PVS Prover Guide

Version 2.4 • November 2001

N. Shankar
S. Owre
J. M. Rushby
D. W. J. Stringer-Calvert
{Owre,Shankar,Rushby,Dave_SC}@csl.sri.com
http://pvs.csl.sri.com/

SRI International
Computer Science Laboratory • 333 Ravenswood Avenue • Menlo Park CA 94025

The initial development of PVS was funded by SRI International. Subsequent enhancements were partially funded by SRI and by NASA Contracts NAS1-18969 and NAS1-20334, NRL Contract N00014-96-C-2106, NSF Grants CCR-9300044, CCR-9509931, and CCR-9712383, AFOSR contract F49620-95-C0044, and DARPA Orders E276, A721, D431, D855, and E301.

Contents

Chapter 1

Introduction

PVS stands for "Prototype Verification System," and as the name suggests, it is a prototype environment for specification and verification. This document is a reference manual for the commands employed in constructing proofs using the PVS proof checker. The *PVS System Guide* [11] should be consulted for information on how to use the system to develop specifications and proofs. The *PVS Language Reference* [10] describes the language of PVS.

The primary purpose of PVS is to provide formal support for conceptualization and debugging in the early stages of the lifecycle of the design of a hardware or software system. In these stages, both the requirements and designs are expressed in abstract terms that are not necessarily executable. We find that the best way to analyze such an abstract specifications is by attempting proofs of desirable consequences of the specification. Our own experience with PVS in this regard has been that such attempted proofs of *putative theorems* very quickly highlight even subtle errors and infelicities. These would be costly to detect and correct at later stages of the design lifecycle.

The primary emphasis in the PVS proof checker is on supporting the construction of readable proofs. The automation underlying PVS serves to ensure that the process of verification yields human insights that can be easily communicated to other humans, and encapsulated for future verifications. PVS therefore pays a lot of attention to simplifying the process of developing, debugging, maintaining, and presenting proofs. In order to make proofs easier to develop, the PVS proof checker provides a collection of powerful proof commands to carry out propositional, equality, and arithmetic reasoning with the use of definitions and lemmas. These proof commands can be combined to form *proof strategies*. To make proofs easier to debug, the PVS proof checker permits proof steps to be undone, and it also allows the specification to be modified during the course of a proof. To support proof maintenance, PVS allows proofs (and partial proofs) to be edited and rerun. Currently, the proofs generated by PVS can be made presentable but they still fall short of being humanly readable. The readability of proofs will be one focus of future enhancements to PVS.

PVS is meant to provide effective theorem proving support for a richly expressive specification language. This is unusual because systems that support expressive specification languages usually only have modest deductive support and, conversely, the highly mechanized theorem provers are generally associated with fairly restricted specification notations. For example, the powerful Boyer-Moore theorem prover [4] uses a constructive, untyped, quantifier-free first order logic as its specification language. On the other hand, "proof assistants" for Z and VDM (such as *Mural* [1]) provide attractive mouse-and-windows interfaces for natural deduction but lack the machinery needed to automate the tedious aspects of proof construction. The combination of an expressive logic and a powerful theorem proving capability in PVS hinges on a careful integration between the typechecker and the proof checker. The typechecker exploits the deductive power of the proof checker to automatically discharge proof obligations generated by the typechecker. These proof obligations, termed *type correctness conditions* (or TCCs), arise for instance, when a term is typechecked against an expected predicate subtype. Such proof obligations can also arise as subgoals during proof checking since the typechecker is frequently invoked to check user-supplied expressions and quantifier instantiations.

PVS' combination of direct control by the user for the higher levels of proof development, and powerful automation for the lower levels, is also somewhat unusual. On the whole, PVS provides more automation than a low-level proof checker (such as LCF [7], HOL [8], Nuprl [5], Automath [6]), and more control than a highly automatic theorem prover (such as Otter [9] or Nqthm [3, 4]). Compared with thoroughly automated theorem provers such as Nqthm, PVS's deductive component may be considered a proof checker—but it seems like a theorem prover to those accustomed to systems such as HOL which provide limited automation. We reflect this ambivalence by sometime referring to PVS as a theorem prover, and sometimes as a proof checker. The PVS proof checker is somewhat in the spirit to the IMPLY prover of Bledsoe and his colleagues [2].

While there are clearly many avenues for further improvement of the PVS system, the combination of a highly expressive specification language and a powerful interactive proof checking capability already yields a productive verification environment. There are a number of examples, both big and small, that support this observation. A list of applications of PVS and a bibliography of PVS related reports and papers is maintained at the PVS web site at `http://pvs.csl.sri.com/`.

1.1 PVS Proof Display and Construction

We give a brief overview of the sequent-style proof representation used in PVS since this is needed to understand the effect of the PVS proof commands. The PVS proof checker is interactive, but also supports a batch mode in which proofs can be easily rerun. The prover maintains a *proof tree*, and it is the goal of the user to construct a proof tree which is complete, in the sense that all of the leaves are recognized as true. Each node of the proof tree is a *proof goal* that follows from its offspring nodes by means of a proof step. Each proof goal is a *sequent* consisting of a sequence of formulas called *antecedents* and a sequence of formulas called *consequents*. In PVS, such a sequent is displayed as

$$
\begin{array}{ll}
\texttt{\{-1\}} & A_1 \\
\texttt{\{-2\}} & A_2 \\
\texttt{[-3]} & A_3 \\
 & \vdots \\
\hline
\texttt{\{1\}} & B_1 \\
\texttt{[2]} & B_2 \\
\texttt{\{3\}} & B_3 \\
 & \vdots
\end{array}
$$

where the A_i and B_j are PVS formulas collectively referred to as *sequent formulas*: the A_i are the antecedents and the B_j are the consequents; the row of dashes serves to separate the antecedents from the consequents.[1] The sequence of antecedents or consequents (but not both) may be empty. The intuitive interpretation of a sequent is that the conjunction of the antecedents implies the disjunction of the consequents, *i.e.*, $(A_1 \wedge A_2 \wedge A_3 \ldots) \supset (B_1 \vee B_2 \vee B_3 \ldots)$. The proof tree starts off with a root node of the form $\vdash A$, where A is the theorem to be proved. PVS proof steps build a proof tree by adding subtrees to leaf nodes as directed by the proof commands. It is easy to see that a sequent is *true* if any antecedent is the same as any consequent, if any antecedent is *false*, or if any consequent is *true*. Other sequents can also be recognized as *true*, using more powerful inferences that will be described later. Once a sequent is recognized as *true*, that branch of the proof tree is terminated. The goal is to build a proof tree whose branches have all been terminated in this way.

At any time in a PVS proof, attention is focussed on some sequent that is a leaf node in the current proof tree—this is the sequent that is displayed by the PVS prover while awaiting the user's command. The numbers in brackets, *e.g.*, `[-3]`, and braces, *e.g.*, {3}, before each formula in the displayed sequent are used to name the corresponding formulas. The formula numbers in square brackets (*e.g.*, `[-3]` above) indicate formulas that are unchanged in a subgoal from the parent goal whereas the numbers in braces (*e.g.*, {2} in the example above), serve to highlight those formulas that are either new or different from those of the parent sequent.

PVS *interactive commands* allow the user to shift the focus (using the `postpone` command) to a sibling of the current sequent (if any), or to abandon (using the `fail` or `undo` command) a portion of the proof containing the current sequent in order to

[1]In written text, sequents may also be written as $A_1, A_2, A_3, \ldots \vdash B_1, B_2, B_3, \ldots$

return to some ancestor node representing an earlier point in the proof. PVS *proof steps* cause a subtree of sequents to "grow" from the current sequent, and shift the focus to one of the leaves of the new subtree. For example, one proof step (called `split` in PVS) takes a sequent of the form

$$\Gamma \vdash A \wedge B$$

(where Γ is any sequence of formulas) and creates the pair of child sequents

$$\Gamma \vdash A \quad \text{and} \quad \Gamma \vdash B$$

(*i.e.*, in order to prove a conjunction, it is sufficient to prove each of the conjuncts separately).

A PVS *proof command* when applied to a sequent provides the means to construct proof trees. These commands can be used to introduce lemmas, expand definitions, apply decision procedures, eliminate quantifiers, and so on; they affect the proof tree, and are saved when the proof is saved. Proof commands may be invoked directly by the user, or as the result of executing a strategy. We refer to the action resulting from a proof command as a *proof step* or a *proof rule* and often use these terms interchangeably.

The proof commands that really define the PVS logic are called the *primitive rules*; they either recognize the current sequent as true and terminate that branch of the proof tree, or they add one or more child nodes to the current sequent and transfer the focus to one of these children. PVS *strategies* are combinations of proof-steps that can, in principle, add a subtree of any depth to the current node (*i.e.*, the step may invoke substeps and so on). On the other hand, those proof steps called *defined rules* (which can be the result of invoking strategies with the `apply` control strategy) silently prune those branches of the subtrees which they generate that are recognized as true, and collapse all remaining interior nodes, so that the subtree actually generated has depth zero (*i.e.*, the sequent is recognized as true and this branch of the proof terminates) or one (*i.e.*, it simply adds children to the current node).

As mentioned earlier, some of the individual proof steps in the PVS prover are extremely sophisticated and make heavy use of arithmetic and equality decision procedures. Various properties of function, record, and tuple types, and abstract datatypes, are also built into the operation of the PVS prover. The interplay between type information (from the specification) and inference is also mechanized in a significant way by PVS. For example, if the function definition

$$\mathit{factorial}(n)\text{: } \textbf{recursive}\ \mathit{nat} = \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times \mathit{factorial}(n-1)\ \textbf{endif}$$

is used to expand the term $\mathit{factorial}(i+1)$, where i is of type nat, the PVS prover will retrieve the type predicate for nat, namely

$$(\lambda n\colon n \geq 0),$$

instantiate it with i and call the arithmetic decision procedures to deduce that $i+1 \neq 0$, and thereby select just the relevant branch of the definition to produce the result

$$(i+1) \times \mathit{factorial}(i+1-1).$$

Though there are only a few proof commands in PVS, many of these commands are quite powerful and flexible. It is wise to experiment with all of the commands in order to more thoroughly understand how they work, and to employ the more powerful commands whenever possible. As with any automated reasoning system, the form of the specification can significantly affect the ease or difficulty of the accompanying proofs. The specifier must demonstrate good taste in writing abstract specifications, using definitions to name useful concepts, employing types, subtypes, and abstract datatypes appropriately, and in stating lemmas in their most useful forms. In a system like PVS, it is quite easy to carry out a less than elegant proof; the user must exercise enough discipline to structure proofs so that they are less cluttered, easy to read, and can be robustly rerun in the face of minor changes. It is also important to introduce useful lemmas as they arise in the proof, and define strategies to encapsulate patterns of proof steps.

The remainder of this chapter summarizes how interactive PVS proof attempts are initiated and terminated. Chapter 2 contains a small example proof to illustrate how the PVS proof checker is used. Chapter 3 gives a brief overview of the logical underpinnings of PVS. Chapter 4 describes the syntax of the proof commands. Chapter 5 is a guide to the PVS proof strategy language and also gives examples of proof strategies and derived inferences rules.

1.2 Interaction Basics

The following paragraphs summarizes how proof attempts are initiated, abandoned, or interrupted, and how help information can be obtained. Full details are presented in the PVS user guide [11].

Initiating a Proof Attempt A proof session is initiated from within an Emacs buffer containing a PVS specification by using the Emacs command `M-x pr` with the cursor at the formula that is to be proved. If the formula has already been proved, the user is asked whether the proof attempt should proceed. If a proof or partial proof for the relevant formula already exists, then the user is asked if this proof should be rerun. The Emacs command `M-x xpr` can be used to initiate a proof that generates a Tcl/Tk display of the proof structure as it is being developed.

Exiting a Proof Attempt In the proof sessions shown below, all user input is displayed in boldface. The proof commands follow the `Rule?` prompt and the remaining text is generated by PVS. A proof attempt can be abandoned by typing `q` or `quit` at the prompt. At the end of a successful or abandoned proof attempt, the user is queried as to whether the resulting partial proof should be saved. The timing characteristics are displayed at the end. The saved partial proof can be rerun in a subsequent attempt so that the unfinished parts of the proof can be completed.

Getting Help There are several ways of getting helpful information about the proof checking commands. The easiest way is to invoke `M-x help-pvs-prover` or `M-x x-prover-commands`. See page 25 for more details.

Interrupting Proofs The proof checker can be interrupted when it is working on a command by typing `C-c C-c`. This places the system at a Lisp break. Typing `(restore)` at the break returns the system to the `Rule?` prompt corresponding to the last interaction.

Chapter 2

An Example Proof

We consider a simple proof using induction to show that when given two functions `f` and `g` on the natural numbers, the sum of the first `n` values of `f` and `g` is the same as the sum of the first `n` values of the function `(LAMBDA n: f(n) + g(n))`. The theory `sum` below defines the summation operator `sum` and states the desired theorem as `sum_plus`.

In the first proof attempt described below, `sum_plus` is proved using simple, low-level inference steps, and in the second proof attempt, the same theorem is proved by invoking a single high-level proof strategy.

Once the proof is initiated[1], the main goal is displayed in the `*pvs*` buffer followed by a `Rule?` prompt. The user commands are typed in at this prompt. The first command, `skolem!`, introduces Skolem constants `f!1`, `g!1`, and `n!1` for the universally quantified variables in the theorem. The second command, `lemma`, introduces the induction scheme for natural numbers `nat_induction` as an antecedent formula. This induction scheme is proved as a lemma in the theory `naturalnumbers` in the PVS prelude.[2]

[1]See the previous chapter or the System Guide [11] for details on how to initiate proof attempts.

[2]The lemma name can also be given in its full form `naturalnumbers.nat_induction` if the theory name is needed for disambiguation. See the PVS language manual [10] for more details of how name resolution is performed.

```
sum: THEORY
  BEGIN

   n: VAR nat
   f, g: VAR [nat -> nat]

   sum(f, n): RECURSIVE nat =
  (IF n = 0
      THEN 0
      ELSE f(n-1) + sum(f, n - 1)
   ENDIF)
  MEASURE n

 sum_plus: LEMMA
    sum((lambda n: f(n) + g(n)), n)
   = sum(f, n) + sum(g, n)

  square(n): nat = n*n

  sum_of_squares: LEMMA
    6 * sum(square, n+1) = n * (n + 1) * (2*n + 1)

  cube(n): nat = n*n*n

  sum_of_cubes: LEMMA
    4 * sum(cube, n+1) = n*n*(n+1)*(n+1)

 END sum
```

Figure 2.1: `sum`

```
sum_plus :

  |-------
{1}   (FORALL (f: [nat -> nat], g: [nat -> nat], n: nat):
         sum((LAMBDA (n: nat): f(n) + g(n)), n) = sum(f, n) + sum(g, n))

Rule? (skolem!)
Skolemizing,
this simplifies to:
sum_plus :

  |-------
{1}   sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n!1)
          = sum(f!1, n!1) + sum(g!1, n!1)

Rule? (lemma "nat_induction")
Applying nat_induction where
this simplifies to:
sum_plus :

{-1}   (FORALL (p: pred[nat]):
         (p(0) AND (FORALL (j: nat): p(j) IMPLIES p(j + 1)))
             IMPLIES (FORALL (i: nat): p(i)))
  |-------
[1]   sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n!1)
          = sum(f!1, n!1) + sum(g!1, n!1)
```

The next step is to instantiate the induction scheme with a suitable induction predicate. This instantiation is supplied manually using the `inst` command.

```
Rule? (inst - "(LAMBDA n: sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n)
          = sum(f!1, n) + sum(g!1, n))")
Instantiating the top quantifier in - with the terms:
 (LAMBDA n: sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n)
          = sum(f!1, n) + sum(g!1, n)),
this simplifies to:
```

The effect of `inst` command is to generate a subgoal where the universally quantified variable `p` has been replaced by the given induction predicate.

```
sum_plus :

{-1}   ((LAMBDA n:
          sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n)
              = sum(f!1, n) + sum(g!1, n))(0)
           AND
         (FORALL (j: nat):
            (LAMBDA n:
               sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n)
                   = sum(f!1, n) + sum(g!1, n))(j)
                IMPLIES
              (LAMBDA n:
                 sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n)
                     = sum(f!1, n) + sum(g!1, n))(j
                                                       + 1)))
          IMPLIES
        (FORALL (i: nat):
           (LAMBDA n:
              sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n)
                  = sum(f!1, n) + sum(g!1, n))(i))
  |-------
[1]   sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n!1)
          = sum(f!1, n!1) + sum(g!1, n!1)
```

The resulting instantiated induction scheme now contains several redexes that are beta-reduced by means of the beta command as shown below.

```
Rule? (beta)
Applying beta-reduction,
this simplifies to:
sum_plus :

{-1}   (sum((LAMBDA (n: nat): f!1(n) + g!1(n)), 0) = sum(f!1, 0) + sum(g!1, 0)
           AND
         (FORALL (j: nat):
            sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j)
                = sum(f!1, j) + sum(g!1, j)
                IMPLIES sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j + 1)
                  = sum(f!1, j + 1) + sum(g!1, j + 1)))
          IMPLIES
        (FORALL (i: nat):
           sum((LAMBDA (n: nat): f!1(n) + g!1(n)), i)
               = sum(f!1, i) + sum(g!1, i))
  |-------
[1]   sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n!1)
          = sum(f!1, n!1) + sum(g!1, n!1)
```

