a hypothesis that is not a simple expression. The rule `modus-ponens` permits one to discharge the hypothesis of an implication, without doing more (or less).

Because controlling the number of subgoals in a strategy can be important, being able to apply fine control to `SPLIT` is useful. However, one can also apply fine control to `FLATTEN` as well. This is done by replacing it with `FLATTEN-DISJUNCT` with an appropriate `:depth` argument.

One case in which the default action of a PVS step may be too limited is in a context where there is extensive use of `CASES` expressions. The default of `ASSERT` and `SIMPLIFY` is to not simplify inside these expressions. This choice often results in more efficient proofs, but experience has shown this may not be true when proofs involve large, complex, and possibly many-layered `CASES` expressions. In such a case, one may wish to use `(ASSERT :CASES-REWRITE? T)` and `(SIMPLIFY :CASES-REWRITE? T)` instead.

### 3.3 Observing the proof state.

The PVS proof state and related data structures are represented as classes in the Common Lisp Object System (CLOS). In particular, during the execution of any proof in PVS:

- The current proof state is in the global variable `*ps*`.
- The current proof goal is in the global variable `*goal*`. It can be also accessed as `(current-goal *ps*)`.
- The list of current sequent formulae, each one an instances of the CLOS class `s-formula`, can be accessed as `(s-forms (current-goal *ps*))`.

A more comprehensive list of PVS global variables and data structures and the information they contain can be found in [10, 11].

The proof state (and in fact the value of any Lisp expression) can be observed during a proof using the proof command `LISP`. Thus, to observe the sequent formulae of the current goal at some point in the proof, one can issue

```
(LISP (s-forms (current-goal *ps*)))     (5)
```

at the top-level. When making extensive observations about the proof state, it can become inconvenient to have to embed all the Lisp expressions to be evaluated in a `LISP` command. Another inconvenience of this command is that it interleaves the desired information with repetitions of the current proof goal, making it difficult to make a coherent sequence of observations. (This applies only to PVS versions earlier than 3.1.) An alternative is to send Lisp into a break; this can be done by typing `(LISP (BREAK))`.

Each `s-formula` in `(s-forms (current-goal *ps*))` corresponds to one of the labeled formulae in the sequent of the current goal. An example of how a list of sequent formulae appears when displayed is:

```
(NOT A B C NOT D E)     (6)
```

where `A`, `B`, `C`, `D`, and `E` represent particular PVS formulae. The actual members of the list (6) print out as `NOT A`, `B`, `C`, `NOT D`, `E`. The list (6) represents the sequent:

```
[-1]   A
[-2]   D
|-------                (7)
[1]    B
[2]    C
[3]    E
```

(or a variant in which some square brackets are replace by curly braces). In particular, the negative formulae, in order, correspond to the sequent formulae numbered `-1`, `-2`, and so on, while the positive formulae, in order, correspond to the sequent formulae numbered `1`, `2`, and so on. In general, the list of antecedent (negative) formulae and consequent (positive) formulae can be extracted from the proof state as `(n-sforms (current-goal *ps*))` and `(p-sforms (current-goal *ps*))`, respectively.

Note that formulae in the antecedent, such as `A` and `D` in the sequent (7), appear negated in the representation of the PVS proof state. The following Lisp code retrieves a formula in positive form, i.e., as it appears to the user in the PVS theorem prover, from the formula number:[5]

```
; Get formula from current goal (unnegated if antecedent formula)
; Assumes that fnum is a formula number
(defun get-fnum (fnum)
  (let ((index (- (abs fnum) 1))
        (goal (current-goal *ps*)))
    (if (> fnum 0)
        (formula (nth index (p-sforms goal)))
        (argument (formula (nth index (n-sforms goal)))))))
```

To determine that one needs `argument` and `formula` to extract the desired part of an `s-formula` in `(p-sforms goal)` and `(n-sforms goal)`, one can use technique 4 described in Section 3.4.

The inverse of the operation `get-fnum` is to find the formula number or numbers corresponding to formulae with a given property. The PVS Lisp function `(gather-fnums` *`s-forms yes-fnums no-fnums pred`*`)`, described in [10, 11], returns the list of formula numbers (taken from *yes-fnums*/*no-fnums*) of sequent formulae in *s-forms* that satisfy *pred*. For example, given the property

```
(defun is-forall (sform) (forall-expr? (formula sform)))
```

the Lisp code:

```
(gather-fnums (s-form *goals*) ’* nil #’is-forall)          (8)
```

retrieves all the formula numbers in the current sequent that are universally quantified.

### 3.4 Using CLOS probes.

Most values manipulated by PVS proof steps are CLOS objects. For instance, `*ps*` is a CLOS object which has a component `current-goal`; in turn, `(current-goal *ps*)` is a CLOS object which has a component `(s-forms (current-goal *ps*))`. To probe the CLOS structure of an object and its components, one can use the Lisp functions `describe` or `show`. Given an object *object*, one can probe its CLOS representation in depth by repeatedly using `describe` to discover components to be probed further:

```
(describe object)
(describe (component object))
(describe (component (component object)))
. . .
```

[5] More involved versions of this function that take care of special symbols, labels, and error handling are available in the Manip (http://shemesh.larc.nasa.gov/people/bld/manip.html) and Field (http://research.nianet.org/∼munoz/Field) packages.

The function describe provides explicit names of the component slots in the representations of objects, and these names can then be used like function names to retrieve the elements in these slots, which are themselves objects. The description of *object* starts with a sentence of the form:

*object* is an instance of #<STANDARD-CLASS *object-class*>

This information generally tells you that *object-class*? is a recognizer for objects of class *object-class*. An element x of class *object-class* can also be recognized by the fact that (typep x *object-class*) will be true.

When one needs a shortcut to a sequence of CLOS probes, or when one cannot be sure of the sequence or sequences needed, one can use the function mapobject. The function mapobject provides an analog for objects of mapcar for lists: it traverses (most of) the object structure, applying a given function to each component. Thus, to determine whether an s-formula sform contains a universal or existential quantifier, one can use the predicate has-quantifier, defined as:

```
(defun has-quantifier (sform)
   (let ((has-quant nil))
         (mapobject #'(lambda (x) (if has-quant t
                                      (when (or (forall-expr? x)
                                                (exists-expr? x))
                                            (setq has-quant t) t)))
                    sform)
   has-quant))
```

### 3.5 Defining helper functions.

Helper functions from Lisp are useful for writing strategy expressions that involve Lisp code, i.e., those using either LET or IF. They generally involve CLOS probes into the current proof state; thus, we have already seen the following examples of potential helper functions in Sections 3.3 and 3.4:

- get-fnum
- is-forall
- has-quantifier

The helper function get-fnum is used in a LET in the strategy add-eq in Figure 12 below in Section 4.1. Examples of definition and use of additional helper functions can be found below in Section 4.2.