Apply the conjunctive splitting command `split` to the goal yields three subgoals. The first goal is to demonstrate that the conclusion of the instantiated induction scheme implies the original conjecture following the introduction of Skolem constants. The second subgoal is the base case, and the third subgoal is the induction step. The first subgoal is easily proved by using the heuristic instantiation command `inst?`.

```
Rule? (split)
Splitting conjunctions,
this yields  3 subgoals:
sum_plus.1 :

{-1}   (FORALL (i: nat):
         sum((LAMBDA (n: nat): f!1(n) + g!1(n)), i) = sum(f!1, i) + sum(g!1, i))
  |-------
[1]   sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n!1)
          = sum(f!1, n!1) + sum(g!1, n!1)

Rule? (inst?)
Found substitution:
i gets n!1,
Using template: sum((LAMBDA (n: nat): f!1(n) + g!1(n)), i) =
                 sum(f!1, i) + sum(g!1, i)
Instantiating quantified variables,

This completes the proof of sum_plus.1.
```

The second subgoal, the base case, contains an irrelevant formula numbered `2` which was only needed for the first subgoal proved above. This formula can be suppressed with the `hide` command. The hidden formulas can be examined using the Emacs command `M-x show-hidden-formulas`, and revealed or reintroduced into the sequent using the `reveal` command.

```
sum_plus.2 :

  |-------
{1}   sum((LAMBDA (n: nat): f!1(n) + g!1(n)), 0) = sum(f!1, 0) + sum(g!1, 0)
[2]   sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n!1)
          = sum(f!1, n!1) + sum(g!1, n!1)

Rule? (hide 2)
Hiding formulas:  2,
this simplifies to:
sum_plus.2 :

  |-------
[1]   sum((LAMBDA (n: nat): f!1(n) + g!1(n)), 0) = sum(f!1, 0) + sum(g!1, 0)
```

We are then left with the formula numbered `1` which is easily proved by expanding the definition of `sum` using the `expand` command. Notice that this command uses the PVS decision procedures to simplify the definition of `sum` and to reduce the equality to `TRUE`.

```
Rule? (expand "sum")
Expanding the definition of sum,
this simplifies to:
sum_plus.2 :

  |-------
{1}   TRUE

which is trivially true.
```

The remaining subgoal is the induction step. It too contains the irrelevant formula numbered 2 that is again suppressed using the `hide` command.

```
sum_plus.3 :

  |-------
{1}   (FORALL (j: nat):
         sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j) = sum(f!1, j) + sum(g!1, j)
             IMPLIES sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j + 1)
               = sum(f!1, j + 1) + sum(g!1, j + 1))
[2]   sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n!1)
          = sum(f!1, n!1) + sum(g!1, n!1)

Rule? (hide 2)
Hiding formulas:  2,
this simplifies to:
sum_plus.3 :

  |-------
[1]   (FORALL (j: nat):
         sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j) = sum(f!1, j) + sum(g!1, j)
             IMPLIES sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j + 1)
               = sum(f!1, j + 1) + sum(g!1, j + 1))
```

Applying the `skosimp` command, which is a compound of the `skolem!` and `flatten` commands, the resulting simplified sequent contains an antecedent formula, the induction hypothesis, and a consequent formula, the induction conclusion.

```
Rule? (skosimp)
Skolemizing and flattening,
this simplifies to:
sum_plus.3 :

{-1}   sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j!1)
          = sum(f!1, j!1) + sum(g!1, j!1)
  |-------
{1}   sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j!1 + 1)
          = sum(f!1, j!1 + 1) + sum(g!1, j!1 + 1)
```

If we apply the `expand` command selectively to expand those occurrences of `sum` on the consequent side, we get a sequent that is tautologously true. Notice, once again, that the expand command makes significant use of type information within the PVS decision procedures in order to simplify not only the expanded definition of `sum` but also the resulting equality between arithmetic expressions.

```
Rule? (expand "sum" +)
Expanding the definition of sum,
this simplifies to:
sum_plus.3 :

[-1]   sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j!1)
          = sum(f!1, j!1) + sum(g!1, j!1)
  |-------
{1}   sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j!1)
          = sum(f!1, j!1) + sum(g!1, j!1)

which is trivially true.

This completes the proof of sum_plus.3.

Q.E.D.


Run time  = 0.79 secs.
Real time = 38.10 secs.
```

This successfully completes the proof attempt. The CPU time and wallclock time for the proof attempt are displayed above.

2.1 The Example Proof Redone

PVS has a language in which proof strategies can be written. The essence of the above proof can actually be captured as a strategy. The strategy `induct-and-rewrite!` invokes induction according to the scheme appropriate to the given induction variable and then completes the proof by expanding the functions used in the theorem and applying heuristic instantiation and the decision procedures.

```
sum_plus :

  |-------
{1}   (FORALL (f: [nat -> nat], g: [nat -> nat], n: nat):
         sum((LAMBDA (n: nat): f(n) + g(n)), n) = sum(f, n) + sum(g, n))

Rule? (induct-and-rewrite! "n")
sum rewrites sum((LAMBDA (n: nat): f!1(n) + g!1(n)), 0)
  to 0
sum rewrites sum(f!1, 0)
  to 0
sum rewrites sum(g!1, 0)
  to 0
sum rewrites sum((LAMBDA (n: nat): f!1(n) + g!1(n)), 1 + j!1)
  to f!1(j!1) + g!1(j!1) + sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j!1)
sum rewrites sum(f!1, 1 + j!1)
  to f!1(j!1) + sum(f!1, j!1)
sum rewrites sum(g!1, 1 + j!1)
  to g!1(j!1) + sum(g!1, j!1)
By induction on n and rewriting,
Q.E.D.


Run time  = 0.85 secs.
Real time = 6.47 secs.
```

Such high-level strategies might not always succeed. When a strategy fails to complete a proof, it is possible to continue proving the resulting subgoals interactively using further proof commands, or to backtrack (using `undo`) in order to try alternative proof commands. When a strategy is invoked with a `$` suffix, e.g., `induct-and-rewrite!$`, the strategy is executed so that the expanded internal steps are visible. This mode of strategy invocation provides more information and is useful when debugging.

The Emacs command `M-x show-last-proof` can be used to get a summary of the most recently completed proof. Note that the Emacs commands `M-x add-declaration` and `M-x modify-declaration` can be used to alter the specification even while a proof is progress. Since these commands can affect the validity of a proof so proofs should always be rerun in order to check their validity. During the course of a proof, the Emacs command `M-x ancestry` displays the sequence of goals leading back to the root goal of the proof, `M-x siblings` displays the sibling subgoals of the current goal, `M-x show-hidden` displays the hidden formulas of the current sequent, and `M-x show-auto-rewrites` displays those rewrite rules that are automatically applied.

Chapter 3

The Logic of PVS

While using the PVS proof checker, it is useful to be aware of the rules underlying the PVS logic. The proof rules presented here form the theoretical basis for PVS but are not the ones that are directly implemented in the system. There is, of course, a great deal more to building an effective proof checker than merely codifying the proof rules.

The following sections present the notation used throughout this document, then the logical rules. Note that to be complete we should include the type rules; these will be included in a later version of this document.

3.1 Notation

PVS employs a sequent calculus. We have already introduced the notions of *sequent*, *antecedent*, and *consequent*. In the following, we will use the Greek letters Γ and Δ to represent (finite) sequences of formulas, and latin letters A, B, and C to represent individual formulas. As usual, these can have indices. *Inference rules* are of the form

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \cdots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}\ \mathbf{R}.$$

This says that if we are given a leaf of a proof tree of the form $\Gamma \vdash \Delta$, then by applying the rule named $\mathbf{R}$, we may obtain a tree with n new leaves.

In the following, we will be using the usual logical notation for the connectives, quantifiers, etc. The following table relates them to those used in PVS.

$\neg$	`NOT`
$\wedge$	`AND, &`
$\vee$	`OR`
$\supset$	`IMPLIES, =>`
$\Longleftrightarrow$	`IFF, <=>`
$\forall$	`FORALL`
$\exists$	`EXISTS`
λ	`LAMBDA`

Note that an expression of the form A `WHEN` B is equivalent to $B \supset A$, and hence such expressions will not be explicitly mentioned in the rules. A PVS `IF` expression of the form

`IF` A `THEN` B `ELSIF` C `THEN` D `ELSE` E `ENDIF`

will be abbreviated below as $\mathtt{IF}(A, B, \mathtt{IF}(C, D, E))$

3.2 The Structural Rules

The structural rules permit the sequent to be rearranged or weakened via the introduction of new sequent formulas into the conclusion. All of the structural rules can be expressed in terms of the single powerful weakening rule shown below. It allows a weaker statement to be derived from a stronger one by adding either antecedent formulas or consequent formulas. The relation $\Gamma_1 \subseteq \Gamma_2$ holds between two lists when all the formulas in Γ_1 occur in the list Γ_2.

$$\frac{\Gamma_1 \vdash \Delta_1}{\Gamma_2 \vdash \Delta_2}\,\mathbf{W} \qquad \text{if } \Gamma_1 \subseteq \Gamma_2 \text{ and } \Delta_1 \subseteq \Delta_2$$

Both the Contraction and Exchange rules shown below are absorbed by the above *Weakening* rule. The *Contraction* rule allows multiple occurrences of the same sequent formula to be replaced by a single occurrence.

$$\frac{A, A, \Gamma \vdash \Delta}{A, \Gamma \vdash \Delta}\,\mathbf{C}\vdash \qquad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\vdash\mathbf{C}$$

The *Exchange* rule asserts that the order of the formulas in the antecedent and the consequent parts of the sequent is immaterial. It can be stated as

$$\frac{\Gamma_1, B, A, \Gamma_2 \vdash \Delta}{\Gamma_1, A, B, \Gamma_2 \vdash \Delta}\,\mathbf{X}\vdash \qquad \frac{\Gamma \vdash \Delta_1, B, A, \Delta_2}{\Gamma \vdash \Delta_1, A, B, \Delta_2}\vdash\mathbf{X}$$

3.3 The Propositional Rules

The rules about conjunction, disjunction, implication, and negation are quite straightforward. The propositional axiom rule requires the notion of two formulas A and B being syntactically equivalent modulo the renaming of bound variables. Thus, the syntactic equivalence: $(\forall x, y\colon (\lambda z\colon f(y,z))(x) < y) \equiv (\forall z, x\colon (\lambda y\colon f(x,y))(z) < x)$, holds. The Propositional Axiom rule is then given as:

$$\frac{}{\Gamma, A \vdash B, \Delta}\ \textbf{Ax} \qquad \text{where } A \equiv B$$

The Cut rule can be seen as a mechanism for introducing a case-split into a proof of a sequent $\Gamma \vdash \Delta$ to yield the subgoals $\Gamma, A \vdash \Delta$ and $\Gamma \vdash A, \Delta$, which can be seen as assuming A along one branch and $\neg A$ along the other.

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma \vdash A, \Delta}{\Gamma \vdash \Delta}\ \textbf{Cut}$$

There are two rules for each of the propositional connectives of conjunction ($\wedge$), disjunction ($\vee$), implication ($\supset$), and negation ($\neg$), corresponding to the antecedent and consequent occurrences of these connectives.

$$\frac{A, B, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta}\ \wedge\vdash \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}\ \vdash\wedge$$

$$\frac{A, \Gamma \vdash \Delta \qquad B, \Gamma \vdash \Delta}{A \vee B, \Gamma \vdash \Delta}\ \vee\vdash \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}\ \vdash\vee$$

$$\frac{B, \Gamma \vdash \Delta \qquad \Gamma \vdash A, \Delta}{A \supset B, \Gamma \vdash \Delta}\ \supset\vdash \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \supset B, \Delta}\ \vdash\supset$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}\ \neg\vdash \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}\ \vdash\neg$$

3.4 The Equality Rules

The rules for equality can be stated as below. The rules of transitivity and symmetry for equality can be derived from these rules. The notation $A[e]$ is used to highlight one or more occurrences of e in the formula A. The notation $\Delta[e]$ similarly highlights occurrences of e in Δ.

$$\frac{}{\Gamma \vdash a = b, \Delta}\ \textbf{Refl} \quad \text{if } a \equiv b \qquad \frac{a = b, \Gamma[b] \vdash \Delta[b]}{a = b, \Gamma[a] \vdash \Delta[a]}\ \textbf{Repl}$$

3.5 The Quantifier Rules

The quantifier rules are stated below. The notation $A\{x \leftarrow t\}$ represents the result of substituting the term t for all the free occurrences of x in A with the possible renaming of bound variables in A to avoid capturing any free variables in t. In the $\vdash \forall$ and $\exists \vdash$ rules, a must be a new constant that does not occur in the conclusion sequent.

$$\frac{\Gamma, A\{x \leftarrow t\} \vdash \Delta}{\Gamma, (\forall x\colon A) \vdash \Delta}\ \forall\vdash \qquad \frac{\Gamma \vdash A\{x \leftarrow a\}, \Delta}{\Gamma \vdash (\forall x\colon A), \Delta}\ \vdash\forall$$

$$\frac{\Gamma, A\{x \leftarrow a\} \vdash \Delta}{\Gamma, (\exists x\colon A) \vdash \Delta}\ \exists\vdash \qquad \frac{\Gamma \vdash A\{x \leftarrow t\}, \Delta}{\Gamma \vdash (\exists x\colon A), \Delta}\ \vdash\exists$$

3.6 Rules for `IF`

It is extremely useful to have the branching operation `IF` in the language for expressing conditional expressions. For each type α, there is an `IF` operation with the signature $[\texttt{bool}, \alpha, \alpha \to \alpha]$. The transformation of $A[e]$ to $A[b]$ represents the replacement of the highlighted occurrences of e in A by b. Note that for the $\texttt{IF}\uparrow\vdash$ and $\vdash\ \texttt{IF}\uparrow$ rules, the A in $B[\texttt{IF}(A, b, c)]$ must not contain any free variable occurrences that are bound in $B[\texttt{IF}(A, b, c)]$. The inference rules for `IF` are:

$$\frac{\Gamma, \texttt{IF}(A, B[b], B[c]) \vdash \Delta}{\Gamma, B[\texttt{IF}(A, b, c)] \vdash \Delta}\ \texttt{IF}\uparrow\vdash \qquad \frac{\Gamma \vdash \texttt{IF}(A, B[b], B[c]), \Delta}{\Gamma \vdash B[\texttt{IF}(A, b, c)], \Delta}\ \vdash\texttt{IF}\uparrow$$

$$\frac{\Gamma, A, B \vdash \Delta \qquad \Gamma, \neg A, C \vdash \Delta}{\Gamma, \texttt{IF}(A, B, C) \vdash \Delta}\ \texttt{IF}\vdash \qquad \frac{\Gamma, A \vdash B, \Delta \qquad \Gamma, \neg A \vdash C, \Delta}{\Gamma \vdash \texttt{IF}(A, B, C), \Delta}\ \vdash\texttt{IF}$$

The PVS proof checker is founded on the sequent calculus rules described above, but the actual proof construction steps provided by PVS are very different. We will point out those commands which relate to the rules given above in the **notes** section of the command description.

Chapter 4

The PVS Proof Commands

The sequent calculus inference rules displayed in Chapter 3 for the basis for the proof commands used to construct proofs with PVS. The PVS proof commands are however significantly more powerful than these simple inference rules so as to make proof construction process more illuminating and less tedious. Proof commands can be typed in by the user at the `Rule?` prompt or they can be automatically applied by PVS as part of a proof strategy. A PVS proof command when applied to a goal sequent either

1. Succeeds in proving the goal sequent
2. Generates one or more subgoal sequents
3. Does nothing, which provides crucial control information to the strategy mechanism
4. Signals a failure that is propagated up the proof tree in order to control proof search
5. Postpones proof construction on the current goal sequent, transferring focus to the next remaining subgoal.

For example, in the list of commands below, a command like `bddsimp` succeeds in proving goal sequents that are just propositionally true, whereas the `case` command typically generates two or more subgoals. The `skip` command does nothing, but many other commands can also have no effect on the state of the proof particularly when the arguments to the command cause the parser or typechecker to signal errors. The `fail` command is the only command that signals failure. Failure is used to either backtrack or to abandon a proof. The `postpone` command is the only command that causes the current subgoal to be postponed.

The commands implemented by the PVS proof checker can be classified as:

1. Help: `help`.
2. Annotation: `comment`, `label`, `unlabel`, and `with-labels`.
3. Control: `fail`, `postpone`, `quit`, `rewrite-msg-off`, `rewrite-msg-on`, `set-print-depth`, `set-print-length`, `set-print-lines`, `skip`, `skip-msg`, `trace`, `track-rewrite`, `undo`, `untrace`, and `untrack-rewrite`.
4. Structural rules: `copy`, `delete`, `hide`, `hide-all-but`, and `reveal`.
5. Propositional rules: `bddsimp`, `case`, `case*`, `flatten`, `flatten-disjunct`, `iff`, `lift-if`, `prop`, `propax`, `split`, and `merge-fnums`.
6. Quantifier rules:
   (a) Existential: `inst`, `instantiate`, `instantiate-one`, `inst-cp`, `inst?`, and `quant?`.
   (b) Universal: `detuple-boundvars`, `skolem`, `skolem-typepred`, `skolem!`, `skosimp`, `skosimp*`, `generalize`, and `generalize-skolem-constants`.
7. Equality rules: `beta`, `case-replace`, `name`, `name-replace`, `name-replace*`, `replace`, `replace*`, and `same-name`.
8. Rules for using definitions and lemmas:
   (a) Definition expansion: `expand`, and `expand*`.
   (b) Using lemmas: `forward-chain`, `lemma`, `use` and `use*`.
   (c) Rewriting with definitions/lemmas: `rewrite`, `rewrite-lemma`, and `rewrite-with-fnum`.
9. Extensionality rules: `apply-eta`, `apply-extensionality`, `decompose-equality`, `eta`, `extensionality`, `replace-eta`, and `replace-extensionality`.
10. Induction rules: `induct`, `induct-and-rewrite`, `induct-and-rewrite!`, `induct-and-simplify`, `measure-induct`, `measure-induct+`, `measure-induct-and-simplify`, `name-induct-and-rewrite`, `rule-induct`, `rule-induct-step`, `simple-induct`, and `simple-measure-induct`.
11. Rules for simplification using decision procedures and rewriting: `assert`, `bash`, `both-sides`, `do-rewrite`, `ground`, `grind`, `record`, `reduce`, `simplify`, `simplify-with-rewrites` and `smash`.