One can classify Lisp helper functions into general purpose and special purpose functions. General purpose helper functions include functions such as get-fnum and is-forall, which can be applied, respectively, to any valid formula number (or label) and to any valid s-formula. An example of a special purpose helper function is the function get_sk_constructor_exprs from Figure 18 in Section 4.2. The function get_sk_constructor_exprs will cause a Lisp break if it is called incorrectly; it must be called only on s-formulae of a very limited form. Special purpose helper functions generally use CLOS probes that are either unusual or grouped in a long series, making them hard to match. Thus, extra care must be taken when these functions are used: they should either be used in a context where they are known to be valid (as in the example in Section 4.2), or else a strategy should test the classes of a CLOS structure and its substructures before applying them.

Alternatively, helper functions can take advantage of Common Lisp's exception handling features to deal with errors. While the language specification [12] explains these features in full detail, the following idiom based on the handler-case macro is sufficient for most applications:

```
(handler-case
    <expression>
  (error (condition) <alt value/action>))
```

If the evaluation of <expression> proceeds normally, its value is returned as the value of the handler-case construct. If the evaluation of <expression> raises any type of Lisp error, it will be caught and the <alt value/action> will be returned/performed.

### 3.6 Using global variables.

As in any type of programming, global variables must be used carefully in PVS. Clearly, two rules should be followed:

1. Choose variable names not already in use;
2. *Never* change a predefined PVS global variable, such as `*ps*` or `*goal*`.

Towards satisfying rule 1, one can easily test whether a variable `x` is currently in use: either type the command `(LISP x)` when the prover is running, or else type `x` into the `*pvs*` buffer when the prover is not running. For run-time use, the Lisp functions `boundp` and `fboundp` are available to test whether a symbol is currently bound as a variable or a function. Note that if one violates rule 2 by changing `*ps*`, even if the new value of `*ps*` is a valid proof state object, one is creating a nonconservative extension of PVS, and losing PVS's soundness guarantees.

In general, global variables should be avoided. However, they can be useful as switches. In TAME, for example, the user can control whether saved proofs will be in verbose form (recording specific facts introduced in the proofs), or in bare-bones, nonverbose form, by invoking the rules `(VERBOSE)` and `(NONVERBOSE)`. These rules work simply by setting a specific global variable to `t` or `nil`.

### 3.7 Computing the command to be invoked.

When a strategy definition has parameters, it can happen that the proof step the strategy is to implement depends on some information that must be computed from the parameter values.

A typical example is when the strategy definition has an `&rest` parameter. When the strategy (or corresponding defined rule) is applied, the `&rest` parameter is bound to a list of actual parameters. The strategy will typically need to extract the `car` and `cdr` of this list as it proceeds. Because proof rules cannot be applied directly to `car` or `cdr` expressions, commands involving the application of proof rules to the `car` or `cdr` of a list of actual parameters must be first computed and then called. Examples where this technique is used are in the definitions of the strategies `apply-lemma`, `else*`, and `rewrite-one` in Figures 7, 8, and 9, respectively, in Section 4.1. (Note that `apply-lemma` computes two commands, `lemma-step` and `inst-step`, though actually, only `inst-step`, which depends on the `&rest` parameter, needs to be computed.)

Another example in Section 4.1 in which commands are computed is in the strategy `add-eq` in Figure 12. Here, two commands `case-step` and `steplist` are computed. Because `case-step` applies `CASE` to values computed from its formula-number arguments, it *must* be computed. Here again, one of the steps, `steplist`, need not be computed. However, note that "unnecessary" computation of a step often adds to the readability of a strategy definition, particularly when companion steps must be computed.

### 3.8 Rewriting and forward chaining with lemmas.

PVS provides a variety of steps for controlling the use of rewrites. An example of a strategy that takes advantage of PVS's `REWRITE` rule is `rewrite-one` in Figure 9 on page 18. The strategy `rewrite-one` does rewriting once using its lemma arguments as the rewrite rules.

For automatic or "large step" strategies, it is useful to do auto-rewriting. Auto-rewriting on a set of lemmas can be initiated by calling `AUTO-REWRITE` on a list of the lemmas. Similarly, auto-rewriting on a set of lemmas can be terminated by calling `STOP-REWRITE` on a list of the lemmas. Rather than explicitly listing lemmas, it can be convenient to collect a set of rewrites into a theory, and calling `AUTO-REWRITE-THEORY` (and `STOP-REWRITE-THEORY`) on that theory. Any lemmas installed as auto-rewrites will be used as rewrites whenever `DO-REWRITE` is called. Since `ASSERT` and `SIMPLIFY` call `DO-REWRITE`, these two PVS strategies also cause auto-rewrites to be performed. Auto-rewrites must clearly be used carefully, to avoid possible nontermination of rewriting.

Rewrites in PVS can be *conditional* rewrites, where a rewrite rule is applied only if its condition simplifies to `TRUE`. Lemmas with conditions (i.e., hypotheses) can also be used for *forward chaining*, in which the (possibly parameterized) hypothesis is matched to some formula or formulae in the current sequent. Any match defines an instance of the conclusion, that is then added as an antecedent formula to the current sequent. The PVS rule `FORWARD-CHAIN` allows forward chaining on a lemma (or on a formula in the current sequent). Note that using `REPEAT` or `REPEAT*` in combination with `FORWARD-CHAIN` can lead to nontermination if the conclusion of the lemma used for forward chaining matches its hypothesis; therefore, care must also be taken in using repeated forward chaining. There is currently no FORWARD-CHAIN-THEORY, although one is expected to be available in the near future [9].

There are many uses for rewriting and forward chaining; for example, TAME uses both auto-rewriting and forward chaining to automate certain reasoning about the relationships between constructor and accessor functions in DATATYPEs that is not handled by ASSERT or GRIND.

### 3.9 Using labels and comments.

A simple use of comments and labels in a strategy has already been illustrated in Figure 1, which shows the definition of the strategy suppose. This strategy uses WITH-LABELS to introduce a set of labels simultaneously, and the command COMMENT is for introducing comments. There is also a command LABEL for introducing a single label.

Labels are applied to formulae. Once a formula has a label, it can be referred to by that label. This fact has many uses in strategies. For example, a labeled formula can be hidden and revealed by calling HIDE and REVEAL on its label. One use of this device is to prevent expansion of definitions in the labeled formula except when such expansion is desired. Another example use for labels is to coordinate skolemization of one quantified formula with instantiation of another. It is possible to give a formula multiple labels by using the optional argument :push? T with either WITH-LABELS or LABEL. This allows all information in original labels to be retained, while adding new information, so that formulae can, if desired, be included in multiple categories for multiple purposes. The use of labels can also increase the stability of strategies. For simplicity, several example strategies in this tutorial use explicit references to formula numbers (see Sections 3.10 and 4). However, provided one knows the number, ordering, and nature of the new formulae that will be created by a command, by wrapping that command using WITH-LABELS and an appropriate list of labels, one can avoid explicit formula number references. On the assumption that the ordering in the set of newly created formula is less likely to change in new PVS versions than the explicit formula numbers that will be assigned to the new formulae, user strategies using WITH-LABELS and label references will be less fragile than those using explicit formula number references. An example of how labels appear in a sequent is shown in Figure 2, which shows a subgoal from a TAME proof for the invariant lemma lemma_5 of *TIP* [4, 3].

```
lemma_5.1 :
;;;Case add_child(addE_action)

{-1,(pre-state-reachable)}
      reachable(prestate)
{-2,(inductive-hypothesis)}
      length(mq(basic(prestate))(e_theorem)) <= 1
{-3,(general-precondition)}
      enabled_general(add_child(addE_action), prestate)
{-4,(specific-precondition)}
      enabled_specific(add_child(addE_action), prestate)
{-5,(post-state-reachable)}
      reachable(poststate)
  |-------
{1,(inductive-conclusion)}
      IF NOT (mq(basic(prestate))(addE_action) = null)
      THEN length(mq(basic(prestate)) WITH
                  [(addE_action) :=
                   cdr(mq(basic(prestate))(addE_action))]
                  (e_theorem))
      ELSE length(mq(basic(prestate)(e_theorem)))
      ENDIF
       <= 1
```

**Fig. 2.** An example TAME sequent illustrating labels.

In contrast to labels, which attach to formulae, comments attach to subgoals. Note that subgoal in Figure 2 also contains a comment which identifies the case to which the subgoal corresponds. Comments also appear in saved proofs, immediately after the command that introduces them. When a command creates branches, it is possible to "label" the branches in the saved proof with comments by wrapping the command creating the branches in a SPREAD or BRANCH construct that then applies multiple calls to COMMENT to the branches, as illustrated in Figure 1 on page 4. An example saved proof showing how comments can be used to make saved proofs more understandable is shown in Figure 3, which shows the saved TAME proof of the the *TIP* property lemma_5. The subgoal in Figure 2 is the first subgoal of the first branch of the proof in Figure 3, so the comment in this subgoal "labels" the first branch of the proof. The saved proof in Figure 3 illustrates the effect of suppose, and also shows that comments can be used to capture ephemeral information from proof goals, such as facts being used in the reasoning.

```
Inv_5(s:states): bool = (FORALL (e:Edges): length(mq(e,s)) <= 1);
;;; Proof lemma_5-like-hand for formula tip_invariants.lemma_5
(""
 (AUTO_INDUCT)
 (("1" ;;Case add_child(addE_action)
   (APPLY_SPECIFIC_PRECOND)
   ;;Applying the precondition
   ;;init(target(addE_action), prestate)
   ;; & NOT (mq(addE_action, prestate)=null)
   (SUPPOSE "e_theorem = addE_action")
   (("1" ;;Suppose e_theorem = addE_action
     (TRY_SIMP))
    ("2" ;;Suppose not [e_theorem = addE_action]
     (TRY_SIMP))))
  ("2" ;;Case children_known(childV_action)
   (SUPPOSE "source(e_theorem) = childV_action")
   (("1" ;;Suppose source(e_theorem) = childV_action
     (APPLY_SPECIFIC_PRECOND)
     ;;Applying the precondition
     ;;init(childV_action, prestate)
     ;;    &
     ;;  (FORALL (e: Edges):
     ;;     FORALL (f: tov(childV_action)):
     ;;       child(e, prestate) OR child(f, prestate) OR e = f)
     (APPLY_INV_LEMMA "2" "e_theorem")
     ;;Applying the lemma
     ;;(FORALL (e: Edges): init(source(e), prestate)
     ;; => mq(e, prestate)=null)
     (TRY_SIMP))
    ("2" ;;Suppose not [source(e_theorem) = childV_action]
     (TRY_SIMP))))
  ("3" ;;Case ack(ackE_action)
   (SUPPOSE "e_theorem = ackE_action")
   (("1" ;;Suppose e_theorem = ackE_action
     (APPLY_SPECIFIC_PRECOND)
     ;;Applying the precondition
     ;;NOT (init(target(ackE_action), prestate))
     ;;    & NOT (mq(ackE_action, prestate) = null)
     (TRY_SIMP))
    ("2" ;;Suppose not [e_theorem = ackE_action]
     (TRY_SIMP)))))
```

**Fig. 3.** A verbose TAME proof illustrating comments in a saved proof.

### 3.10 Using Lisp functions from PVS.

As illustrated in Section 3, one can use PVS Lisp functions documented in [10, 11] in writing Lisp code to be used in strategies.[6] These documented functions can be a convenience in writing Lisp code, but one can generally achieve the same effects in one's Lisp code by combining standard Lisp constructs with CLOS probes. For example, the effect of the code in 8 on page 9, which solves the problem of listing all formula numbers in a goal corresponding to quantified formulae, can also be achieved by the code

```
(gather-fnums-property ‘is-forall (current-goal *ps*))          (9)
```

where gather-fnums-property is defined by:

```
(defun gather-fnums-property (prop goal)
   (let ((negfnums
          (let ((fnum 0))
          (loop for x in (n-sforms goal) do (setq fnum (- fnum 1))
                when (funcall prop x) collect fnum)))
         (posfnums
          (let ((fnum 0))
          (loop for x in (p-sforms goal) do (setq fnum (+ fnum 1))
                when (funcall prop x) collect fnum))))
           (append negfnums posfnums)))
```

However, there are PVS Lisp functions that are not formally documented that allow one to solve problems in ways not so easily duplicated.

Consider the following problem. PVS expressions that are parameters to proof commands are input as strings. In general, these expressions are built from other expressions in the proof state, where they appear as CLOS structures, and converted to strings with the Lisp function format. In some special cases, we may want to perform the inverse operation, i.e., to get a CLOS structure from the string representation of a PVS expression. A simple way to achieve this operation is to bring the PVS expression to the proof state, for example using a harmless (CASE "*expr* = *expr*"), and then observing the CLOS structure of the proof state as explained in Sections 3.3 and 3.4. The following piece of code implements this technique:

```
(LET ((casestr (format nil "(~A) = (~A)" expr expr)))
    (THEN
      (CASE casestr)
      (LET ((closexpr (args1 (get-fnum -1))))
        (THEN
         (DELETE -1)
           ;; closexpr is the CLOS representation of expr
          (... closexpr ...)))))
```

The code above (which makes use of the documented PVS Lisp function args1) has the side effect of temporarily modifying the proof state. In most cases, the modification has no logical consequences. However, if expr generates TCCs, these TCCs will appear in the new proof state.