12. Installation and Removal of rewrite rules: `auto-rewrite`, `auto-rewrite!`, `auto-rewrite!!`, `auto-rewrite-defs`, `auto-rewrite-explicit`, `auto-rewrite-theories`, `auto-rewrite-theory`, `auto-rewrite-theory-with-importings`, `install-rewrites`, `stop-rewrite`, and `stop-rewrite-theory`.

13. Making type constraints explicit: `typepred`, and `typepred!`.

14. Model Checking: `model-check`, and `musimp`.

15. Converting a strategy to a rule: `apply`.

16. Strategies: `branch`, `checkpoint`, `else`, `if`, `just-install-proof`, `let`, `query*`, `quote`, `repeat`, `repeat*`, `rerun`, `spread`, `spread!`, `spread@`, `then`, `then*`, `then@`, `time`, `try`, and `try-branch`

4.1 Formal and Actual Parameters of Rules

Each of the proof commands takes a list of zero or more required and optional parameters. Each optional parameter has an associated default value. If the *[default]* part of an optional parameter is missing, it is taken to be `nil`. A rule with its formal parameter list is presented in the form:

(⟨*rulename*⟩ ⟨*required*⟩* &OPTIONAL ⟨*optional[default]*⟩* &REST ⟨*argument*⟩)

The &OPTIONAL and &REST are metalanguage keywords used in this reference guide to indicate how the arguments are to be provided; they are never legal arguments themselves.[1] The &OPTIONAL keyword indicates that the arguments which follow it are *optional*. Such arguments may be provided either by *position* or by *keyword*. To provide the argument by position, simply include values for all the preceding arguments followed by the value of the argument in question. This is usually the most convenient way to use the commands. Occasionally, you will want the default taken for most of the optional arguments, and only want to specify a different value for one near the end of the list. In this case, you may provide a pair of arguments, the first being the name of the argument preceded by a colon, and the second the value for the argument. This will be made clear in the examples below. The &REST keyword indicates that zero or more values may be provided for the indicated argument and these are accumulated into a list. The &REST argument can also by supplied as a list by keyword.

[1]Those with a background in Lisp will note the resemblance. However, note that &OPTIONAL as used in PVS is a combination of &OPTIONAL and &KEY.

Note that many proof rules have arguments indicating the number or numbers of the sequent formulas where the rule is to be applied. The syntactic convention is that when a single such number is expected, we indicate the argument as being an *fnum* (for "sequent formula number"), and where a list of such numbers is expected, we indicate the argument as *fnums*. Typically, a single number is acceptable where a list is expected, and denotes the singleton list containing that number. The list of antecedent sequent formulas can be indicated by '`-`', the list of consequent sequent formulas can be indicated by '`+`', and the list of all sequent formulas can be indicated by '`*`'. The use of this notation will be illustrated in the examples below. Lists here mean a sequence of formula numbers separated by whitespace (space, tab or carriage return) and surrounded by parentheses. The `-`, `+`, `*` indicators are preferable to specific numbers since they are more robust in the face of changes to the formula being proved.

The value provided for a *name*, *expr*, or *type* argument should be a legal corresponding PVS expression enclosed in a pair of string quotes (`"`). As with *fnums*, lists of these may be expected when the argument is *names*, *exprs*, or *types*.

When interacting with the prover you are essentially interacting with Lisp, and it is possible to give arguments that are ill-formed enough to cause problems. One such problem occurs when parentheses or string quotes are unbalanced. The immediate sign of this is that the system does not respond. To verify this, look on the right hand side of the status line of the `*pvs*` buffer; it will display `:ready` when waiting for a (complete) command, and `:run` when processing. Until it says `:run`, you may freely edit the command, even if it takes multiple lines. Other keystrokes can cause problems that break into Lisp (for example, typing a period at the `Rule?` prompt). If this happens, type `(restore)`, which causes the focus to return to the last point of interaction with the `Rule?` prompt. Rarely, the system will not respond correctly to the `(restore)` function, in which case you will have to abort to the top level (by typing `:reset`). In this case, the proof attempt is lost and you will have to start the proof over—though it still has the previously saved proof attempt.

Here are some examples of rules with their formal parameter lists:

- (lemma *name* &OPTIONAL *subst*)
- (replace *fnum* &OPTIONAL *fnums[*]* *dir[LR]* *hide?* *actuals?*)
- (delete &REST *fnums*)

The following are possible invocations of the above proof rules:

- `(lemma "assoc")`
- `(lemma "assoc" ("x" "1" "y" 2 "z" 3))`

- `(lemma "assoc" :subst ("x" "1" "y" 2 "z" 3))`
- `(replace -1)`
- `(replace -1 :dir RL)`
- `(replace -1 (2 -2 3) RL)`
- `(delete)`
- `(delete 1)`
- `(delete -2 1 -3)`
- `(delete :fnums (-2 1 -3))`

4.1.1 Rules versus Strategies

A PVS proof command given at the `Rule?` prompt can either invoke a rule or a strategy. A rule in PVS is an atomic operation that typically generates zero or more subgoals from the given goal. A strategy need not be atomic. An application of a strategy expands into a number of atomic steps. The atomic proof steps resulting from this expansion of the strategy are saved in the final proof and these are executed directly when the proof is rerun. The `apply` rule applies a given strategy as an atomic step thus converting a strategy into a rule. Rules are either primitive or defined, and the defined rules are defined as strategies but applied as atomic proof steps. For example, the rule for replacement using an antecedent equality, `replace`, is a primitive rule whereas `rewrite` is a defined rule and is defined by a strategy that uses the `replace` rule. Several proof commands that are rules have non-atomic analogues given by strategies: `prop` is the atomic propositional simplification rule and `prop$` is the corresponding strategy.

In pragmatic terms, strategies should be used when the expanded proof is of interest and otherwise, rules should be used. So for instance, `prop` should almost always be favored over `prop$` since the details of propositional simplification are seldom interesting. It is also useful to invoke the strategy corresponding to a rule in order to observe the inner workings of the strategy.

PVS features a strategy language for defining new rules and strategies. There is a corresponding interpreter for the strategy expressions defined using this language. The strategy language contains constructs for selecting among alternative proof strategies (`if`), for backtracking (`try`), and for invoking Lisp code (`let`). Strategies can also be defined using recursion.

4.1.2 Proof Checker Pragmatics

It is not necessary to master all the proof commands in order to use the PVS proof checker effectively. It is advisable to learn the most powerful commands first and try these and only rely on the simpler commands when the powerful ones fail. Broadly speaking, there are two typical kinds of proofs: those that require induction and those that do not. For proofs by induction, one of the induction commands is usually the first step, and `induct-and-simplify` is the most powerful and useful of these. The commands `induct-and-rewrite` and `induct-and-rewrite!` are variants of `induct-and-simplify`.

The `grind` command is usually a good way to complete a proof that does not require induction, and only requires definition expansion, and arithmetic, equality, and quantifier reasoning. The behavior of `grind` can be controlled through its various optional arguments, particularly `if-match` and `defs`. Simpler forms of `grind` such as `bash`, `reduce`, and `smash` can be used when `grind` becomes difficult to control. The `grind` command can sometimes instantiate existential strength quantifiers prematurely, and when this happens, it is often more appropriate to first apply `(grind :if-match nil)`, which performs all the simplifications of `grind` except quantifier instantiation, followed by `(grind)` to pick up the instantiations exposed by the first `grind`.

In a more interactive proof attempt, the initial step in a proof is usually the introduction of Skolem constants, and the preferred and most powerful form here is `skosimp*`. Note that a universal quantifier is needed for induction and in such cases, `skosimp*` might go too far and `skolem!` or `skosimp` might be more appropriate.

The decision procedure command `assert` is used very frequently particularly since it does simplification, automatic rewriting, and records type information and the sequent formulas in the decision procedure database for use in future simplifications. The more restrictive forms of `assert`, namely, `simplify`, `do-rewrite`, and `record` also come in handy. The command `simplify-with-rewrites` can be used to temporarily install and apply rewrite rules using `assert`.

The `inst?` command is the most powerful way to automatically instantiate quantifiers of existential strength. It has several options to control the selection of suitable instances.

The `bddsimp` command is the most efficient way to do propositional simplification, but `prop` will do when efficiency is not important. Propositional simplification has to be used with care since it can often generate lots of subgoals that share the same proof. The `flatten` and `split` commands must be used to do the propositional simplification more delicately. The `case` command is very useful as a way of introducing case splits into a proof. The `lift-if` command is typically needed to bring the case analyses in an expanded definition to the surface of the sequent where it can be propositionally simplified.

Of the control commands, `postpone` is used to cycle through the pending subgoals in the proof, and `undo` is used to recover from fruitless paths in a proof.

In addition to the above commands, it helps to be familiar with the prelude theories which contain a lot of useful background mathematics. Advanced users wishing to define their own proof strategies should examine the definitions of the basic strategies supplied with PVS. A file containing these definitions is distributed with the system.

We now describe each of these groups of rules. The table at the beginning of each group briefly summarizes the effect of each command and indicates whether the rule is *primitive* or *defined*. This distinction is not crucial to the use of the theorem prover. The defined commands can be redefined by the user, but the primitive commands capture the underlying PVS logic and therefore cannot be changed. Each rule is an atomic step in the proof. There are 'glass-box' versions for some of the defined rules where the rule is executed as a strategy. The name for such a proof strategy is typically the rule name with a '`$`' suffix. For example, the glass-box version of `induct-and-simplify` is `induct-and-simplify$`. The documentation indicates the presence of both the black-box and glass-box versions of this rule by listing the name as `induct-and-simplify/$`.

4.2 The Help Rule

`help`	*primitive*	provides brief documentation

4.2.1 `help`: Get help for proof commands

syntax: `(help` &OPTIONAL *`name[*]`*`)`

effect: Displays a brief description of a specific primitive proof command, defined rule, or strategy, or of all of the rules, defined rules, and strategies. Apart from displaying help information, the `help` rule behaves as a `(skip)`, *i.e.*, it has no effect on the proof.

usage: `(help)` **:** Displays help on all of the rules, defined rules, and strategies

`(help rules)` **:** Displays help on all the primitive rules.

`(help defined-rules)` **:** Displays help on all of the defined rules.

`(help strategies)` **:** Displays help on all of the strategies.

`(help skolem)` **:** Displays help on the `skolem` rule.

`(help prop$)` **:** Displays help on the `prop$` strategy.

notes: This command should only be used interactively. It is usually better to use the Emacs commands `M-x help-pvs-prover`, `M-x help-pvs-prover-command`, `M-x help-pvs-prover-strategy`, or `M-x help-pvs-prover-emacs`, since they display the information in a buffer for repeated reference. The Emacs commands `M-x x-prover-commands` when used in conjunction with X Windows displays a mousable window listing the prover commands.

4.3 The Annotation Rules

`comment`	*primitive*	attach comments to proof sequents
`label`	*primitive*	attach labels to sequent formulas
`unlabel`	*primitive*	removes labels from sequent formulas
`with-labels`	*defined*	labels resulting formulas

4.3.1 `comment`: Attach Comments to a Proof Sequent

syntax: `(comment` *`string`*`)`

effect: Attaches a comment string to the current proof sequent that is printed with preceding semi-colons above the sequent formulas. This comment string is also saved with the proof. The `comment` command can be nested within strategies and the comments are retained on the subgoals generated by the strategy.

usage: `(comment "3rd induction case")` : Prints the comment string `;;;3rd induction case` between the sequent label and the sequent formulas.

4.3.2 `label`: Attaches a Label to Sequent Formulas

syntax: `(label` *`string-or-symbol fnums`* &OPTIONAL *`push?`*`)`

effect: It is often useful to group and label a collection of related formulas in a proof sequent. The `label` command is used for this purpose. Each sequent formula can bear at most one label. The label is printed alongside the *fnum* whenever a proof sequent is displayed. A label can be used wherever an *fnum* is expected. A label can supplied as either a string, e.g., `"label"` or a symbol, e.g., `|label|`, though it is stored internally as a symbol. Labels are automatically inherited by any subformulas of a sequent formulas that appear through the application of an inference rule, e.g., `flatten` applied to a consequent formula $A \vee B$ labelled `main` results in two sequent formulas A and B both labelled `main`.

When *push?* is `t`, the new label is added to any existing labels on the formula. Otherwise, the given label replaces any existing ones.

usage: `(label "uniqueness" -3)` : Labels the formula numbered `-3` by the label `uniqueness`.

`(label "type-constraints" (-1 -3 -4))` : Labels the formulas numbered `-1`, `-3`, and `-4` by the label `type-constraints`.

`(label "antecedents" - :push? t)` : Adds the label `type-constraints` in addition to any other labels on the formulas numbered `-1`, `-3`, and `-4`.

`(bddsimp "type-constraints" )` : Applies BDD-based propositional simplification to the formulas labelled `type-constraints`.

notes: Note that the `bddsimp` command does not retain labels since there is no simple way to retain the connection between the formula returned by BDD-simplification and its original parent formula.

4.3.3 `unlabel`: Removes Labels from Sequent Formulas

syntax: `(unlabel` &OPTIONAL *`fnums`*`)`

effect: Removes all labels from the formulas in *fnums*. When *fnums* is not specified, all labels from all formulas are removed.

4.3.4 `with-labels`: Labels New Sequent Formulas

syntax: `(with-labels` *`rule labels`* &OPTIONAL *`push?`*`)`

effect: Given a proof step *rule* and a list of list of *labels* $((l_{11}\ldots)\ldots(l_{n1}\ldots))$, if the rule generates n subgoals, then the j'th new sequent formula in the i'th subgoal is assigned the label l_{ij}. If there are more subgoals than label lists, then the last label list is applied to the remaining subgoals. In each pairing of new formulas with labels in a list, if there are more formulas than labels, the last label is applied to the remaining new formulas. A singleton list of labels can be replaced by a single label.

When *push?* is `t`, the new label is added to any existing labels on the formula. Otherwise, the given label replaces any existing ones.

usage: `(with-labels (flatten) (("l1" "l2" "l3")))`: Applies `flatten` rule to the current proof subgoal and labels the new sequent formulas thus produced as `l1`, `l2`, and `l3`, respectively.

`(with-labels (prop) (("l11" "l12" "l13") ("l21" "l22")))`: Applies the `prop` rule and labels the new formulas in the first subgoal by labels `l11`, `l12`, and `l13`, and the new formulas in any remaining subgoals are labelled by labels `l21` and `l22`.

`(with-labels (prop) "prop-formulas")`: Labels all the new sequent formulas resulting from the application of `prop` by the label `prop-formulas`.

4.4 The Control Rules

`fail`	*primitive*	has the effect of signalling a failure
`postpone`	*primitive*	causes the current goal to be left pending
`quit`	*primitive*	quits a proof attempt
`rewrite-msg-off`	*defined*	inhibits rewriting commentary
`rewrite-msg-on`	*defined*	turns on rewriting commentary
`set-print-depth`	*defined*	sets the print-depth for formulas
`set-print-length`	*defined*	sets the print-length for formulas
`set-print-lines`	*defined*	sets the print-lines for formulas
`skip`	*primitive*	has no effect but is useful in defining strategies
`skip-msg`	*defined*	like *skip* but generates message
`trace`	*defined*	turns on tracing of proof commands
`track-rewrite`	*defined*	explains why a rewrite was not applied
`undo`	*primitive*	undoes proof steps along a branch of the proof
`untrace`	*defined*	turns off tracing of proof commands
`untrack-rewrite`	*defined*	turns off rewrite explanation

4.4.1 `fail`: Propagate Failure to the Parent

syntax: `(fail)`

effect: A failure signal is propagated to the parent proof goal. If the parent goal is not able to act on this signal, it further propagates the failure to its parent. This rule, like `skip`, is mainly employed in constructing strategies where it is used to control backtracking. Applying `fail` to the root sequent causes the proof to be unsuccessfully terminated.

usage: `(fail)`

errors: No error messages are generated.

notes: See the description of the `try` strategy in page 109 for examples of the use of `fail`.

4.4.2 `postpone`: Go to Next Remaining Goal

syntax: `(postpone` &OPTIONAL *`print?`*`)`

effect: Marks the current goal as pending to be proved and shifts the focus to the next remaining goal. By successively invoking `postpone` sufficiently often, it is possible to cycle back to the original focus. When *print?* is `t` commentary is suppressed.

usage: `(postpone)`

errors: No error messages are generated.

notes: The Emacs command `M-x siblings` shows the sibling subgoals of the current subgoal in an emacs buffer.

4.4.3 `quit`: Terminate the Proof Attempt

syntax: `(quit)`

effect: Terminates the current proof attempt, and queries whether the partial proof in progress should be saved. This way, it is possible to break and resume a long proof attempt by saving the partial proof and rerunning it when the proof attempts is resumed.

notes: This strategy should only be used interactively.

4.4.4 `rewrite-msg-off`: Inhibit Rewriting Commentary

syntax: `(rewrite-msg-off)`

effect: In the default mode, automatic rewriting by commands such as assert and do-rewrite generate a fairly verbose commentary. This can be entirely shut off by the rewrite-msg-off command. Behaves like a skip otherwise.

notes: Finer-grain control over the terseness of the rewriting commentary can be obtained by the Emacs commands `M-x set-rewrite-depth` and `M-x set-rewrite-length`.

4.4.5 `rewrite-msg-on`: Turn on Rewriting Commentary

syntax: `(rewrite-msg-on)`

effect: The rewriting commentary turned off by rewrite-msg-off can be restored by this command. Behaves like a skip otherwise.

4.4.6 `set-print-depth`: Set the Print Depth

syntax: `(set-print-depth num)`

effect: Sets the print depth for displaying formulas. Num must be a number. 0 means print the entire formula, any other number causes terms below the given depth to be elided. Behaves like a skip otherwise.

4.4.7 `set-print-length`: Set the Print Length

syntax: `(set-print-length num)`

effect: Sets the print length for displaying formulas. Num must be a number. 0 means print the entire formula, any other number causes terms longet than the given number to be elided. Behaves like a skip otherwise.

4.4.8 `set-print-lines`: Set the Number of Print Lines

syntax: `(set-print-lines num)`

effect: Sets the number of print lines for displaying formulas. Num must be a number. 0 means print the entire formula, any other number causes only the first num lines of each formula of the sequent to be displayed. Behaves like a skip otherwise.

4.4.9 `skip`: A Do Nothing Rule

syntax: `(skip)`

effect: Has no effect on the proof. The primary utility of `skip` is in writing strategies where a step is required to have no effect unless some condition holds. Typing `(skip)` in response to a goal sequent returns the same proof state with a `"No change."` message.

usage: `(skip)`

errors: No error messages are generated.

4.4.10 `skip-msg`: A Do Nothing, Print Something Rule

syntax: (skip-msg *string* &OPTIONAL *force-printing?*)

effect: Has no effect on the proof but prints the given *string*. The main use of `skip-msg` is in generating error messages from within strategies, typically as: `(if good?(input) ...(skip-msg "Bad input."))`.

usage: `(skip-msg "No such theory in current context.")` : Does nothing but prints the error message `"No such theory in current context."`.

`(skip-msg "No such theory in current context." :force-printing? T)` : Does nothing but prints the error message `"No such theory in current context."` even when the `skip-msg` appear within an `apply` where the printing of such messages is usually suppressed.

```
(let ((string
        (format nil "No such theory: ~a in current context."
                                                     theory)))
   (skip-msg string))
```

: Builds the string `string` for use within `skip-msg`.

errors: No error messages are generated.

4.4.11 `trace`: Trace Given Commands

syntax: (trace &REST *names*)

effect: Turns on the tracing of the proof commands named in *names* so that any time any one of the named rules or strategies is used in a proof, the entry into and exit out of such commands is traced. This makes it possible to check if the command is being properly invoked and has the desired effect. Behaves like a skip otherwise.

usage: `(trace assert inst? induct)`

errors: No error messages.

notes: `untrace` turns off tracing initiated by `trace`.

4.4.12 `track-rewrite`: Explains Failure to Apply Rewrite Rule

syntax: (track-rewrite &REST *names*)

effect: Explains why the attempt to apply a rewrite rule named in *names* was not applied. The typical reasons are:

1. The expression being rewritten did not match the left-hand side of the rewrite rule.
2. The match succeeded but generated type-correctness proof obligations that could not be simplified to `TRUE`.
3. The match succeeded but the corresponding conditions of the rewrite rule could not be simplified to `TRUE`.
4. The match succeeded and the corresponding conditions did simplify to `TRUE`, but the top-level conditional or `CASES` branch in the corresponding right-hand side of the rewrite rule was not simplifiable. This top-level conditional on the right-hand side should be simplifiable in the case of recursive definitions and ordinary rewrite rules which are not installed with the `always?` flag set to `T`.

Other than setting up the names of the rewrite rules to be tracked during simplification, `track-rewrite` behaves like a skip. It has no effect on the current proof sequent and is not saved as part of the partial or completed proof.

usage: `(track-rewrite "assoc" "append" "reverse_append" )`: Tracks the given rewrite rules during simplification and reports any failures corresponding to when the rewrites are unsuccessful.

errors: No error messages.

notes: `untrack-rewrite` turns off the tracking of rewrite rules initiated by track-rewrite.

4.4.13 `undo`: Undo Proof to an Ancestor

syntax: `(undo` &OPTIONAL *to[1]*`)`

effect: The undo command undoes the proof back to an ancestor node of the current node as indicated by the *to* argument. The user is then shown the sequent at that ancestor node, and asked for verification. The *to* argument can either be:

1. a positive number indicating the number of levels in the proof tree to be undone
2. a label, in which case the proof is undone to the lowest occurrence of a sequent with that label above the current sequent (since there can be many sequents in the proof with the same label; labels are only extended when there are multiple subgoals)

3. a proof rule or strategy in which case the proof is undone up to the lowest occurrence of a sequent where the given rule was applied by the prover or the given strategy was supplied by the user, or
4. a rule name or strategy name, so that the proof is undone to the lowest occurrence of a sequent where a rule with the given name was applied by the prover or a strategy of the given name was supplied by the user.

Undo applies its effects relative to the current node, not the last command. Thus undoing immediately after a branch has been proved or postponed will not, in general, go back to the state of the proof tree just before the last command. However, undo can be used to undo the effect of an undo command if invoked *immediately* afterwards.

usage: `(undo)` : Undoes a single step of the proof.

`(undo 3)` : Undoes three steps in the proof.

`(undo undo)` : Undoes an undo, if it was the last command executed. If anything has been executed since the undo command, it is not possible to undo the undo.

`(undo (skolem 1))` : Undoes back to the lowest ancestor node where `(skolem 1)` was applied.

`(undo skolem)` : Undoes back to the lowest ancestor node where a proof rule or strategy of the form `(skolem ...)` was issued or applied.

`(undo skolem!)` : Undoes back to the lowest ancestor node where `skolem!` was issued.

`(undo "assoc.2")` : Undoes back to the lowest ancestor node labelled `assoc.2`, unless `assoc.2` labels the current node in which case there is no change.

notes:

- The `undo` command is only meant to be used interactively. Noninteractive strategies should use the `fail` command for the same effect.
- The Emacs command `M-x ancestry` shows the chain of proof goals leading back to the root node of the proof.
- A `(rerun)` command immediately after an `undo` will cause the undone proof to be rerun and restored.

4.4.14 `untrace`: Disables Tracing of Given Commands

syntax: `(untrace` &REST *`names`*`)`

effect: Turns off the tracing of proof commands named in *names*, as initiated by `(trace)`. Behaves like a `skip` otherwise.

usage: `(untrace assert)`

errors: No error messages

4.4.15 `untrack-rewrite`: Disables Tracking of Rewrite Rules

syntax: `(untrack-rewrite` &REST *names*`)`

effect: Disables the tracking of rewrite rules invoked by `track-rewrite`. When untrack-rewrite is invoked with no arguments, then tracking is discontinued for all currently tracked rewrite rules. Other than removing the given *names* from list of rewrite rules to be tracked during simplification, `untrack-rewrite` behaves like a skip. It has no effect on the current proof sequent and is not saved as part of the partial or completed proof.

usage: `(untrack-rewrite "assoc" "append" "reverse_append" ):` Disables tracking on the given rewrite rules.

`(untrack-rewrite):` Disables tracking on all currently tracked rewrites rules.

errors: No error messages.

4.5 The Structural Rules

`copy/$`	*defined*	inserts a copy of a sequent formula
`delete`	*primitive*	deletes selected formulas from a goal sequent
`hide`	*primitive*	temporarily hides selected formulas from the displayed goal
`reveal`	*primitive*	reveals hidden formulas

Sequent calculus based proof systems employ structural rules to rearrange the formulas in a sequent. The typical structural rules are described in Chapter 3. In PVS, the Exchange rule is entirely omitted since the PVS proof commands already ignore the order of formula occurrences in a sequent except for the use of formula numbers. The Contraction rule is not built into PVS and only appears in a limited form; the rule for instantiating quantifiers of existential strength permits the copying of these quantified formulas so that they can be reused, if needed. Some use of Contraction is already built into the rules so that the non-principal formulas are shared between the premises of a rule. The defined rule `copy` also implements Contraction. The Weakening rule is present in PVS as the `delete` rule below. The `hide` rule is a more cautious form of `delete`, where certain sequent formulas can be suppressed and recovered later in the proof using the `reveal` rule. The emacs command `M-x show-hidden-formulas` displays hidden formulas along with their numbers.

4.5.1 `copy/$`: Copy Selected Formula

syntax: `(copy` *fnum*`)`

effect: Inserts a copy of the sequent formula numbered *fnum* into the sequent. If the given formula is an antecedent formula, then the copy becomes the first antecedent formula, and if it is a consequent formula, then the copy becomes the first consequent formula.

usage: `(copy -3)` : Makes a copy of the formula numbered -3 and inserts it as the first antecedent formula.

errors: Could not find formula number *foo*: attempt to copy a formula which does not exist.

4.5.2 `delete`: Delete Selected Formulas

syntax: `(delete` &REST *fnums*`)`

effect: Returns the subgoal that is the result of deleting all of the sequent formulas in the current goal that are indicated by *fnums*. If there are no formulas in the sequent corresponding to those indicated in *fnums*, then the effect is that of a `(skip)`.

usage: `(delete *)` : Deletes every formula in the sequent yielding a subgoal that is an empty sequent. This invocation of the rule is not advisable because the empty sequent is unprovable.

`(delete +)` : Yields the subgoal where all the consequent formulas in the current goal sequent have been deleted.

`(delete -)` : Same as above with antecedent formulas.

`(delete 2)` : Yields the subgoal where formula number `2` in the current subgoal is deleted.

`(delete (-1 4 -3 2))` : Yields the subgoal where formulas numbered `-1`, `4`, `-3`, and `2` in the current subgoal are deleted.

`(delete -1 4 -3 2)` : Same as above.

errors: No error messages are generated.

notes: When in doubt, use `hide` instead of `delete`.

4.5.3 `hide`: Hide Selected Formulas

syntax: `(hide` &REST *`fnums`*`)`

effect: This is a more cautious version of `delete`. The `hide` rule saves the deleted sequent formulas that are indicated by *fnums* so that they can be restored to a descendant of the current sequent by the `reveal` rule (see below). Note that the non-copying version of the `instantiate` rule and the `inst` rule automatically hide the quantified formula so that quantifiers can be later reinstantiated along the same branch of the proof if needed.

usage: `(hide 2)` : Yields the subgoal sequent that results from hiding the formula number `2` in the current goal sequent.

`(hide (-1 4 -3 2))` : Yields the subgoal sequent that results from hiding the formulas numbered `-1`, `4`, `-3`, and `2` in the current goal sequent.

`(hide -1 4 -3 2)` : Same as above.

errors: No error messages are generated.

notes: Hidden formulas play no role in a proof until they are revealed. Thus in addition to eliminating "clutter" in the display, they can also affect the performance of the ground prover which contains decision procedures for equality and linear arithmetic.

4.5.4 `hide-all-but/$`: Hide Selected Formulas

syntax: `(hide-all-but` &OPTIONAL *`keep-fnums fnums[*]`*`)`

effect: This is a variant of the `hide` rule that hides all the formulas indicated by *fnums* except those indicated by *keep-fnums*. As with `hide`, hidden sequent formulas are saved and can be restored to a descendant of the current sequent by the `reveal` rule.

usage: `(hide-all-but (-1 -4) -)` : Yields the subgoal sequent that results from hiding all the antecendent formulas except the formulas numbered `-1` and `-4` in the current sequent.

`(hide-all-but * (-2 1))` : Hides formulas numbered `-2` and `1` in the current sequent.

`(hide-all-but :keep-fnums (-2 3))` : Hides all formulas except those numbered `-2` and `3` in the current sequent.

errors: No error messages are generated.

4.5.5 `reveal`: Reveal Some Hidden Formulas

syntax: `(reveal` &REST *`fnums`*`)`

effect: To see the hidden formulas in sequent form, use the emacs command `M-x show-hidden`. Invoking `reveal` yields a subgoal that reintroduces the hidden formulas numbered *fnums* (in the sequent displayed by `M-x show-hidden`) into the current sequent. The formulas thus revealed are not removed from the list of hidden formulas.

usage: `(reveal -2)` : Reveals the formula numbered `-2` in the sequent displayed by `M-x show-hidden`.

`(reveal (-1 4 -3 2))` : Reveals the formulas numbered `-1`, `4`, `-3`, `2` in the sequent displayed by `M-x show-hidden`.

`(reveal -1 4 -3 2)` : Same as above.

errors: No error messages are generated.

4.6 The Propositional Rules

`bddsimp`	*primitive*	propositional simplification rule
`case`	*primitive*	introduces case splits (the Cut rule)
`case*/$`	*defined*	introduces case splits (the Cut rule)
`flatten`	*defined*	disjunctive simplification
`flatten-disjunct`	*primitive*	controlled disjunctive simplification
`iff`	*primitive*	converts boolean equalities to *if and only if* form
`lift-if`	*primitive*	the `IF`-lifting rule
`prop/$`	*defined*	propositional simplification rule
`propax`	*primitive*	the propositional axiom rule
`split`	*primitive*	the conjunctive splitting rule
`merge-fnums`	*defined*	combines sequent formulas

4.6.1 `bddsimp`: Propositional Simplification

syntax: `(bddsimp` &OPTIONAL *fnums[*] dynamic-ordering? irredundant?[t]*`)`

effect: Generates subgoals by applying propositional simplification using an external package written in C and based on binary decision diagrams (BDDs). Each distinct atomic Boolean formula in the sequent is converted into a literal and the top-level propositional structure is translated into input that is accepted by the BDD package. The result is translated back into the a list of subgoal sequents.

The *dynamic-ordering?* flag when set to `T`, allows the BDD package to reorder literals to reduce the BDD size.

The *irredundant?* flag, when set to `T`, normalizes the BDD so that the generated subgoals are independant, *i.e.*, no subgoal is subsumed by any of the others. This is quite expensive, and large BDDs can take a long time to process, but without it proofs may need to be repeated on multiple subgoals.

usage: `(bddsimp)`: Repeatedly applies the propositional rules to all the formulas in the sequent to generate zero or more subgoals.

`(bddsimp +)`: Applies propositional rules to the consequent formulas.

`(bddsimp + T)`: Uses the dynamic reordering heuristic to control BDD size while applying propositional simplification to the consequent formulas.

errors: No error messages.

4.6.2 `case`: Case Analysis on Formulas

syntax: `(case` &REST *exprs*`)`

effect: If the current sequent is of the form $\Gamma \vdash \Delta$, then the rule `(case` $A_1 \ \ldots A_n$`)` generates the subgoals

$$\begin{array}{rcl} A_n, \ldots, A_1, \Gamma & \vdash & \Delta \\ A_{n-1}, \ldots, A_1, \Gamma & \vdash & A_n, \Delta \\ A_{n-2}, \ldots, A_1, \Gamma & \vdash & A_{n-1}, \Delta \\ & \vdots & \\ A_1, \Gamma & \vdash & A_2, \Delta \\ \Gamma & \vdash & A_1, \Delta \end{array}$$

Note that the `case` command generates $n+1$ subgoals given n formulas. This allows us to assume a formula or a collection of formulas and subsequently prove these formulas to be true. The formulas A_i are given as strings, *e.g.*,, `"x > 0 AND y > 0"`. The given formulas $A_1, \ldots, A_n$ are parsed and typechecked and are expected to be of type `bool`. The typechecking of these formulas could generate additional subgoals corresponding to the type correctness conditions *e.g.*, `(case "(1/x) > 0")` would generate an additional subgoal with the proof obligation requiring that `x /= 0`. It is quite common for this command to generate parser and typechecker errors which simply return control back to the proof checker without affecting the state of the proof.

This command is extremely useful for transforming an undesirable expression t into a more desirable form s when this cannot be achieved by the other proof commands. The command `(case "`t` = `s`")` followed by `replace` can achieve the desired transformation. This combination is encapsulated in the defined rule `case-replace` (page 57).

usage: `(case)` : same as a `(skip)`.

`(case "x > 0")` : splits into two cases, assuming `x > 0` as an antecedent on one branch of the proof and placing `x > 0` as a proof obligation along the other branch of the proof. There might be additional branches if the typechecking of the formula in place of `x > 0` generates type correctness proof obligations.

`(case "x > 0" "y > 0")` : splits a goal $\Gamma \vdash \Delta$ into three subgoals:

- $\mathtt{y > 0}, \mathtt{x > 0}, \Gamma \vdash \Delta$
- $\mathtt{x > 0}, \Gamma \vdash \mathtt{y > 0}, \Delta$
- $\Gamma \vdash \mathtt{x > 0}, \Delta$

errors: The `case` rule can generate the following error messages:

No formulas given. This means the argument list was empty.

Irrelevant free variables . . . occur in formulas. No sequent in a PVS proof can have free variables in it. They can only contain bound variables, ordinary constants, and Skolem constants. The formulas given as arguments to the `case` rule must not contain any free variables.

boolean expected here. This indicates that one of the given formulas did not typecheck to the expected type `boolean`.

Parser error: . . . One of the given formulas did not parse correctly.

Typecheck error: . . . The typechecking of the given formulas failed in one of a variety of ways.

notes: The `case` rule corresponds to applications of the Cut rule. This command is surprisingly useful for explicitly controlling case splits in a proof, and for introducing assumptions that will eventually be discharged.