An alternative, cleaner way to get a CLOS structure of a PVS expression is by using the PVS parser and type-checker functions pc-parse and pc-typecheck directly. These functions are not properly documented and they must be used with care; otherwise, the PVS prover could get into an unstable state. The function (pc-parse *expr gramtyp*) returns a non-type-checked CLOS structure of the expression *expr*. The parameter *gramtyp* is a grammar nonterminal, *in most cases* with the same name as the CLOS type of the structure to be parsed. For instance,

```
(pc-parse "(# x:=1, b:=true #)" ’expr)
```

[6] An API document that covers all the Lisp calls needed for strategies and integration with other tools is being written at SRI [7].

returns the CLOS structure corresponding to the PVS record `(# x:=1, b:= true #)`. On the other hand,

```
(pc-parse "[# x:int, b:bool #]" 'type-expr)
```

returns the CLOS structure corresponding to the PVS type record `[# x:int, b:bool #]`. CLOS structures should not be used in a proof state unless they are appropriately type-checked. The function `(pc-typecheck` *`closexpr`*`)` adds PVS type information to the CLOS structure *`closexpr`*. Usually, a call to `pc-parse` is followed by a call to `pc-typecheck`.

An example where converting a string to a CLOS structure in this fashion is useful is in defining a strategy whose behavior depends on the type of one or more of its arguments. Provided the string `x` names a valid expression that is type correct in the current proof goal, the value of

```
(type (pc-typecheck (pc-parse x 'expr)))                    (10)
```

will be the (CLOS representation of the) type of that expression. (Note that `type` is a CLOS probe—i.e., the name of a slot or method—rather than a function from PVS.) The string

```
(princ-to-string (type (pc-typecheck (pc-parse x 'expr))))
```

can then be compared to any specific type name represented as a string, or, more safely, the (not yet documented) PVS Lisp function `tc-eq` can be used to compare the type (10) with another (analogously computed) type.

### 3.11 Applying wrappers.

Wrappers are strategicals that prevent their strategy arguments from causing unintended effects. We have already seen one example use for wrapping: wrapping a command that may lead to failure in `(APPLY ...)` so that any failure caused will be local (undoing the proof only to the subgoal where the command was applied).

Another instance in which one may wish to use a wrapper is when a strategy has potential side effects, for example through the use of auto-rewrites or global variables, and one wishes to be sure no permanent side effects result from execution of the strategy. Even a strategy that ultimately follows every auto-rewrite command with an appropriate corresponding stop-rewrite command can leave "dangling rewrites" active if it produces multiple branches and proves the last branch before it reaches a needed stop-rewrite command. In such a case one can wrap the strategy, together with a "cleanup step" that removes any potential side effects, in the strategical `unwind_protect` defined in Figure 4. To protect against auto-rewrites remaining unintentionally active, the `cleanup-step` argument to `unwind-protect` can be a strategy that performs the needed sequence of stop-rewrite commands.

```
(defstep unwind-protect (main-step cleanup-step)
  (spread (case "id(true)")
          ((then (delete -1) main-step)
           (then cleanup-step (expand "id" 1))))
  "Invoke MAIN-STEP followed by CLEANUP-STEP, which is performed
even if MAIN-STEP leads to a proof of the current goal."
  "Invoking proof step with cleanup")
```

**Fig. 4.** An example "safety wrapper" strategical.

### 3.12 Naming a subexpression.

Field axioms, such as associativity, commutativity, distributivity, etc., are known to the PVS decisions procedures. For instance, the sequent

```
|-------
{1}  x * x >= 0
```

is automatically discharged by the proof command (GRIND). Surprisingly, the sequent

```
|-------
{1}  (x - 1) * (x - 1) >= 0
```

is not discharged by (GRIND). In this case, GRIND yields the sequent:

```
|-------
{1}   1 - x + (x * x - x) >= 0
```

which is not further simplified by the PVS decision procedures.

The reason for this behavior is that the decision procedures always apply fields axioms, and in particular the distributive law, before other simplifications. Since PVS does not provide an explicit mechanism to customize these simplifications, they can be problematic for writing strategies where proof control is fundamental.

One way to avoid certain implicit simplifications, such as the distributive law, is to wrap a subexpression in an application of the identity function, e.g., id(x - 1). This renders the expression ineligible for the distributive law. When this protection is no longer desired, the id function can be expanded to restore the original expression. For simple cases this technique is often adequate.

For more advanced uses, undesired simplification can be avoided by *naming* the expression that should not be simplified. This can be achieved with the commands NAME and REPLACE, or the command NAME-REPLACE. The commands NAME introduce a new name definition to the current sequent. This name is then used by REPLACE to abbreviate the original expression.

Figure 5 illustrates a strategy that blocks the first application of the distributive law in a formula by introducing a new name. The strategy NODISTR uses helper functions get-fnum, get-newname, get-distr-expr, and get-distr-plus. The function get-fnum (see Section 3.3) gets the formula in the formula number fnum. New names are created by the function get-newname, which increments the global variable newname each time a new name is required. Finally, the functions get-distr-expr and get-distr-plus descend the formula tree to find the first expression having the form $(x + y) * z$ or $z * (x + y)$. These functions use PVS functions infix-application? that checks if a formula is an infix application, name-expr? that checks if an operator is a name (as opposed to a lambda expression), and args1 and args2 that projects the first and second argument of an application, respectively.

For instance, (NODISTR 1) applied to the sequent

```
|-------
{1}  (x - 1) * (x - 1) >= 0
```

yields the sequent[7]

```
{-1}  (x - 1) = v7__
  |-------
{1}   v7__ * v7__ >= 0
```

When strategies introduce new names automatically, there is the possibility of conflicts with user supplied names. To prevent such clashes, we recommend following a naming convention that yields distinctive identifiers. For example, the convention followed by the function get-newname is to create identifiers with two trailing underscore characters.

The strategy NODISTR can be used to improve the automation provided by GRIND on the field of real numbers. For example, the simple strategy GRINOD in Figure 6 discharges, among others, the following sequent

```
  |-------
{1}   FORALL (x: real): (x - 1) * (x - 2) * (x - 1) * (x - 2) >= 0
```

[7] The name of the new variable may be different.

```
;; Strategy definition
(defstrat NODISTR (fnum)
  (LET ((form (get-fnum fnum))
        (name (get-newname))
        (expr (get-distr-expr form))
        (str  (when expr (format nil "~A" expr))))
    (IF str (NAME-REPLACE name str :hide? nil) (SKIP)))
  "Introduces a new name in ~A to block the distributive law")

;; Generating new names
(setq newname 0)

(defun get-newname ()
  (progn (setq newname (+ 1 newname))
  (format nil "v~A__" newname)))

;; Helper functions
(defun get-distr-expr (form)
  (when (and (infix-application? form)
             (name-expr? (operator form)))
    (let ((op (id (operator form))))
      (cond ((member op '(= <= >= < > + - /))
             (or (get-distr-expr (args1 form))
                 (get-distr-expr (args2 form))))
            ((eq op '*)
             (or (get-distr-plus (args1 form))
                 (get-distr-plus (args2 form))))
            (t nil)))))

(defun get-distr-plus (form)
  (when (and (infix-application? form)
             (name-expr? (operator form)))
    (let ((op (id (operator form))))
      (cond ((member op '(+ -)) form)
            ((member op '(* /))
             (or (get-distr-expr (args1 form))
                 (get-distr-expr (args2 form))))
            (t nil)))))
```

**Fig. 5.** Naming a subexpression to block the distributive law

```
(defstrat GRINOD (fnum)
  (THEN (SKOSIMP fnum)
        (REPEAT (NODISTR fnum))
        (GRIND :theories "real_props"))
  "Blocks the distributive law in ~A before applying GRIND")
```

**Fig. 6.** Combining `NODISTR` and `GRIND`

### 3.13 Using templates.

The use of templates is an indirect technique that can be used in strategy development. For example, when one is reasoning in a special domain, one may wish to assume some degree of uniformity either in the objects about which one is reasoning or in the formulations of properties of these objects (or both). Templates allow one to enforce a standard naming scheme for objects and their types or a standard scheme for expressing properties. As a result, strategies based on templates can be based on a certain amount of definite information that allows them to make more reasoning automatic, and thus to achieve larger size proof steps.