4.6.3 `case*/$`: Full Case Analysis on Formulas

syntax: `(case*` &REST *exprs*`)`

effect: Like the `case` command, but performs a fully branching case analysis. If the current sequent is of the form $\Gamma \vdash \Delta$, then the rule `(case*` $A_1 \ \ldots A_n$`)` generates the subgoals

$$\begin{array}{rcl} A_n, \ldots, A_1, \Gamma & \vdash & \Delta \\ A_{n-1}, \ldots, A_1, \Gamma & \vdash & A_n, \Delta \\ A_n, A_{n-2}, \ldots, A_1, \Gamma & \vdash & A_{n-1}, \Delta \\ A_{n-2}, \ldots, A_1, \Gamma & \vdash & A_n, A_{n-1}, \Delta \\ & \vdots & \\ A_n, \Gamma & \vdash & A_{n-1}, \ldots, A_1, \Delta \\ \Gamma & \vdash & A_n, \ldots, A_1, \Delta \end{array}$$

Note that the `case*` command generates 2^n subgoals given n formulas. This allows us to assume a formula or a collection of formulas and subsequently prove these formulas to be true.

usage: `(case*)` : same as a `(skip)`.

`(case* "x > 0" "y > 0")` : splits a goal $\Gamma \vdash \Delta$ into four subgoals:

- `y > 0`, `x > 0`, $\Gamma \vdash \Delta$
- `x > 0`, $\Gamma \vdash$ `y > 0`, Δ
- `y > 0`, $\Gamma \vdash$ `x > 0`, Δ
- $\Gamma \vdash$ `y > 0`, `x > 0`, Δ

errors: The `case*` rule can generate the same error message as the `case` command.

4.6.4 `flatten`: Disjunctive Simplification

syntax: `(flatten` &REST *fnums[*]*`)`

effect: A sequent formula is a disjunct if it is either an antecedent formula of the form $\neg A$ or $A \wedge B$, or a consequent formula of the form $\neg A$, $A \supset B$, or $A \vee B$. Disjunctive simplification transforms each indicated formula into a list of formulas that contains no disjuncts by repeatedly transforming

1. An antecedent formula $\neg A$ into the consequent formula A
2. An antecedent formula $A \wedge B$ into the two antecedent formulas A and B
3. a consequent formula $\neg A$ into the antecedent formula A

4. a consequent formula $A \supset B$ into the antecedent formula A and the consequent formula B
5. a consequent formula $A \vee B$ into the two consequent formulas A and B
6. an antecedent formula $A \iff B$ into the two antecedent formulas $A \supset B$ and $B \supset A$.

The `flatten` rule yields a subgoal where the indicated formulas in the current goal are disjunctively simplified. The rule behaves as a `(skip)` if none of the indicated formulas can be disjunctively simplified. The current goal is proved if disjunctive simplification yields an antecedent formula FALSE or a consequent formula TRUE.

usage: `(flatten)` : disjunctively simplifies every formula in the current goal sequent yielding a subgoal that contains no disjuncts that are sequent formulas.

`(flatten 2)` : disjunctively simplifies formula number `2` in the current goal sequent.

`(flatten +)` : disjunctively simplifies all the consequent formulas in the current goal sequent.

`(flatten (-1 4 -2))` : disjunctively simplifies the formulas numbered `-1`, `4`, and `-2` in the current goal sequent.

`(flatten -1 4 -2)` : Same as above.

errors: No error messages are generated.

notes: This command corresponds to repeated applications of the inference rules $\wedge \vdash$, $\vdash \vee$, $\vdash \supset$, $\neg \vdash$, and $\vdash \neg$ in Chapter 3. Note that these are all the propositional rules which do not cause branching. Since this command does not cause any branching, it is always safe to use and generally makes the sequent easier to read.

4.6.5 `flatten-disjunct`: Controlled Disjunctive Simplification

syntax: `(flatten-disjunct` &OPTIONAL *`fnums depth`*`)`

effect: As per `(flatten)`, but with an optional *depth* argument which can be used to control the depth to which the top-level disjuncts in a sequent formula are flattened. If the depth is not given, then the disjunctive simplification is carried out without any bound on the depth.

usage: `(flatten-disjunct + :depth 2)` : flattens the consequent disjunctive formulas, but only up to a depth of 2.

errors: No error messages are generated.

4.6.6 `iff`: Converting Boolean Equality to Equivalence

syntax: `(iff` &REST *`fnums[*]`* `)`

effect: Yields a subgoal where any boolean equalities of the form $A = B$, among the formulas in the current sequent that are indicated by *fnums* are converted to $A \iff B$.[2] Treating all boolean equalities as equivalences is not a good idea since that leads to a combinatorial explosion when the propositional steps are applied, and in many such cases, equality reasoning is sufficient to complete the proof.

usage: `(iff)` : same as `(iff *)`. Converts any boolean equalities among the sequent formulas into equivalences. Behaves like `(skip)` if there are no such boolean equalities.

`(iff -3)` : converts the formula numbered `-3` into an equivalence.

`(iff (4 2 -1))` : converts the formulas number `4`, `2`, and `-1` into equivalences.

`(iff 4 2 -1)` : Same as above.

errors: No error messages are generated.

4.6.7 `lift-if`: Lifting Embedded IF Connectives

syntax: `(lift-if` &OPTIONAL *`fnums updates?[T]`* `)`

effect: In proving properties of programs, the proof often splits up into cases according to the branching structure of the program. This branching structure is typically expressed using the `IF`-connective or the `CASES` construct. Since these `IF` and `CASES` branches could occur embedded within the formula, this branching structure must be lifted to the top level of the formula where the propositional simplification steps can be applied (see the `flatten` and `split` commands above). The `lift-if` rule lifts the leftmost-innermost contiguous `IF` or `CASES` *branching structure* out to the top level. An example of such a transformation is the rewriting of $f(\texttt{IF}(A, B, \texttt{IF}(C, D, E)))$ to

[2] Recall that $\iff$ is written as `iff` or `<=>` in the raw PVS language.

IF$(A, f(B),$ IF$(C, f(D), f(E)))$. On the other hand, $f($IF(IF$(A,B,C),D,E))$ is transformed by `lift-if` to IF$(A, f($IF$(B,D,E)), f($IF$(C,D,E)))$, reflecting the selection of the conditionals of the leftmost-innermost IF-expression. Note that $f($IF$(A,$ IF$(B,C,D),$ IF$(E,F,G)))$ is transformed to IF$(A,$ IF$(B, f(C), f(D)),$ IF$(E, f(F), f(G)))$ reflecting the preservation of the contiguous IF-branching structure. It is more effective to lift a contiguous block of IF branches since it more accurately reflects the case structure of the resulting argument and results in a more efficient IF-expression (since the branches B and E in previous example are kept independent). Note that only conditionals without bound variables can be lifted, and this is used as a criterion by the `lift-if` rule in selecting the branching structure. The leftmost-innermost branching structure typically turns out to be the most appropriate one. If this choice of branching structure turns out to be inappropriate, the `case` command (see below) can be used to carry out the desired case analysis.

Unless the *update?* flag is NIL, the `lift-if` command has been extended to extract the case structure from an array or function update. An expression of the form `(f WITH [(x)(u) := 3, (y)(v) := 5])(z)(w)` is converted to

```
(IF z = y THEN
    (IF w = v THEN 5
     ELSIF z = x THEN (IF w = u THEN 3 ELSE f(z) ENDIF)
     ELSE f(z)
     ENDIF)
 ELSIF z = x THEN 3
 ELSE f(z)
 ENDIF)
```

The resulting IF-expression is then lifted by the `lift-if` command.

usage: `(lift-if)` : same as `(lift-if *)`. Yields the subgoal got by lifting the leftmost-innermost branching structure in each of the formulas in the current sequent. Applications where the operator is an update are converted into IF-expressions which are also lifted.

`(lift-if +)` : yields the subgoal got by lifting the leftmost-innermost branching structure in each of the consequent formulas in the current sequent.

`(lift-if -3)` : lifts the branching structure from the formula numbered `-3` in the current sequent.

`(lift-if (-1 3 -2))` : lifts the branching structure from the formulas numbered `-1`, 3, and `-2` in the current sequent.

`(lift-if (-1 3 -2)) :updates? NIL` : Same as above without the conversion of applications of updates to IF-expression form.

errors: No error messages are generated.

notes: This command roughly corresponds to the `IF` $\uparrow\vdash$ and $\vdash$ `IF` $\uparrow$ rules of Chapter 3. The `simplify`, `record`, and `assert` commands do a limited amount of "if-lifting."

4.6.8 `prop/$`: Propositional Simplification

syntax: `(prop)`

effect: Carries out propositional simplification on the current goal returning just those subgoals that are not propositional axioms and do not have any top-level propositional connectives. Using `prop` indiscriminately could lead to a combinatorial explosion of cases caused by splitting irrelevant conjunctions. This in turn could lead to a number of subgoals requiring identical proofs so `prop` should be used with some care. The recursive definition for `(prop)` is simply

```
(try (flatten) (prop) (try (split)(prop) (skip)))
```

notes: `prop` can be used for small-scale propositional simplification. For larger formulas, the primitive rule `bddsimp` which uses a BDD-based propositional simplifier is usually more efficient.

The `prop` rule should almost always be preferred to its strategy version `prop$`.

4.6.9 `propax`: Propositional Axioms

syntax: `(propax)`

effect: An application of `propax` either proves the sequent or behaves like a `(skip)`. Invoking `propax` proves sequents of the form

1. $\ldots, \text{FALSE}, \ldots \vdash \Delta$,
2. $\Gamma \vdash \ldots, \text{TRUE}, \ldots$,
3. $\Gamma \vdash \ldots, t = t, \ldots$, or
4. $\ldots, A, \ldots \vdash \ldots, B, \ldots$, where the sequent formulas A and B are syntactically equivalent (*i.e.*, identical upto the renaming of bound variables).

The first two forms are not actually propositional axioms as described in Chapter 3, but may easily be inferred. The third form above is actually an equality rule, but it is useful to group it with the other propositional steps. These forms correspond to sequents that are structurally true, but it might be difficult to notice these forms in a complicated looking sequent.

usage: `(propax)`

errors: No error messages are generated.

notes: It is important to note that the `propax` step is automatically applied to every sequent that is ever generated in a proof, so that there is never any need to actively invoke it. It is simply included here for the sake of completeness.

4.6.10 `split`: Conjunctive Splitting

syntax: `(split` &OPTIONAL *`fnum[*] depth`*`)`

effect: Selects and splits a conjunctive formula in the current goal sequent based on the information given in *fnum*. The `split` command splits every top-level conjunction in the selected formula so that the resulting formulas are no longer conjunctions. A conjunctive formula A in a goal sequent of the form $\Gamma, A \vdash \Delta$ or $\Gamma \vdash A, \Delta$ is split by collecting lists of antecedent and consequent formulas by recursively collecting subformulas of A as follows:

1. if A is an antecedent formula of the form $B \vee C$, then collect antecedent formulas B and C
2. if A is an antecedent formula of the form $B \supset C$, then collect antecedent formula C and consequent formula B
3. if A is a consequent formula of the form $B \wedge C$, then collect consequent formulas B and C
4. if A is a consequent formula of the form $B \iff C$, then collect consequent formulas $B \supset C$ and $C \supset B$
5. if A is an antecedent formula of the form $\texttt{IF}(B, C, D)$, then collect antecedent formulas $B \wedge C$ and $\neg B \wedge D$
6. if A is a consequent formula of the form $\texttt{IF}(B, C, D)$, then collect consequent formulas $B \supset C$ and $\neg B \supset D$.

If this process yields the collection of antecedent formulas $A_1, \ldots, A_m$ and the consequent formulas $B_1, \ldots, B_n$, then the `split` command yields the subgoals $\Gamma, A_i \vdash \Delta$ for $0 < i \leq m$, and $\Gamma \vdash B_j, \Delta$ for $0 < j \leq n$. When there is no conjunction in the current sequent as indicated by `fnum`, then the `split` rule behaves as a `(skip)`.

When the *depth* argument is given, the top-level conjuncts are only split to that given depth.

usage: `(split)` : (Same as `(split *)`.) Yields the subgoals resulting from splitting the first conjunction found in the current sequent. It is not easy to tell which conjunct would be split since a sequent is internally represented as a list of formulas, but is displayed in sequent form where the negated formulas appear in the antecedent part, and the unnegated formulas appear in the consequent part. The `(split +)` and `(split -)` rules (see below) might be more appropriate when such a confusion exists.

`(split +)` : splits the first consequent conjunctive formula in the current goal sequent.

`(split -)` : splits the first antecedent conjunctive formula in the current goal sequent.

`(split -3)` : splits the formula numbered `-3` in the current sequent.

`(split + :depth 2)` : splits the top-level consequent conjuncts to a maximum of two levels.

errors: No error messages are generated.

notes:

- This command causes branching, so should be used with caution; otherwise you will find yourself doing essentially the same proof on many different branches.
- The `split` command corresponds to the $\vdash \wedge$, $\vee \vdash$, $\supset\vdash$, `IF` $\vdash$, and $\vdash$ `IF` inference rules given in Chapter 3. These are all the inference rules which cause branching.
- To get the effect of repeatedly applying `flatten` and `split`, use the `prop` or `bddsimp` commands.

4.6.11 `merge-fnums/$`: Combine Sequent Formulas

syntax: `(merge-fnums` *fnums* `)`

effect: If the sequent is of the form $A_1, \ldots, A_m \vdash C_1, \ldots, C_n$, and *fnums* picks out the sequent formulas A_i, A_j, C_k, and C_l, then in the resulting sequent, these formulas are replaced by the single formula $A_i \wedge A_j \supset C_k \vee C_l$. Presently, this command is mainly useful in defining the `generalize` command which needs sequent formulas merged into one formula so that a universal quantifier can be wrapped around it. Applying `merge-fnums` to a single formula has no effect.

usage: `(merge-fnums (-1 -3 4))`: Merges and replaces the sequent formulas numbered `-1`, `-3`, and `4` by a single formula, namely a consequent formula asserting the implication between the conjunction of `-1` and `-3` and `4`.

notes: The `merge-fnums/$` command is used in defining the `generalize` strategy in order to collect together the formulas where the term to be generalized by a universally quantified variable appears.

4.7 The Quantifier Rules

`detuple-boundvars/$`	*defined*	distributes tuple and record quantification
`generalize/$`	*defined*	generalizes term by universal quantification
`generalize-skolem-constants/$`	*defined*	generalizes Skolem constants
`inst/$`	*defined*	instantiates existentially quantified variables
`instantiate`	*primitive*	instantiates existentially quantified variables
`instantiate-one`	*defined*	instantiates existentially quantified variables
`inst-cp/$`	*defined*	copies and instantiates existential formulas
`inst?/$`	*defined*	heuristically instantiates existential formulas
`quant?/$`	*defined*	heuristically instantiates existential formulas
`skolem`	*primitive*	*Skolemizes* universal formulas
`skolem!/$`	*defined*	automatically introduces *Skolem* constants
`skolem-typepred`	*primitive*	*Skolemizes* with type constraints
`skosimp/$`	*defined*	introduces *Skolem* constants and flattens
`skosimp*/$`	*defined*	repeatedly Skolemizes and flattens

4.7.1 `detuple-boundvars/$`: Distributes Tuple and Record Quantification

syntax: `(detuple-boundvars` &OPTIONAL *fnums* `[*]` *singles?*`)`

effect: A top-level sequent formula of the form `(FORALL (x: [S1, S2, S3]): g(x))` or `(FORALL (x: [# s : S, t : T #]) : g(x))` is replaced by `(FORALL (x1: S1), (x2: S2), (x3: S3): g(x1, x2, x3))` and `(FORALL (x1: S), (x2: T)): g((# s := x1, t := x2 #)))`. This decomposition of tuple and record quantification is needed, for example, to carry out an induction over one of the components. Tuple quantification can be introduced when instantiating parameterized theories such as the `function` theory in the prelude. This command is used within the `measure-induct` strategy.

4.7.2 `generalize/$`: Generalizes Term by Universal Quantification

syntax: `(generalize` *term var* &OPTIONAL *type fnums* `[*]` *subterms-only?* `[T])`

effect: If the sequent is of the form `a1(t), a2(t)` $\vdash$ `c1(t), c2(t)`, then applying the `generalize` term `t` with variable `x` yields a sequent of the form $\vdash$ `(FORALL x: (a1(x) AND a2(x)) IMPLIES (c1(x) OR c2(x)))`. More specifically, the `generalize` command collects together all the sequent formulas from the given *fnums* containing the term *term*, applies `merge-fnums` to obtain a single formula which is then generalized by replacing *term* by a universally quantified variable *var*.

The *type* option is to indicate that the universally quantified variable should be bound with the given *type*. This is useful if the term has a more specific type than is required of the generalization.

The generalization applies only to the subterms that are not within types or actuals. If a more sweeping generalization is needed, then the *subterms-only?* flag should be set to `NIL`.

4.7.3 `generalize-skolem-constants/$`: Generalizes Skolem Constants

syntax: `(generalize-skolem-constants` &OPTIONAL *fnums*`[*])`

effect: Applies universal generalization to the Skolem constants that occur in the given *fnums*. Such a step is useful in rearranging quantifiers by introducing skolem constants and generalizing them over selected formulas.

4.7.4 `inst/$`: Instantiation of Existential Strength Quantifiers Without Copying

syntax: `(inst` *`fnum`* &REST *`terms`*`)`

effect: This is simply a convenient form of the `instantiate` rule where the quantified formula is not copied and the terms are given in &REST form.

usage: `(inst - "a + 3" "_" "b")`: Instantiates the first universally quantified antecedent formula with exactly three bound variables with `a + 3` for the first bound variable and `b` for the third bound variable while leaving the second bound variable uninstantiated.

4.7.5 `inst-cp/$`: Instantiation of Existential Strength Quantifiers With Copying

syntax: `(inst-cp` *`fnum`* &REST *`terms`*`)`

effect: This is simply a convenient form of the `instantiate` rule where the quantified formula is copied and the terms are given in &REST form.