Templates for both specifications and lemmas are used to advantage by TAME.

### 3.14 Using PVS's multiple proof feature.

For proof steps that do a significant amount of automatic reasoning, and which therefore can take a long time to execute, efficiency is an important design goal. Once one has designed a strategy that achieves an intended purpose, one can compare the strategy for efficiency against alternate versions by saving proofs that use the different versions. The saved proofs include run time information that can be used for efficiency comparison.

Comparisons for efficiency should be done over several examples, as there are often tradeoffs in the choice between two near-optimal versions of a strategy. Note that the PVS command `TIME`, which is similar to `APPLY` in that it turns a strategy into an atomic rule, has the additional effect of giving timing information for any branches created by the strategy on which the strategy does not terminate. Thus, `TIME` provides an additional resource in studies of efficiency: it can be used for strategy efficiency comparisons between the cases in the branches a strategy generates.

## 4 Examples of strategy design

In this section, we provide several examples to further illustrate the kinds of reasoning steps that can be supported with PVS strategies, and to provide new PVS strategy developers with some useful ideas that they may wish to recycle in their own strategies.

### 4.1 Some small-step strategy examples

The example strategies in this section are geared towards carrying out tasks during interactive proving, and can be viewed as providing slightly more powerful versions of existing prover rules. Included are examples of:

- capturing a commonly used pattern of steps within a single step,
- using `TRY` together with recursion to define a step that iterates a command over the list of arguments to the step,
- forking a "proof obligation" proof to simplify introducing a fact (as a conjecture) on the current proof branch, and
- creating a new arithmetic reasoning step that is not supported by any standard PVS proof step.

Several of these examples also illustrate techniques from Section 3, including computing and then executing a command, use of CLOS probes into the proof state, use of Lisp helper functions, and use of PVS functions.

Figure 7 shows a modest strategy `apply-lemma` that invokes a lemma after accepting a list of expressions for instantiating the variables. The strategy expands into a prover command of the form:

```
(THEN (LEMMA name) (INST -1 expr-1 ... expr-n))
```

Note that the bindings of the LET construct in `apply-lemma` could have been written using Lisp's backquote feature:

```
(let ((lemma-step ‘(lemma ,lemma))
      (inst-step  ‘(inst -1 ,@exprs)))
  (then lemma-step inst-step))
```

```
(defstep apply-lemma (lemma &rest exprs)
  (let ((lemma-step (list 'lemma lemma))
        (inst-step  (cons 'inst (cons -1 exprs))))
    (then lemma-step inst-step))
  "Apply a lemma with explicit variable instantiations.
Lemma variables appear in alphabetical order when introduced
by the LEMMA rule.  That order needs to be observed when
entering EXPRS."
  "~%Invoking lemma ~A on given expressions")
```

**Fig. 7.** Applying a lemma and instantiating its variables.

```
(defstep else* (&rest steps)
  (if (null steps)
      (skip)
      (let ((try-step `(try ,(car steps)
                            (skip)
                            (else*$ ,@(cdr steps)))))
        try-step))
  "Try STEPS in sequence until the first one succeeds."
  "~%Trying steps in sequence")
```

**Fig. 8.** Generalization of the prover's ELSE strategical.

In many cases this type of notation simplifies the coding effort and improves readability. We will make use of it in the remaining examples.

Figure 8 illustrates a basic strategy pattern for trying a series of actions until one succeeds. When the first step is encountered that has an effect on the proof state, the strategy terminates without attempting any of the remaining steps. ELSE* can be thought of as a generalization of the prover's built-in ELSE strategical. It is likely to be useful as a building block for higher level strategies.

The TRY strategical together with recursive invocation is employed to achieve the effect of conditional iteration. For each element of argument STEPS, if trying the step has no effect, ELSE* is invoked again on the remaining steps. TRY is applied to achieve the following general scheme:

```
(TRY current-step (SKIP) recursive-invocation)
```

Note the use of the strategy form (ELSE*$) rather than the rule form (ELSE*) in the recursive invocation. This is a convention often followed in PVS strategies. It ensures that when the top-level command is invoked as a nonatomic strategy, all subordinate strategies will be as well, resulting in a full expansion into predefined rules.

```
(defstep rewrite-one (fnums &rest lemmas)
  (if (null lemmas)
      (skip)
      (let ((try-step
              `(try (rewrite ,(car lemmas) ,fnums)
                    (skip)
                    (rewrite-one$ ,fnums ,@(cdr lemmas)))))
        try-step))
  "Try rewriting LEMMAS in sequence within FNUMS until the
first one succeeds."
  "~%Trying lemma rewrites in sequence")
```

**Fig. 9.** Using the pattern for ELSE* in a more concrete setting.

Figure 9 demonstrates how the pattern of ELSE* can be applied to a more concrete objective. Given a list of lemma names, REWRITE-ONE tries to rewrite with each lemma in turn until one is successful. It also provides an argument FNUMS to control which part of the sequent should be subject to rewriting. It follows the same recursive pattern presented in Figure 8. (See Section 3 for more on the use of term rewriting in PVS.)

```
(defstep claim-cond (cond)
  (let ((case-step (list 'case cond))
        (steplist
          (list '(skip)
                '(try (then (grind) (fail))
                      (skip)
                      (skip-msg "Claim justification not proved"
                                t)))))
    (spread case-step steplist))
 "Try claiming a condition holds.  A proof of the justification
step is attempted using (GRIND)."
  "~%Claiming the condition ~A holds")
```

**Fig. 10.** Claiming a condition and trying to prove its justification.

Figure 10 illustrates a different use for the TRY strategical. In CLAIM-COND, we wish to accept a PVS expression COND as a condition that holds in the current goal and introduce it as a new antecedent formula. We would also like to automatically prove that the condition holds.