4.7.6 `inst?/$`: Instantiation of Existential Strength Quantifiers

syntax: `(inst?` &OPTIONAL *`fnums[*] subst where[*] copy? if-match polarity?`*`)`

effect: This rule extends the capabilities of the `inst` rule. Here the given substitution *subst* is used to select a quantified formula of existential strength where the quantifier binds all of the variables in *subst*. The rule then collects those atomic subformulas or subterms in this formula that contain free occurrences of all the outermost existentially quantified variables and tries to find a match (extending *subst*) for these in the sequent (or as constrained by the *where* argument). These pattern subterms of the quantified formula are collected as successive lists of templates containing all the quantified variables, all but one of the quantified variables, and so on. The templates are essentially collected starting from the leftmost-outermost one except for implications where the templates for the conclusion part of the implication precede those from the hypotheses.

In the default case when *if-match* is `nil`, the first successful match (for all or some of the quantified variables) for the first template with a successful match is used to generate one or more, partial or total instances of the chosen quantified formula. If a partial substitution is given but no match is found and *if-match* is `nil`, then the rule goes ahead and instantiates using the given partial substitution *subst*. If *if-match* is `T`, then the instantiation only takes place if the matching process succeeds. If *if-match* is `all`, then the command returns all possible instantiations for all of the templates in the first list of templates that yields a successful match. Note that these lists of templates are ordered by the number of quantified variables that occur free in them. If the *if-match* flag is `best`, then the command chooses an instantiation from the `all` case that generates the fewest TCCs when typechecked. If the *if-match* flag is `first*`, the command chooses all the instantiations of the first successful template.

The *polarity?* argument can be either `T` or `NIL`. When this is set to `T`, the `inst?` command is sensitive to the polarity with which the patterns occur and it matches these patterns only against expressions of opposing polarity. The polarity-sensitive matching pays attention to both boolean polarity (i.e., whether the expression occurs under an even or odd number of negations) as well as arithmetic polarity (i.e., whether the expression occurs on the lesser or greater side of an inequality.

Sometimes a single `inst?` can only find a partial instantiation where successive invocations of `inst?` can succeed in fully instantiating all of the quantified variables.

Note that if a bound variable name contains `$`, it is sufficient to only give that part of the name preceding `$` in *subst*. The *copy?* argument works exactly as in `quant` and is used to retain a copy of the quantified formula. Note that an uncopied, quantified formula is automatically hidden.

usage: `(inst? -1 ("x" 1) + T)`: Tries to instantiate the quantified variables in the formula number `-1` by pattern-matching against the subexpressions in the consequent formulas using the given partial substitution where `x` is instantiated to `1`. The first acceptable substitution found by pattern matching is returned. A copy of the original formula is retained.

`(inst? - :if-match best)`: Tries to instantiate the first universally quantified antecedent formula by pattern-matching the subexpressions of this formula against all the formulas in the sequent. The best substitution, namely one that generates the fewest TCCs when typechecked, is returned.

errors: **Given Substitution . . . is not of the form . . . :** A substitution must be of the form `(`*`var term var term`*` ...)`.

Couldn't find a suitable quantified formula: No instantiable formula found in the range specified by *fnums*.

Couldn't find a suitable instantiation for any quantified formula. Please provide partial instantiation: Pattern-matching was unable to instantiate the quantified variables so that a further hint in the form of a partial substitution might be needed.

Given substitution . . . is not of the form: (¡var¿ ¡term¿...): An odd length substitution was given.

The supplied terms should not contain free variables: Terms in given substitution should not contain free variables.

The types of the substituted variables contain free occurrences of the following quantified variables: . . . : If the type of one variable given in a substitution contains another quantified variable, then that variable must also be instantiated in the substitution.

4.7.7 `instantiate`: Instantiating Existential Quantifiers

syntax: `(instantiate` *`fnum exprs`* &OPTIONAL *`copy?`*`)`

effect: As the sequent calculus rules indicate, the universally quantified formulas in the antecedent and the existentially quantified formulas in the consequent are reduced by instantiating the quantified variables with the terms that are being *existentially generalized* in the proof. In an application of the `instantiate` rule, *fnum* is used to select the suitable quantified formula that is either an antecedent formula of the form $(\forall x_1, \ldots, x_n \colon A)$ or a consequent formula of the form $(\exists x_1, \ldots, x_n \colon A)$. The argument *exprs* provides the list of n terms $t_1, \ldots, t_n$ so that the chosen quantified formula is replaced by $A[t_1/x_1, \ldots, t_n/x_n]$ in the generated subgoal. Note that each t_i is typechecked to be of the type of x_i, and this typechecking could generate additional goals corresponding to the type correctness conditions. As with `skolem`, it is possible to leave some of the x_i uninstantiated by supplying `"_"` for the corresponding t_i in the *exprs* argument.

When the *copy?* parameter is `t`, then a copy of the quantified formula is saved in the subgoal sequent so that the quantifier can be reused. When the *copy?* parameter is `nil`, the quantified formula is automatically hidden so that the quantifier can be reused by revealing the hidden formulas using the `reveal` rule.

usage: `(instantiate * ("x + 3" "y - z"))` : Finds the first formula in the sequent that is either an antecedent formula of the form $(\forall x_1, x_2 \colon A)$, or a consequent formula of the form $(\exists x_1, x_2 \colon A)$ and replaces this formula with $A[\mathtt{x + 3}/x_1, \mathtt{y - z}/x_2]$.

`(instantiate - ("x + 3" "y - z"))` : Searches for the first suitable antecedent formula in the current sequent, and has the same effect as the above invocation of `instantiate`.

`(instantiate -3 ("x + 3" "y - z") T)` : Has the same effect as above on the formula numbered `-3`, but also makes a copy of the formula numbered `-3` provided it is a formula of the form $(\forall x_1, x_2 \colon A)$.

`(instantiate -3 ("x + 3" "y - z") :copy? T)` : Same as above.

errors: **No suitable quantified formula found:** There was no formula of existential strength in the range given by *fnum* with the same number of bound variables as the length of the supplied list of *exprs*.

Expecting ... terms, but ... terms provided: Wrong number of terms given in the *exprs* argument.

The supplied terms should not contain free variables: There can be no free variables in a PVS sequent and none are allowed to sneak in through the `instantiate` rule.

The types of the substituted variables contain free occurrences ...: If x and y are both bound by the top quantifier where the type of y contains free occurrences of x, then if the `instantiate` rule supplies a substitution for y, it must also include a substitution for x.

In addition to the above, the `instantiate` rule can generate parser and typecheck errors.

notes: The defined rule `inst` is a non-copying form of `instantiate`, and the rule `inst-cp` is a copying form of `instantiate`. In the related `inst?` rule, if the *exprs* argument is missing then the rule attempts to find a suitable instantiation by matching selected subterms of A with subterms in the rest of the sequent. Note that `inst?` does not always succeed with the right match. It typically succeeds when there are very few possible matches. The `instantiate` rule captures the $\forall\vdash$ and $\vdash\exists$ sequent rules.

4.7.8 `instantiate-one`: Instantiating Existential Quantifiers

syntax: `(instantiate-one` *`fnum terms`* &OPTIONAL *`copy?`*`)`

effect: Same as `instantiate`, but behaves as a `skip` if the instantiation would yield a formula already in the sequent. `instantiate-one` may be used in `repeat` in situations where `instantiate` would never terminate.

4.7.9 `quant?/$`: Instantiation of Existential Strength Quantifiers

syntax: `(quant?` &OPTIONAL *`fnum[*] subst where[*] copy? if-match`*`)`

effect: This rule is similar to `inst?` and is retained here for backward compatibility with PVS 1.x. It should no longer be used.

4.7.10 `skolem`: Replacing Universal Quantification with Constants

syntax: `(skolem` *`fnum constants`* &OPTIONAL *`skolem-typepreds?`*`)`

effect: If the formula in the current sequent indicated by *fnum* is either an antecedent formula of the form $(\exists x_1, \ldots, x_n \colon A)$ or a consequent formula of the form $(\forall x_1, \ldots, x_n \colon A)$, and *constants* is a list of new identifiers of the form `(`c_1 $\ldots c_n$`)`, then the `skolem` rule generates a subgoal where the indicated formula has been replaced by $A[c_1/x_1, \ldots, c_n/x_n]$.[3] When *fnum* is `*`, then the first appropriate sequent formula (*i.e.*, either an antecedent existential formula or a consequent universal formula binding n variables) is chosen for quantifier elimination. The parameter *fnum* can also be either `+` or `-`. If any of the c_i are given as `"_"`, then those bound variables are left alone and no corresponding skolem constants are introduced. If *skolem-typepreds?* is `T`, then typepreds will be introduced for the new constants.

usage: `(skolem * ("a3" "b3" "c3"))` : The first suitable formula in the sequent with the form $(\forall/\exists x_1, x_2, x_3 \colon A)$ is replaced by $A[\texttt{a3}/x_1, \texttt{b3}/x_2, \texttt{c3}/x_3]$. As mentioned earlier, it is not always easy to determine the first such suitable formula in the sequent since the displayed sequent only captures the correct order between and antecedent formulas and between consequent formula. The ordering of all formulas is not visible from the displayed sequent.

`(skolem - ("a3" "b3" "c3"))` : Same as above, but for the first suitable antecedent formula.

`(skolem -3 ("a3" "b3" "c3"))` : Same as above, but only for formula number `-3` in the current sequent.

`(skolem - ("a3" "_" "c3"))` : Replaces the first antecedent formula of the form $(\exists x_1, x_2, x_3 \colon A)$ with $(\exists x_2 \colon A)[\texttt{a1}/x_1, \texttt{a3}/x_3]$.

errors: No suitable quantified expression found: Either there is no antecedent existentially quantified formula or consequent universally quantified formula, or the list of skolem constants is of the wrong length.

[3]This is slightly inaccurate since in PVS, the type for the bound variable x_i in $(\forall x_1, \ldots, x_n \colon A)$ can contain free occurrences of x_j where $j < i$ which is not reflected by the substitution instance $A[c_1/x_1, \ldots, c_n/x_n]$.

Formula ... is not skolemizable: The indicated formula in *fnum* is not of the right form.

Expecting ... skolem constant(s), but ... supplied: The number of supplied skolem constants does not correspond to the number of bound variables in the formula specified by *fnum*.

The supplied skolem constants must all be new names: Either a skolem constant given as part of the *constants* argument was not a name or was already present in the context.

Duplicate use of skolem constants: The list of skolem constants contained a duplicate.

The types of the skolemized variables contain free occurrences ...: This error is triggered when the top quantifier binds variables x and y where the type of y contains a free occurrence of x. It is not legal in this situation to supply a skolem constant for y without providing one for x.

notes: The `skolem!` rule automatically generates skolem constants and is usually an easier alternative to the `skolem` rule. The `skolem` rule is useful when adopting certain conventions about naming skolem constants within proof strategies. The `skolem` rule captures the $\exists\vdash$ and $\vdash\forall$ sequent rules.

4.7.11 `skolem!`: Automatic Introduction of Skolem Constants

syntax: `(skolem!` &OPTIONAL *`fnum[*] keep-underscore?`*`)`

effect: Automatical generates Skolem names for the *names* argument for the `skolem` rule. These names have the form `x!`n when the bound variable being named is `x`. This rule is a dangerous one to include in defined rules or within an `apply` rule since the names being generated could change when a proof is rerun. When the *fnum* argument is `*`, `+`, or `-`, the first Skolemizable formula in this range is selected. There is a small loss of robustness with the commands `skolem!`, `skosimp`, and `skosimp*` since any changes in the constants generated due to a change in the formula or a reordering of proof steps can affect the behavior of subsequent commands that explicitly mention these constants.

The *keep-underscore?* flag when `T` ensures that the bound variable `x_1` is replaced by a skolem constant of the form `x_1!n` rather than the default `x!n` for some number `n`.

4.7.12 `skolem-typepred/$`: Skolemize with Type Constraints

syntax: (`skolem-typepred` &OPTIONAL *fnum[*]*)

effect: This is a variant of `skolem!` where the type constraints on the generated skolem constants are introduced as antecedent formulas.

4.7.13 `skosimp/$`: Skolemize then Flatten

syntax: (`skosimp` &OPTIONAL *fnum[*] preds?*)

effect: This is a short form for `(then (skolem` *fnum*`) (flatten))`. When `preds?` is `T`, `skolem-typepred` is used in place of `skolem` so that the type constraints on the generated skolem constants are introduced as antecedent formulas.

4.7.14 `skosimp*/$`: Repeatedly Skolemize then Flatten

syntax: (`skosimp*` &OPTIONAL *preds?*)

effect: This is a short form for `(repeat (then (skolem) (flatten)))`. As with `skosimp`, when the `preds?` flag is `T`, the command `skolem-typepred` is used in place of `skolem`.

4.8 The Equality Rules

`beta`	*primitive*	reduces λ-, record, tuple, update, datatype redexes
`case-replace/$`	*defined*	case-split and replace on an equality
`name`	*primitive*	introduces a name for a complex expression
`name-replace/$`	*defined*	replaces an expression with a name
`name-replace*/$`	*defined*	replaces expressions with names
`replace`	*primitive*	replaces equality LHS for RHS (or vice-versa)
`replace*`	*primitive*	replaces with equalities
`same-name`	*primitive*	equates two constants with equal actuals

4.8.1 `beta`: Beta Reduction

syntax: (beta &OPTIONAL *fnums[*] rewrite-flag*)

effect: The `beta` rule rewrites certain expressions called *redexes* to their reduced forms. The various forms of redexes and their reduced forms are:

- $(\lambda x_1 \ldots x_n : e)(t_1, \ldots, t_n)$ reduces to $e[t_1/x_1, \ldots, t_n/x_n]$. Note that `LET` and `WHERE` expressions are syntactic sugar for λ-redexes and will also be beta-reduced by `beta`.
- $\mathtt{proj}_i(t_1, \ldots, t_n)$ reduces to t_i.
- $label_i$`((#`$label_1$`:=`$t_1, \ldots, label_n$`:=`t_n`#))` reduces to t_i.
- $accessor_i(constructor(t_1, \ldots, t_n))$ reduces to t_i, where *constructor* is an abstract datatype constructor, and $accessor_i$ is an accessor of that constructor.
- `CASES` $c(t_1, \ldots, t_n)$ `OF ...,` $c(x_1, \ldots, x_n)$ `:`e`, ... ENDCASES` reduces to $e[t_1/x_1, \ldots, t_n/x_n]$, where c is a constructor for some datatype.
- *label*(r `WITH [...,` *label* `:= e,...]`) reduces to e, provided *label* `:=` e is the last update of *label* in the assignment above.
- (f `WITH [...,` (i) `:=` e`, ...])`(j) reduces to e if it can be shown that $i = j$, and the assignments following (i) `:=` e do not affect $f(j)$. If it can be shown using the decision procedures that none of the updates affect the value of f at j, then the expression simply reduces to $f(j)$.

The *rewrite-flag* argument is typically omitted. When its value is `LR`, it indicates that only the right-hand side of a formula that is an equality should be simplified using beta-reduction. Similarly, if the value of *rewrite-flag* is `RL`, then only the left-hand side of any equality is simplified.

usage: `(beta)` : Same as `(beta *)`. Yields the subgoal obtained by reducing all of the redexes in the current sequent.

`(beta +)` : Reduces all the redexes in the consequent formulas in the current sequent.

`(beta (-1 2 3))` : Reduces all of the redexes in the formulas numbered `-1`, 2, and `3` in the current sequent.

errors: There are no error messages generated. If there are no redexes to be reduced in the given range *fnums*, then the message `No suitable redexes found` is generated.

notes: The commands `assert` and `simplify` also carry out beta-reduction among many other simplifications.

4.8.2 `case-replace/$`: Introducing and Applying Equality

syntax: `(case-replace` *`expr`*`)`

effect: This is just the strategy `(then@ (case` *`expr`*`)(replace -1))`. It case splits on an equality (note that a non-equality A is interpreted as an equality $A =$ TRUE) and replaces the LHS by the RHS in the rest of the sequent.

usage: `(case-replace "c = 0")`: Replaces `c` by `0` in the current subgoal and generate a second subgoal with the proof obligation `c = 0`.

errors: This command can generate a parse or type error if the given *expr* is not well-formed.

4.8.3 `name`: Introducing Names for Terms

syntax: `(name` *`name expr`*`)`

effect: Yields a subgoal where a formula of the form *expr* = *name* is added as a new antecedent formula. This is typically useful as a step towards generalizing a formula by replacing expressions with constants (using the `replace` rule). The given *name* must be new. In addition to the new antecedent, a definition is generated for the name, which may subsequently be expanded or rewritten. It is also treated as an `AUTO_REWRITE-`, so that it is not expanded accidentally with the next `grind`, for example.

usage: `(name "d5" "(a + b + c)")` : Introduces the antecedent formula `(a + b + c) = d5`.

errors: This command can generate a parse or type error if the given *expr* is not well-formed.

. . . is not a symbol: The *name* argument must be a symbol.

. . . is already declared: The name argument must be a new name.

4.8.4 `name-replace/$`: Replacing a Term by a Name

syntax: `(name-replace` *`name expr`* &OPTIONAL *`hide?[T]`*`)`

effect: This command is just the strategy `(then@ (name` *`name expr`*`) (replace -1))`. It replaces the expression *expr* by *name* everywhere in the current sequent. The equality between `name` and `expr` is hidden by default, unless *hide?* is `NIL`.

4.8.5 `name-replace*/$`: Replacing Terms by Names

syntax: `(name-replace*` ***`name-and-exprs`*** &OPTIONAL ***`hide?[T]`***`)`

effect: This command is an iterated form of name-replace.

The `name-and-exprs` argument must be of the form `(`$\langle name_1 \rangle$ $\langle expr_1 \rangle$ `...)`. The command replaces each $expr_i$ in the current sequent with the corresponding $name_i$.

4.8.6 `replace`: Rewriting Using Equalities

syntax: `(replace` ***`fnum`*** &OPTIONAL ***`fnums[*] dir[LR] hide? actuals?`***`)`

effect: The `replace` rule is typically used to rewrite some selection of the formulas in the current sequent using an antecedent equality formula of the form $l = r$. The equality formula to be used is indicated by the *fnum* argument. The targets of the rewrites are listed in the *fnums* argument. When the *direction* argument is `RL` (denoting "right-to-left"), the target occurrences of r in the sequent are rewritten to l. Otherwise, when the *direction* parameter is different from `RL`, the target occurrences of l are rewritten to r. If *fnum* is the number for an antecedent formula A that is not an equality, then the formula is regarded as an equality of the form $A =$ TRUE. If *fnum* is the number of a consequent formula A, then the formula is regarded by `replace` as an equality of the form $A =$ FALSE. Note that the formula indicated by *fnum* is unaffected by `replace`.

When *hide?* is `T`, the formula indicated by *fnum* that is used for replacement is hidden using the `hide` command. When *actuals?* is `T`, the replacement is carried out within actual parameters of names. Otherwise, the replacement only occurs at the expression level.

usage: `(replace -1)` : If the formula numbered `-1` in the current sequent has the form $l = r$, then this application of the replace rule generates a subgoal where every occurrence of a term syntactically equivalent to l is replaced by r in every formula of the sequent other than in formula number `-1`. If the formula numbered `-1`, call it A, is not an equality, then it is treated as being an equality of the form $A =$ TRUE.

`(replace -1 (-1 2 3) RL)` : If the formula numbered `-1` in the current sequent has the form $l = r$, then this application of `replace` generates a subgoal where all the occurrences of r in the formulas numbered `2` and `3` are replaced by l. Note that formula number `-1` remains untouched.

`(replace 2)` : Yields the subgoal got by replacing all occurrences of the formula numbered `2` in the rest of the current sequent, by FALSE. Note that in the sequent representation, it is okay to use the negation of a consequent formula as an assumption.

errors: **No sequent formula corresponding to ...:** This means that the *fnum* argument was out of range and did not refer to a formula in the current sequent.

... must be $*, +, -$, an integer, or list of integers: The given *fnums* argument did not meet the criterion listed in the error message.

notes: One open issue regarding the `replace` rule is whether it is useful to have more refined control over the target occurrences of the rewrite than is provided by the *fnums* argument. The defined rule `case-replace` is used to assume an equality and apply it in the form of a replacement. The `replace` rule corresponds to the sequent rule **Repl**.

4.8.7 `replace*`: Rewriting Using Equalities

syntax: `(replace*` &REST *fnums*`)`

effect: Iteratively applies the `replace` command to each of the formulas indicated by the numbers in `fnums`. Unlike `replace`, only left-to-right replacement is possible and the formulas used in the replacement cannot be hidden.

4.8.8 `same-name`: Equating Names Under Actuals

syntax: `(same-name` *name1 name2* &OPTIONAL *type*`)`

effect: Two names such as `cons[{i : nat | i > 0}]` and `cons[{i : nat | i /= 0}]` can be denotationally equal yet syntactically distinct. The command `same-name` can be used to introduce an antecedent formula asserting the equality of two such names, *e.g.*,, `cons[{i : nat | i > 0}] = cons[{i : nat | i /= 0}]`, while generating the proof obligations required to show that the actuals coincide. In the above case, the obligation would be to show that `FORALL (i: nat): i > 0 IFF i /= 0`.

The *type* argument can be used to disambiguate the *name* references in case it has been overloaded.

errors: **Argument ... is not a name:** Names can have a theory prefix and/or actuals, but cannot be compound expressions.

Argument ... does not typecheck uniquely: Need to supply either theory prefixes and/or actuals for the name, or the *type* argument to disambiguate the name reference.

Argument ... must have actuals: It does not make sense to use `same-name` unless the names have actuals that can be demonstrated to be equal.

Arguments ... and ... must have identical identifiers: Only the actuals can be syntactically different between *name1* and *name2*.

4.9 Using Definitions and Lemmas

`expand`	*primitive*	expands (and simplifies) function definition
`expand*/$`	*defined*	expands function definitions
`forward-chain/$`	*defined*	forward chains on an implication lemma
`lemma`	*primitive*	introduces axiom, lemma, or definition instance
`rewrite/$`	*defined*	match and rewrite using lemma
`rewrite-lemma/$`	*defined*	rewrite using instance of lemma
`rewrite-with-fnum/$`	*defined*	rewrite using antecedent
`use/$`	*defined*	invokes lemma with instantiation
`use*/$`	*defined*	invokes lemmas with instantiation

4.9.1 `expand`: Expanding Definitions

syntax: `(expand` ***`name`*** &OPTIONAL ***`fnum[*] occurrence if-simplifies assert?`***`)`

effect: Expands (and simplifies) the definition of *name* at a given *occurrence*. If *occurrence* is not given, then all instances of the definition are expanded. The *occurrence* is given as a number n referring to the nth occurrence of the function symbol counting from the left, or as a list of such numbers. If the *if-simplifies* flag is `t`, then any expansion within a sequent formula occurs only if the expanded form can be simplified (using the decision procedures). The *if-simplifies* flag is needed to control infinite expansions in case `expand` is used repeatedly inside a strategy. In the default case when *assert?* is `NIL`, `expand` applies the `simplify` step with the default settings to any sequent formula in which a definition is expanded. When *assert?* is `T`, `expand` applies the `assert` version of `simplify` to any sequent formulas affected by definition expansion (This latter option has to be exercised for compatibility with PVS 1.x). The *assert?* flag can also be `NONE` in which case no simplification is applied to the sequent formula following expansion.

usage: `(expand "sum")` : Expands the definition of `sum` throughout the current sequent, whether they simplify or not. The resulting expressions are all simplified using decision procedures and rewriting.

`(expand "sum" 1)` : Expands `sum` throughout the formula labeled `1`.

`(expand "sum" 1 2)` : Expands the second occurrence of `sum` in the formula labeled `1`.

`(expand "sum" :if-simplifies t)` : Expands those occurrences of `sum` whose definitions can be simplified by means of the decision procedures. This is only relevant in the situation where the definition is a `CASES` or `IF` expression. The definition expansion only occurs if such an expression simplifies to one of its branches.

`(expand "sum" :assert? T)` : Expands `sum`, but uses `assert` instead of `simplify` in the simplification process.

errors: Occurrence ... must be nil, a positive number or a list of positive numbers: Self-explanatory.

notes: Typically, the defined rule `rewrite` can be used instead of `expand` but `expand` has some advantages:

- `expand` is faster, since definitions are simple (unconditional) equations.
- `expand` does not require *name* to be fully resolved; it can use the occurrence to get the type information needed.
- `expand` allows a specific *occurrence* or occurrences of a function symbol to be expanded.
- `expand` can rewrite subterms containing variables that are bound in some superterm, *e.g.*,, If $f(x)$ is defined as $g(h(x))$, then `expand` would be able to rewrite $(\forall x.f(x) = 0)$ as $(\forall x.g(h(x)) = 0)$, but `rewrite` would not.

4.9.2 `expand*/$`: Expanding Several Definitions

syntax: `(expand*` &REST *`names`*`)`

effect: An iterated version of `expand` that applies the `expand` command to each of a list of function names.

4.9.3 `forward-chain/$`: Forward Chaining

syntax: `(forward-chain` *`name-or-fnum`*`)`

effect: This rule is used to forward chain on the given lemma or antecedent formula number. If the given lemma or antecedent formula has the form $A_1 \wedge \ldots \wedge A_n \supset C$, then this rule tries to match the formulas A_i against the antecedent formulas of the current sequent. If the match succeeds, the corresponding instance of C is added to the as an antecedent formula to current sequent. If this instance of C is already in the current sequent, then the `forward-chain` rule looks for other instances of the A_i in the current sequent. An *fnum* argument can be either `-`, `*`, or an antecedent formula number.

4.9.4 `lemma`: Employing Instances of Lemmas

syntax: `(lemma` *`name`* &OPTIONAL *`subst`*`)`

effect: This rule introduces an instance of the lemma named *name* corresponding to the substitutions supplied in *subst* as a new formula in the sequent. Axioms, assumptions, and function definitions are also seen as lemmas for the purpose of this rule. There can be several forms for the definition of a function. For example, there are four possible forms for the definition of a function f such that $f(x, y)(u)(v)$ is given to be e, for some term e:

1. $f(x, y)(u)(v) = e$
2. $f(x, y)(u) = (\lambda v\colon e)$
3. $f(x, y) = (\lambda u\colon (\lambda v\colon e))$
4. $f = (\lambda x, y\colon (\lambda u\colon (\lambda v\colon e)))$

In such a situation, the `lemma` rule picks the first form if *subst* contains substitutions for x, y, u, and v, and in general, it picks the last definition (in the order of presentation above) in which all the variables in the substitution *subst* occur free in the definition.

In using the `lemma` rule, *name* must name a lemma that is visible in the context of the statement being proved, and *substs* must be a list of substitutions of the form $(x_1\ t_1\ \ldots\ x_n\ t_n)$. Let A be the universally closed form of the lemma named *name*, the lemma rule checks that each x_i in *subst* is a *substitutable* variable in A, *i.e.*, A must have the form $(\forall y_1, \ldots, y_m\colon B)$ and x_i is either identical to some y_j or is substitutable in B. PVS checks that each t_i is of the type of the corresponding x_i and does not contain any free variables. Note that it is possible that there are many possible instances of lemmas named *name* either because these lemmas come from different theories that are both visible in the current context or from different instances of the same parametric theory. The `lemma` rule attempts to resolve such an ambiguity by using the information given by the substitution *subst*. If this information is not enough to unambiguously choose the named lemma, then the lemma name must be supplied in a more complete form. The form *identifier[actuals]* for *name* can be used in the case when the generic theory where *name* occurs is unique, but the required instantiation for the parameters of the theory is not obvious from the context (*i.e.*, the sequent). The form *theoryname.identifier[actuals]* is to be used to further disambiguate the reference to the lemma.

The lemma rule can generate additional subgoals due to the type correctness conditions. As a consequence of the substitution, this rule can also generate parsing and typechecking errors.

usage: Consider the situation where the theory `boolean_props` contains a lemma named `assoc` stating the associativity of conjunction, and the theory `listprops` with a type parameter `T` has a lemma also named `assoc` stating the associativity of `append`. Both associativity lemmas are stated in terms of the variables `x`, `y`, and `z`.

`(lemma "assoc")` : This generates an error message asserting that the given name could not be resolved, and behaves like `(skip)`, otherwise.

`(lemma "boolean_props.assoc")` : Adds the formula

```
(FORALL x, y, z: ((x AND y) AND z) = (x AND (y AND z)))
```

to the antecedent of the current sequent.

`(lemma "assoc" ("x" "TRUE" "z" "FALSE"))` : Adds the formula

```
(FORALL y: ((TRUE AND y) AND FALSE) = (TRUE AND (y AND
                                    FALSE)))
```

to the antecedent of the current sequent. Notice that the substitution has been used to resolve the lemma name.

`(lemma "assoc" ("x" "true" "z" "false" "y" "A"))` : Adds the formula

```
((TRUE AND A) AND FALSE) = (TRUE AND (A AND FALSE))
```

to the antecedent of the current sequent.

`(lemma "assoc[list[nat]]")` : Adds the statement of the associativity of `append` for lists of natural numbers to the antecedent of the current sequent.

errors: The following are not possible variables . . . : One of the expressions in a variable position in the given substitution was not a name. Check the substitution to see if it has the form (x_1 t_1 . . .x_n t_n), where the x_i are all variable names.

The form of a substitution is . . . : This means that the substitution argument was a list of odd length and did not have the form (x_1 t_1 . . .x_n t_n).

Irrelevant free variables ... in substitution: As with other rules that introduce new expressions into the sequent, no free variables can be allowed.

Couldn't find a definition or a lemma named ... : There is no lemma or definition with the given name in the current context. Check the name.

Unable to resolve ... relative to substitution: The given name led to an ambiguity that could not be resolved from the given substitutions. Check the name or the substitutions, make the name more explicit, or provide additional substitutions.

notes: The defined rules `rewrite` and `rewrite-lemma` employ the `lemma` and `replace` rules to apply a lemma as a rewrite rule. The rules `rewrite`, `rewrite-lemma`, and `expand` are ways to expand definitions without introducing any new antecedents.

The `lemma` rule is a form of the Cut rule where one of the branches has been separately proved.

4.9.5 `rewrite/$`: Matching and Rewriting with Lemmas

syntax: (rewrite *name* &OPTIONAL *fnums[*] subst target-fnums[*] dir[LR] order[IN]*)

effect: The `rewrite` rule extends `rewrite-lemma`. It tries to automatically determine the required substitutions by matching the conclusion of the lemma against expressions in the formulas in *fnums*. This rule is always to be preferred to `rewrite-lemma` since it does the hard work of figuring out the substitutions. The *target-fnums* corresponds to the *fnums* argument of `rewrite-lemma`. If the *order* argument is `IN` (which is the default), then the matching is to be done inside-out, and if it is `OUT`, the matching is done outside in.

usage: `(rewrite "assoc")`: Finds and rewrites a single instance of the lemma `assoc` throughout the current goal.

`(rewrite "assoc" +)`: Looks for a matching instantiation for the lemma `assoc` in the consequent formulas but rewrites with this lemma instance throughout the current goal.

`(rewrite "assoc" + ("x" "A" "z" "B") -)`: Looks for a matching instantiation in the consequent formulas for the lemma `assoc` that extends the given substitution but only rewrites with this instance over the antecedent formulas.

`(rewrite "assoc" :dir RL :order OUT)`: Searches for a matching instance of the right-hand side of the lemma `assoc` in a leftmost-outermost order and rewrites the instance of the right-hand side by the corresponding instance of the left-hand side.

errors: No resolution for . . . : No such lemma was found in the current context.

Substitution . . . must be an even length list: Self-explanatory.

No sequent formulas for . . . : The *fnums* argument is incorrectly given.

No matching instance for . . . found: The current goal does not contain any instances of the rewritable part of the given lemma.

notes: It is usually more effective to install a rewrite rule using `auto-rewrite` (or its variants) than to use the `rewrite` command. The `rewrite` command is typically useful when the rewrite generates a condition or a TCC proof obligation that cannot be discharged automatically.

4.9.6 `rewrite-lemma/$`: Rewriting Using a Lemma

syntax: `(rewrite-lemma` *`lemma subst`* &OPTIONAL *`fnums[*] dir[LR]`*`)`

effect: This is an extension of the `lemma` rule that carries out rewriting given the required substitutions. Here *name* is either the name for a lemma or a definition that can be used as a rewrite rule or *subst* is a substitution list of the form (x_1 a_1 ... x_n a_n). Each t_i must be a term with no free variables, and each x_i is the identifier for a substitutable variable in *name*, *i.e.*, one that is either a free variable or is universally quantified at the block of universal quantifiers at the outermost level of the formula. The substitution list must provide substitutions for all the substitutable variables, otherwise the rule will not carry out a rewrite. In the case of definitions of curried operators, this rule picks the least curried form whose left-hand side includes all the variables for which substitutions have been provided. The formula numbers in *fnums* are the target formulas for where the rewriting occurs. The *dir* is either `LR` (by default) indicate a left-to-right use of the lemma as a rewrite rule, or `RL` for a right-to-left use. The `rewrite-lemma` rule also has some capability of resolving the *name* from the given substitutions, *i.e.*, it tries to figure out the theory instance for the lemma to be used.

usage: This command is similar to `rewrite` except that the *subst* argument is required and the entire substitution has to be provided.

notes: This command is rarely used since the `rewrite` command (which is defined in terms of `rewrite-lemma`) is almost always preferable.

4.9.7 `rewrite-with-fnum/$`: Matching and Rewriting with Antecedent Formulas

syntax: `(rewrite-with-fnum` *`fnum`* &OPTIONAL *`subst fnums[*] dir[LR]`*`)`

effect: Applies the `rewrite` command to an antecedent formula indicated by *fnum* which is used as a rewrite rule. The input substitution *subst* is used to guide the matching process to find a match that extends *subst*. The optional *fnums* argument is used to direct the command to look for matches within the sequent formulas indicated by *fnums*. When *dir* is `LR`, the antecedent formula is used a rewrite rule in the left-to-right direction, and when it is `RL`, the rewriting occurs in the right-to-left direction.

4.9.8 `use/$`: Invoke lemma with heuristic instantiation and beta-reduction

syntax: `(use` *`lemma`* &OPTIONAL *`subst if-match[best]`*`)`

effect: An extension of the `lemma` command where the formula introduced is subject to repeated instantiation and beta-reduction using the `inst?` and `beta` commands. This is usually an effective alternative to the `lemma` command. The *subst* argument is as in the `lemma` command. The *if-match* command is as in the `inst?` command. The possible values for if-`match` are:

- `all`: Find all instances of the first matching template in the formula being instantiated.
- `best`: Find the best instance (i.e., one that generates the fewest TCCs) for the first matching template.
- `T`: Ignore the partial substitution given unless some matching template was found.
- `NIL`: Apply the partial substitution even if none of the templates yielded a match.

4.9.9 `use*/$`: Invoke lemmas with heuristic instantiation and beta-reduction

syntax: `(use*` &REST *`names`*`)`

effect: An iterated form of the `use` command which applies `use` with default arguments to a sequence of lemmas.