To carry out this task, we use CASE to introduce the supposition, then apply GRIND on the second branch generated by CASE to prove that COND holds. If GRIND fails to completely prove the justification, we undo the partial proof and leave it to the user to determine how to proceed. This behavior is obtained using the following scheme on the second branch generated by CASE:

```
(TRY (THEN (GRIND) (FAIL)) (SKIP) (SKIP-MSG message t))
```

Backtracking via FAIL is performed if the subgoal is not completely proved. In this case the SKIP-MSG rule is invoked to display a message to the user that the justification proof did not succeed.

To direct the branching of the proof into subgoals, the SPREAD strategical is used. The first argument to SPREAD is a step that causes branching, which is CASE in this instance. The second argument is a list of steps for the follow-up actions to be performed for each subgoal. The second subgoal represents the justification proof for the claim, where the TRY construct is applied.

```
(defstep equate-terms (lhs rhs)
  (let ((case-eq (list 'case
                       (format nil "(~A) = (~A)" lhs rhs)))
        (steplist
          (list '(replace -1 :hide? t)
                '(try (then (grind) (fail))
                      (skip)
                      (skip-msg "Equate justification not proved"
                                t)))))
    (spread case-eq steplist))
  "Try equating two expressions and replacing the LHS by the RHS.
A proof of the justification step is attempted using (GRIND)."
  "~%Equating two expressions and replacing")
```

**Fig. 11.** Claiming two terms are equal and carrying out replacement.

Figure 11 follows the same pattern as found in Figure 10. EQUATE-TERMS accepts two PVS expressions that are claimed to be equal, then substitutes one for the other. A new antecedent equality LHS = RHS will be added as a claim. REPLACE is applied to substitute RHS for LHS. Then a justification proof to establish the equality is carried out in the same manner as CLAIM-COND.

Forming the CASE command requires some string manipulation, which is implemented using Lisp's FORMAT function. This is an example of a common operation in strategy writing. LET bindings are introduced to allow Lisp code to compute prover command invocations having whatever arguments are necessary.

```
(defstep add-eq (fnum1 fnum2)
  (let ((formula1  (get-fnum fnum1))
        (formula2  (get-fnum fnum2))
        (left-sum  (format nil "~A + ~A"
                           (args1 formula1) (args1 formula2)))
        (right-sum (format nil "~A + ~A"
                           (args2 formula1) (args2 formula2)))
        (case-step `(case ,(format nil "~A = ~A"
                                   left-sum right-sum)))
        (steplist  '((skip) (then (assert) (assert)))))
    (spread case-step steplist))
  "Given two antecedent equalities a = b and c = d, introduce
a new formula relating their sums, a + c = b + d."
  "~%Adding terms from ~A and ~A to derive a new equality")
```

**Fig. 12.** Adding two antecedent equalities to generate a third.

Figure 12 illustrates the extraction of expressions from CLOS objects within the current proof state. ADD-EQ accepts two formula numbers for antecedent equalities involving numeric values. It then introduces a new antecedent equality that sums the two equations, i.e., given equations $a = b$ and $c = d$, it forms $a + c = b + d$. The justification proof consists of two applications of ASSERT, which should be sufficient to prove the subgoal.

To extract terms from the proof state, the formula objects are first retrieved using the Lisp function GET-FNUM described earlier. Assuming the formulae are equalities, their left hand and right hand sides can be accessed using the PVS functions ARGS1 and ARGS2. When supplied as values to the FORMAT function, Lisp renders their textual representations as PVS expressions. This allows ordinary string manipulation to be used to construct new PVS expressions from fragments of the current sequent.

Having formed the new antecedent equality as a text string, an application of the CASE rule is used to achieve the desired effect. In a more realistic strategy development effort, error checking code would be inserted at various places to check for invalid inputs. Strategy writers can decide how important such checking is for the intended purpose of their strategies.

### 4.2 Developing high level strategies: an example

Strategies geared to high level proof automation, either of full proofs or of proof steps at a high conceptual level, almost invariably require use of several of the techniques described in Section 3. To illustrate how some of the techniques described in Sections 2 and 3 can be applied to developing an automatic strategy for proving lemmas belonging to a particular class, we will show how the defined rule adt_unique_strat from TAME was developed.[8] Although adt_unique_strat was developed for TAME, it is useful in any context in which the DATATYPE construct is used: it allows the user to supply a one-step proof for any lemma that asserts that

[8] A little history: development of TAME strategies began with an early version of PVS, in which the PVS step DECOMPOSE-EQUALITY was not a standard proof rule. With this rule, one can write a much simpler version of adt_unique_strat. The example in this section nevertheless serves to illustrate a general approach to creating a specialized high level strategy.

if two elements of the same DATATYPE with the same constructor are equal, then the arguments to which the constructor is applied to obtain these elements must be pairwise equal. Figure 13 shows an example DATATYPE and its "uniqueness properties" taken from the TAME specification of the basic TESLA multicast stream authentication protocol [8, 2]. Such a lemma is a corollary of the fact that the elements of any PVS DATATYPE form a free algebra, that is, a term algebra with no nontrivial equalities between terms. Unfortunately, the automatic PVS proof procedures such as ASSERT, SIMPLIFY, and GRIND do not automatically "know" this information. Moreover, as can be seen from the proof of Receive_unique in Figure 14, one does not really want to make an excursion in a PVS proof to establish this property.

```
actions: DATATYPE
  BEGIN
    nu(timeof:(fintime?)): nu?
    SSend (Si:nat, Sc,Sk1,Sk2:Key, Sm:Message): SSend?
    ASend (Ai:nat, Ac:Commit, Ak1,Ak2:Key, Am:Message): ASend?
    Receive (RSentPacket:SentPacket): Receive?
  END actions

nu_unique: LEMMA FORALL (t1, t2: (fintime?)):
    nu(t1) = nu(t2) => t1 = t2;

Send_unique: LEMMA FORALL (i1,i2:nat, c1,c2,k11,k12,k21,k22: Key,
                           m1,m2:Message):
    SSend(i1,c1,k11,k21,m1) = SSend(i2,c2,k12,k22,m2)
    => i1=i2 & c1=c2 & k11=k12 & k21=k22 & m1=m2;

ASend_unique: LEMMA FORALL(i1,i2:nat, c1,c2:Commit,
                           k11,k12,k21,k22: Key, m1,m2:Message):
    ASend(i1,c1,k11,k21,m1) = ASend(i2,c2,k12,k22,m2)
    => i1=i2 & c1=c2 & k11=k12 & k21=k22 & m1=m2;

Receive_unique: LEMMA FORALL (sp1, sp2: SentPacket):
    Receive(sp1) = Receive(sp2) => sp1 = sp2;
```

**Fig. 13.** Example of a PVS DATATYPE declaration, and its "uniqueness lemmas".

The first step in developing adt_unique_strat is to prove several uniqueness lemmas in PVS and look for patterns. Figure 14 shows the pattern to follow in establishing a uniqueness lemma for a constructor with one parameter: One can see that, after skolemizing and flattening the formula in the lemma, one does two case splits, each based on an equality of an individual skolem constant to an application of the single datatype accessor function RSentPacket for Receive actions to an application of the Receive constructor to the same

```
(""
 (SKOLEM!)
 (FLATTEN)
 (CASE "sp1!1 = RSentPacket(Receive(sp1!1))")
 (("1" (CASE "sp2!1 = RSentPacket(Receive(sp2!1))")
   (("1"
     (APPLY (THEN (REPLACE -1 +) (REPLACE -2 +) (HIDE -1 -2)))
     (REPLACE -1)
     (PROPAX))
    ("2" (ASSERT))))
  ("2" (ASSERT))))
```

**Fig. 14.** Proof of a uniqueness lemma for a DATATYPE constructor with one parameter.

```
(""
 (SKOLEM 1
  ("i_1" "i_2" "c_1" "c_2" "k1_1" "k1_2" "k2_1" "k2_2" "m_1" "m_2"))
 (FLATTEN)
 (SPLIT)
 (("1" (CASE "i_1 = Si(SSend(i_1,c_1,k1_1,k2_1,m_1))")
   (("1" (CASE "i_2 = Si(SSend(i_2,c_2,k1_2,k2_2,m_2))")
     (("1"
       (APPLY (THEN (REPLACE -1 +) (REPLACE -2 +) (HIDE -1 -2)))
       (REPLACE -1)
       (PROPAX))
      ("2" (ASSERT))))
    ("2" (ASSERT))))
  ("2" (CASE "c_1 = Sc(SSend(i_1,c_1,k1_1,k2_1,m_1))")
   (("1" (CASE "c_2 = Sc(SSend(i_2,c_2,k1_2,k2_2,m_2))")
     (("1"
       (APPLY (THEN (REPLACE -1 +) (REPLACE -2 +) (HIDE -1 -2)))
       (REPLACE -1)
       (PROPAX))
      ("2" (ASSERT))))
    ("2" (ASSERT))))
  ("3" (CASE "k1_1 = Sk1(SSend(i_1,c_1,k1_1,k2_1,m_1))")
   (("1" (CASE "k1_2 = Sk1(SSend(i_2,c_2,k1_2,k2_2,m_2))")
     (("1"
       (APPLY (THEN (REPLACE -1 +) (REPLACE -2 +) (HIDE -1 -2)))
       (REPLACE -1)
       (PROPAX))
      ("2" (ASSERT))))
    ("2" (ASSERT))))
  ("4" (CASE "k2_1 = Sk2(SSend(i_1,c_1,k1_1,k2_1,m_1))")
   (("1" (CASE "k2_2 = Sk2(SSend(i_2,c_2,k1_2,k2_2,m_2))")
     (("1"
       (APPLY (THEN (REPLACE -1 +) (REPLACE -2 +) (HIDE -1 -2)))
       (REPLACE -1)
       (PROPAX))
      ("2" (ASSERT))))
    ("2" (ASSERT))))
  ("5" (CASE "m_1 = Sm(SSend(i_1, c_1,k1_1,k2_1,m_1))")
   (("1" (CASE "m_2 = Sm(SSend(i_2,c_2,k1_2,k2_2,m_2))")
     (("1"
       (APPLY (THEN (REPLACE -1 +) (REPLACE -2 +) (HIDE -1 -2)))
       (REPLACE -1)
       (PROPAX))
      ("2" (ASSERT))))
    ("2" (ASSERT))))))
```

**Fig. 15.** Proof of a uniqueness lemma for a `DATATYPE` constructor with five parameters.

skolem constant. The technique used in this proof can be adapted to handle the case of a constructor with more arguments. Figure 15 shows a proof of the uniqueness lemma for the constructor `SSend`:

The proof of this lemma also begins with skolemization and flattening, but this is followed by a `SPLIT` command. By executing the proof, one can see that the `SPLIT` splits the proof into subcases, one for each accessor function of `SSend`, and therefore, calling `(SPLIT)` at the third step in the shorter proof would have no effect. In each subcase of the longer proof, the pattern in the shorter proof reappears. Moreover, this pattern is now more detailed: the two individual skolem constants correspond to the variables retrieved by the accessor function, and the constructor `SSend` is applied not just to these skolem constants, but to the two sets of skolem constants corresponding to the variables in the `SSend` expressions in the hypothesis of the lemma.

We now have enough information to design a strategy. We can begin by defining a Lisp function that returns a command that follows the pattern of the subcases. Figure 16 shows the definition of such a function: `mk_adt_unique_case`, which takes as arguments the accessor function name, the two skolem constant names, and the two instantiated constructor expressions used in the pattern. We will expect to begin our strategy as the proof in Figure 15 begins: with a skolemization step, a `(FLATTEN)`, and a `(SPLIT)`. Following the

```
(defun mk_adt_unique_case (acc skconst-1 skconst-2
                              sk-expr-1 sk-expr-2)
   (let ((firstcase
           (format nil "~a~a~a~a~a~a"
            skconst-1 " = " acc "(" sk-expr-1 ")"))
         (secondcase
           (format nil "~a~a~a~a~a~a"
            skconst-2 " = " acc "(" sk-expr-2 ")")))
   `(spread (case ,firstcase)
            ((spread (case ,secondcase)
                     ((then (replace -1 +)
                            (replace -2 +)
                            (hide -1 -2)
                            (replace -1))
                      (assert)))
             (assert)))))
```

**Fig. 16.** A Lisp function that computes a command to prove a uniqueness lemma case.

(SPLIT) command, we then plan to use SPREAD to apply an appropriate subcase command to each of the subgoals.

To apply SPREAD, we we need a list of appropriate subcase commands, so we next define a Lisp function collect_adt_unique_cases that returns such a list, as follows. From the proof of the lemma SSend_unique in Figure 15, we see that there is a uniqueness case for every accessor function. Moreover, the two instantiated constructor expressions are the same for each uniqueness case, and the two skolem constants in each uniqueness case appear in these two expressions in the position corresponding to the accessor function. The function collect_adt_unique_cases, whose definition is shown in Figure 17, expects as arguments 1) the list of accessors for a DATATYPE constructor, 2) a list of skolem constant names for the quantified variables in the uniqueness lemma for the constructor, which *by convention* are arranged in the lemma formulation so that the first two correspond to the first accessor, the second two correspond to the second accessor, and so on, and 3) and 4) two constructor expressions in which the skolem constants are correctly matched with their corresponding accessor positions.

```
(defun collect_adt_unique_cases (acclist skconstlist
                                 sk-expr-1 sk-expr-2)
   (cond ((null acclist) nil)
         (t (cons
             (mk_adt_unique_case
               (car acclist)
               (car skconstlist) (cadr skconstlist)
               sk-expr-1 sk-expr-2)
             (collect_adt_unique_cases
               (cdr acclist) (cddr skconstlist)
               sk-expr-1 sk-expr-2)))))
```

**Fig. 17.** A Lisp function that computes a list of uniqueness-case commands.

Note that to work correctly when it is applied, collect_adt_unique_cases must be given the appropriate arguments. Appropriate arguments can be computed from the formula in the lemma being proved. To compute the constructor expression instances corresponding the list of skolem constants, we need to know the names of the skolem constants. A convenient way to do this is to compute special skolem constant names from the list of bound variables in the lemma. Once the prover is invoked on the lemma, this can be done by using the Lisp function get_binding_names (see Figure 18) to probe the proof state for the names of

```
(defun get_binding_names (sform)
   (mapcar 'id (bindings (formula sform))))
(defun mk_adt_unique_skolem_names (varlis)
   (mapcar #'(lambda (varname)
                (concatenate 'string (string varname) "_uniq"))
           varlis))
(defun get_sk_constructor_exprs (sform)
   (exprs (argument (car (exprs (argument (formula sform)))))))
```

**Fig. 18.** Three auxiliary functions used in datatype_unique_strat.

the bound variables, and then applying the Lisp function mk_adt_unique_skolem_names to transform this list into a list of skolem names for the bound variables. The two constructor expressions are found by again probing the proof state, this time using the function get_sk_constructor_exprs.

Finally, we can define the proof rule adt_unique_strat, using the defstep macro, as shown in Figure 19. Note that both adt_unique_strat and its auxiliary rule adt_unique_strat_continue begin with a probe of the proof state *ps* to retrieve a value sform representing the current proof goal. The expected proof goal for adt_unique_strat corresponds to a uniqueness lemma. The initial call to (ASSERT) in adt_unique_strat assures that PVS has filled in all the fields in the CLOS structure for this goal, rather than lazily leaving them unbound. Both proof steps use the technique of first computing and then applying a command.

```
(defstep adt_unique_strat ()
   (then
     (assert)
     (let ((sform (car (s-forms (current-goal *ps*))))
           (bind-names (get_binding_names sform))
           (uniq-sk-names
             (mk_adt_unique_skolem_names bind-names))
           (cmd
             `(then (skolem 1 ,uniq-sk-names)
                    (adt_unique_strat_continue ,uniq-sk-names))))
     cmd))
   "" "")

(defstep adt_unique_strat_continue (sk-name-list)
   (let ((sform (car (s-forms (current-goal *ps*))))
         (sk-constr-exprs (get_sk_constructor_exprs sform))
         (sk-constr-expr-1 (car sk-constr-exprs))
         (sk-constr-expr-2 (cadr sk-constr-exprs))
         (constr-name (id (operator sk-constr-expr-1)))
         (all-constrs
           (constructors
             (adt (adt-type (operator sk-constr-expr-1))))))
   (let ((constr-form (car
          (select #'(lambda (x) (eq (id x) constr-name))
                  all-constrs)))
         (accessors (mapcar 'id (acc-decls constr-form)))
         (cases
           (collect_adt_unique_cases accessors sk-name-list
                                     sk-constr-expr-1
                                     sk-constr-expr-2))
         (cmd `(then (flatten) (spread (split) ,cases))))
   cmd))
   "" "")
```

**Fig. 19.** Defining a new proof rule adt_unique_strat.

The effect of the part of adt_unique_strat up to the point where it calls adt_unique_strat_continue is to skolemize the formula in the lemma using the skolem constants computed by mk_adt_unique_skolem_names. Thus, the value sform computed at the beginning of adt_unique_strat_continue corresponds to the skolemized version of the uniqueness lemma. Moreover, adt_unique_strat_continue is passed the list of skolem names as an argument so that it need not be recomputed. The step adt_unique_strat_continue proceeds by first computing the arguments it needs to pass to the function collect_adt_unique_cases, and uses the result of applying this function to the arguments in its computation of a proof command in the form of a strategy, which it then applies.

## 5 Discussion

Chapter 5 of the PVS Prover Guide [10, 11] contains much information useful to users who wish to write their own strategies. This information includes a description of global variables used in the prover, the CLOS slots in a proof state, methods for retrieving formulae and recognizing the class of an expression, several useful PVS functions including args1, args2, and gather-fnums, and the macros defstep, defhelper, and defstrat for defining new rules and strategies.

Several things could provide additional help for writing user strategies in PVS. One is simply easily accessible documentation of additional useful PVS functions and macros. Documentation of the helper functions used in the standard PVS strategies would eliminate duplication of effort on the part of PVS users who write their own strategies.

Currently, the CLOS structure must be probed to determine how to retrieve many details of the information on the proof state. Explicit documentation of this structure could allow this "probing" to be done off-line.

Strategies that explicitly reference the CLOS structure used for the internal representation of the PVS proof state must rely on the stability of this internal representation. An extra layer of "retrieval" functions whose names and effects would remain the same despite any changes in the internal representation of the proof state is one possibility for reducing the sensitivity of user strategies to any changes in the PVS implementation.

Even without these extra aids, however, it is possible for users to develop sophisticated strategies to serve their special needs—and to share with others.

## Acknowledgement

Many thanks to N. Shankar for clarifying the semantics of TRY and providing several examples of its use. We also thank both N. Shankar and S. Owre for helpful comments.

## References

1. Myla Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):139–181, 2000. Published Feb., 2001.
2. Myla Archer. Proving correctness of the basic TESLA multicast stream authentication protocol with TAME. In *Informal Proceedings of the Workshop on Issues in the Theory of Security (WITS'02)*, Portland, OR, Jan. 14–15 2002.
3. Myla Archer, Constance Heitmeyer, and Elvinia Riccobene. Proving invariants of I/O automata with TAME. *Automated Software Engineering*, 9(3):201–232, 2002.
4. M. Devillers, D. Griffioen, J. Romijn, and F. Vaandrager. Verification of a leader election protocol—formal methods applied to ieee 1394. *Formal Methods in System Design*, 16(3):307–320, June 2000.
5. B. Di Vito. A PVS prover strategy package for common manipulations. Technical Memorandum NASA/TM-2002-211647, NASA Langley Research Center, Hampton, VA, April 2002.
6. C. Muñoz and M. Mayero. Real automation in the field. Technical Report Interim ICASE Report No. 39, NASA/CR-2001-211271, ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681-2199, USA, December 2001.

7. S. Owre and N. Shankar. *PVS API Reference*. SRI Computer Science Laboratory, Menlo Park, California, USA, September 2003.
8. Adrian Perrig, Ran Canetti, J. D. Tygar, and Dawn Song. Efficient authentication and signing of multicast streams over lossy channels. In *Proc. of IEEE Security and Privacy Symposium (S&P2000)*, pages 56–73, May 2000.
9. John Rushby. Personal communication. 2003.
10. N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. The PVS prover guide. Technical report, Computer Science Lab., SRI Intl., Menlo Park, CA, 1998.
11. N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS Prover Guide, Version 2.4. Technical report, Computer Science Lab., SRI Intl., Menlo Park, CA, November 2001.
12. Guy L. Steele. *COMMON LISP: the Language*. Digital Press, Bedford, MA, 1990. Second edition.