4.10 Using Extensionality

`apply-extensionality/$`	*defined*	uses extensionality to prove equality
`apply-eta/$`	*defined*	uses eta form extensionality
`decompose-equality/$`	*defined*	Reduce equality to component equalities
`eta/$`	*defined*	introduces eta version of extensionality
`extensionality`	*primitive*	introduces extensionality axiom
`replace-eta/$`	*defined*	replace using eta
`replace-extensionality/$`	*defined*	replace using extensionality

4.10.1 `apply-extensionality/$`: Apply Extensionality to an Equality

syntax: `(apply-extensionality` &OPTIONAL *`fnum[+] keep? hide?`*`)`

effect: This rule is an extension of the `extensionality` rule. Given a succedent in the form of an equation $l = r$, where the type of l and r has a corresponding extensionality axiom scheme, `apply-extensionality` will generate a new succedent that is the result of using `replace-extensionality` on l and r.

If the *keep?* flag is set to `T`, the antecedent equality introduced by the `apply-extensionality` command is retained in the resulting goal sequent.

If the *hide?* flag is set to `T`, the equality formula to which the apply-extensionality command has been applied, is hidden in the resulting sequent. It is more typical to require this formula to be hidden than not, which means that the default value of `NIL` for this flag is poorly chosen.

4.10.2 `apply-eta/$`: Apply Eta Form of Extensionality

syntax: `(apply-eta` *`term`* &OPTIONAL *`type`*`)`

effect: This rule is an extension of the `extensionality` rule. Given a succedent in the form of an equation $l = r$, where the type of l and r has a corresponding extensionality axiom scheme, `apply-extensionality` will generate a new succedent that is the result of using `replace-extensionality` on l and r.

4.10.3 `decompose-equality/$`: Decomposes Equality to Component Equalities

syntax: `(decompose-equality` &OPTIONAL *`fnum`*`[*]` *`hide?`*`[T])`

effect: Decomposes an antecedent or consequent equality of the form `t1 = t2` where the terms are of function, record, tuple, or a datatype constructor type. If the terms are of function type, the decomposition returns the universal quantification `(FORALL x: t1(x) = t2(x))`. If the terms are of record type, the decomposition returns the conjunction of equalities of the individual fields of the terms. The decomposition is similar for tuple types. The decomposition on datatype constructors returns the equalities on the corresponding accessor fields. If the equality is on the consequent side, or is a disequality on the antecedent side, then the `decompose-equality` rule is the same as `apply-extensionality`.

4.10.4 `eta/$`: An Axiom Scheme for Functions, Records, Tuples, and Abstract Datatypes

syntax: `(eta` *`type`*`)`

effect: This is a variant of the `extensionality` rule where the *eta* version of the axiom scheme is introduced as an antecedent formula.

usage: `(eta "[nat, nat -> nat]")`: Introduces the antecedent formula

```
(FORALL (u_2: [[nat, nat] -> nat]):
         LAMBDA (x_3: [nat, nat]): u_2(x_3) = u_2)
```

`(eta "[# a: nat, b: int #]")`: Introduces the antecedent formula

```
(FORALL (r_8: [# a: nat, b: int #]):
  (# a := a(r_8), b := b(r_8) #) = r_8)
```

`(eta "(cons?[nat])")` : Introduces the antecedent formula

```
(FORALL (cons?_var: (cons?[nat])):
         cons(car(cons?_var), cdr(cons?_var)) = cons?_var)
```

errors: No suitable eta formula for given type: Self-explanatory.

4.10.5 `extensionality`: An Axiom Scheme for Functions, Records, Tuples, and Abstract Datatypes

syntax: `(extensionality` *`type`*`)`

effect: The `extensionality` rule is similar to the `lemma` rule in that it introduces an extensionality axiom for the given *type* as an antecedent formula. An extensionality axiom can be generated corresponding to function, record, and tuple types, and constructor subtypes of PVS abstract datatypes.

usage: `(extensionality "[nat, nat -> nat]")` : Yields a subgoal got by adding an antecedent formula of the form

```
(FORALL (f, g: [nat, nat -> nat]) :
    (FORALL (i, j: nat) : f(i,j) = g(i,j))
       IMPLIES f = g)
```

to the current sequent.

`(extensionality "[nat, int]")` : Adds an antecedent formula of the form

```
(FORALL (u : [nat, int]), (v : [nat, int]):
    proj_1(u) = proj_1(v) AND proj_2(u) = proj_2(v)
 IMPLIES u = v)
```

`(extensionality "[# a: nat, b: int #]")` : Adds an antecedent formula of the form

```
(FORALL (r : [# a: nat, b: int #]),
        (s : [# a: nat, b: int #]) :
    a(r) = a(s) AND b(r) = b(s)
 IMPLIES r = s)
```

`(extensionality "(cons?[nat])")`[4] : Adds an antecedent formula of the form

```
(FORALL (x: (cons?[nat])), (y: (cons?[nat])) :
    car(x) = car(y) AND cdr(x) = cdr(y)
 IMPLIES x = y)
```

The extensionality rule applied to other constructor subtypes for PVS datatypes behaves similarly.

errors: In addition to parsing and typechecking errors, the following error messages are generated:

The following irrelevant free variables occur in the given type expression: As with the other rules, no free variables can be introduced into a proof sequent through a rule application.

Could not find a suitable extensionality axiom for . . . : This means that there is no extensionality axiom for the given type expression.

Could not find ADT extensionality axiom for . . . : The given type was a subtype of a PVS datatype but not a constructor subtype as is required for generating an extensionality axiom.

[4] `cons?` is defined in the PVS prelude; use the command `M-x view-prelude-theory` on the theory `list_adt` for its definition.

notes: The defined rule `apply-extensionality` makes it possible to directly apply the extensionality scheme to show two terms to be equal. The defined rule `replace-extensionality` uses extensionality to replace one term by another.

4.10.6 `replace-eta/$`: Replace Using Eta

syntax: `(replace-eta` ***term*** &OPTIONAL ***type keep?***`)`

effect: This rule extends the `eta` rule. The eta axiom scheme is instantiated with the given term, which is then used in a replace command. A specific type for term may be specified where typechecking the term may give rise to ambiguity.

When *keep?* is `T`, the instantiated eta axiom scheme that is introduced as an antecedent is retained in the goal sequent, and otherwise, it is discarded.

4.10.7 `replace-extensionality/$`: Replace Using Extensionality

syntax: `(replace-extensionality` ***expr1 expr2*** &OPTIONAL ***expected keep?***`)`

effect: This rule is an extension of the `extensionality` rule. It uses the relevant extensionality axiom scheme to demonstrate the equality of *expr1* and *expr2* and replaces *expr1* in the sequent with *expr2*. In some cases, the optional *expected* type might have to be supplied to resolve any ambiguities in the typechecking of the given expressions.

When *keep?* is `T`, the extensionality scheme that is introduced as an antecedent is retained in the goal sequent, and otherwise, it is discarded.

4.11 Applying Induction

`induct/$`	*defined*	invokes induction
`induct-and-rewrite/$`	*defined*	invokes induction and simplifies
`induct-and-rewrite!`	*defined*	induct and simplify with definitions
`induct-and-simplify/$`	*defined*	induction followed by simplification
`measure-induct/$`	*defined*	invokes measure induction
`measure-induct+/$`	*defined*	a better `measure-induct`
`measure-induct-and-simplify/$`	*defined*	measure induction with simplification
`name-induct-and-rewrite/$`	*defined*	invoke induction scheme by name
`rule-induct/$`	*defined*	induction on inductive relation
`rule-induct-step/$`	*defined*	induction on inductive relation
`simple-induct/$`	*defined*	introduces instance of induction scheme
`simple-measure-induct/$`	*defined*	introduces instance of induction scheme

4.11.1 `induct/$`: Automatically Employ Induction Scheme

syntax: `(induct` *`var`* &OPTIONAL *`fnum[1] name`*`)`

effect: The formula indicated by *fnum* must be a universally quantified, consequent formula. The variable name *var* must be quantified at the outermost level of this formula. As with the substitutions in `inst?`, it is sufficient to give that part of the variable name preceding the last _ mark, if there is one in the bound variable name and it is followed by a number. The bound variable must be of type, i.e., must include as a supertype, `nat` or a PVS abstract datatype for the induction scheme to be selected automatically. The induction scheme corresponding to the type is then instantiated with an induction predicate constructed from the formula *fnum* and the resulting base and induction subgoals are generated. The induction scheme can also be explicitly provided by naming it using the optional *name* argument. Typically, this name will have to be fully instantiated with the actual theory parameters. Note that user-supplied induction schemes must have a form similar to the induction schemes in the prelude or those generated by the abstract datatype mechanism: `(FORALL (p: pred[T]):` *induction subgoal* `IMPLIES` *goal*`)`, where `p` is to be instantiated by the induction predicate.

usage: `(induct "i")` : Given that `i` is of type `nat`, and the formula numbered `1` has the form `(FORALL ..., i, ...:` *p*`(..., i, ...))`, we get the instantiation of the natural number induction scheme with the induction predicate `(LAMBDA i: (FORALL ...:` *p*`(..., i, ...)))`. The resulting formula is then beta-reduced and simplified into the base and induction subcases. If the type of `i` is a subtype of `nat` such as, say, `(even?)`, then the subtype predicate is added to the induction predicate to get `(LAMBDA (i: nat): even?(i) IMPLIES (FORALL ...: p(..., i, ...)))`. If `i` has type that is a datatype such as binary trees or lists, then the induction scheme for that datatype is used by default.

`(induct "x" :fnum 2 :name "below_induction[N]")` : Employs the induction scheme named `below_induction` and instantiates it with a predicate taken from the sequent formula number 2.

errors: Could not find suitable induction scheme : The `simple-induct` rule invoked by this step failed to find an induction scheme for the given variable *var*, *fnum*, and *name*. Check that *var* occurs as an outermost universally quantified variable whose type contains no free occurrences of other bound variables, and that has a supertype that matches what is required by the (default or named) induction scheme.

No formula corresponding to ... : A bad *fnum* was given.

4.11.2 `induct-and-rewrite/$`: Induct then Rewrite/Simplify

syntax: `(induct-and-rewrite` *var* &OPTIONAL *fnum[1]* &REST *rewrites*`)`

effect: This command has been superseded in PVS 2 by the more general `induct-and-simplify` but is retained for backward compatibility with PVS 1. It employs `induct` on *variable* and *fnum* to select, instantiate, and introduce an induction scheme, and then uses the rewrite rules given in *rewrites* to simplify the resulting base and induction cases employing `skosimp*`, `assert`, `lift-if`, `inst?`, and `bddsimp`.

usage: `(induct-and-rewrite "x" 1 "append" "reverse")` : Introduces an instance of the induction scheme according to variable `x` obtained by instantiating it with the predicate formed from formula number 1, simplifies the resulting base and induction cases using skolemization, rewriting, decision procedures, if-lifting, and heuristic instantiation.

`(induct-and-rewrite "x" :rewrites ("append" "reverse"))` : Same as above.

4.11.3 `induct-and-rewrite!/$`: Induct then Rewrite/Simplify

syntax: `(induct-and-rewrite!` *`var`* &OPTIONAL *`fnum[1]`* &REST *`rewrites`*`)`

effect: This is a variant of `induct-and-rewrite` which automatically uses all the definitions in the given sequent in its rewriting/simplification. These definitions are used in their `!` form so that explicit definitions are always rewritten regardless of whether the right-hand sides are simplifiable. Additional rewrite rules can be given can be give using the *rewrites* argument. The usage is similar to `induct-and-rewrite`.

4.11.4 `induct-and-simplify/$`: Induct then Rewrite/Simplify

syntax: `(induct-and-simplify` *`var`* &OPTIONAL *`fnum[1] name defs[T] if-match[best] theories rewrites exclude`*`)`

effect: This is an extremely useful proof command for directing proofs involving induction followed by simplification. It uses `install-rewrites` to install the rewrites in *defs*, *theories* and *rewrites* while excluding those in *exclude*, invokes the the `induct` command on *var*, *fnum* and *name* to get the base and induction cases, which are then simplified by repeated application of `skosimp*`, `assert`, `lift-if`, `bddsimp`, and `inst?` (as controlled by the *if-match* argument).

usage: `(induct-and-simplify "i" :defs ! :theories "real_props" :rewrites "assoc" :exclude ("div_times" "add_div"))`: If `i` has type `nat`, then the natural number induction scheme is instantiated with a predicate constructed from sequent formula `1`, and the resulting cases are simplified using definitions in the given sequent (unconditionally expanding explicit definitions), the rewrites in the prelude theory `real_props` but excluding `div_times` and `add_div`, and the rewrite rule `assoc`.

`(induct-and-simplify "A" :defs nil :if-match nil)` : Inducts according the induction scheme given by the type of `A` and then simplifies without rewriting any of the definitions or rewrite rules, and does not perform any heuristic instantaion. The `:if-match` nil option is useful when the heuristic instantiation fails to find the right quantifier substitutions.

4.11.5 `measure-induct/$`: Invoke Measure Induction

syntax: `(measure-induct` *`measure vars`* &OPTIONAL *`fnum[1] order skolem-typepreds?`*`)`

effect: The command `measure-induct+` is the preferred way to invoke measure induction. The `measure-induct` command takes a *measure* expression and a list of the *induction* variables *vars* in which the measure is defined. These variables must occur universally quantified in the formula numbered *fnum*. This list of variables is needed in order to unambiguously identify those universally quantified variables on which the measure is defined. As with `induct`, the `measure-induct` command forms the induction predicate by lambda-abstracting the formula over the variables given in *vars*. The measure function is also obtained by lambda-abstracting the given *measure* over the variables in *vars*. The `measure_induction` scheme from the prelude is then instantiated with domain type of the measure, the measure, and the ordering on the range of the measure. The well-founded ordering is taken by default to be `<` on natural numbers or ordinals unless a different ordering is given through the *order* argument. The `lemma` rule is used to introduce the measure induction scheme instantiated with the selected induction predicate. The work so far is actually carried out by the `simple-measure-induct` proof step. The `measure-induct` step then beta-reduces, simplifies, and instantiates the measure induction lemma to discharge the goal sequent and in the process generates an induction subgoal consisting of an antecedent induction hypothesis and a consequent induction conclusion. If *skolem-typepreds?* is T, then typepreds are introduced for any introduced skolem constants.

The problem with `measure-induct` is that the arrangement of quantifiers in the induction hypothesis is unhelpful. If the formula numbered *fnum* has the form `(FORALL x, w: p(x, w))` where the induction variable is `x` and the measure is `m`, the induction predicate is `(LAMBDA x: (FORALL w: p(x, w)))`, and the resulting induction hypothesis has the form `(FORALL x: m(x) < m(c) IMPLIES (FORALL w: p(x, w)))`. This form nests the universal quantification on `w` which might be useful in guiding the instantiation of `x`. Therefore a more useful form of the induction hypothesis is with quantification rearranged as `(FORALL x, w: m(x) < m(c) IMPLIES p(x, w))`. This rearrangement is carried out by `measure-induct+`. See `measure-induct+` for usage.

4.11.6 `measure-induct+/$`: Invoke Measure Induction

syntax: `(measure-induct+` *`measure vars`* &OPTIONAL *`fnum[1] order`*`)`

effect: This is the preferred way to invoke measure induction. The `measure-induct+` command takes a *measure* expression and a list of the *induction* variables *vars* in which the measure is defined. These variables must occur universally quantified in the formula numbered *fnum*. The `measure-induct+` command invokes `measure-induct` to introduce the measure induction scheme instantiated with the selected induction predicate, and then to beta-reduce, simplify, and instantiate the measure induction lemma to discharge the goal sequent and in the process generates an induction subgoal consisting of an antecedent induction hypothesis and a consequent induction conclusion. The well-founded ordering is taken by default to be `<` on natural numbers or ordinals unless a different ordering is given through the *order* argument. If *skolem-typepreds?* is T, then typepreds are introduced for any introduced skolem constants.

The problem with `measure-induct` is that the arrangement of quantifiers in the induction hypothesis is unhelpful. If the formula numbered *fnum* has the form `(FORALL x, w: p(x, w))` where the induction variable is `x` and the measure is `m`, the induction predicate is `(LAMBDA x: (FORALL w: p(x, w)))`, and the resulting induction hypothesis has the form `(FORALL x: m(x) < m(c) IMPLIES (FORALL w: p(x, w)))`. This form nests the universal quantification on `w` which might be useful in guiding the instantiation of `x`. A more useful form of the induction hypothesis is with quantification rearranged as `(FORALL x, w: m(x) < m(c) IMPLIES p(x, w))`. This rearrangement is carried out by `measure-induct+`. The command `measure-induct-and-simplify` is similar to `induct-and-simplify` but uses `measure-induct+` instead of `induct`.

usage: `(measure-induct+ "length(x) + length(y)" ("x" "y") 2)` : Applies the instance of measure induction with measure `length(x) + length(y)` on the universally quantified variables `x` and `y` in the formula numbered `2` to return a goal with an induction conclusion and an induction hypothesis.

`(measure-induct+ "m(x)" "x" :order "smaller?")` : Applies measure induction on the measure `m(x)` in the universally quantified variable `x` and the well-founded ordering relation `smaller?` on the range of `m`. Note that this will generate a TCC subgoal where the well-foundedness of the ordering relation has to be established.

4.11.7 `measure-induct-and-simplify/$`: Measure Induction followed by Simplification and Rewriting

syntax: `(measure-induct-and-simplify` *`measure vars`* &OPTIONAL *`fnum[1] order expand defs[T] if-match[best] theories rewrites exclude`*`)`

effect: Invokes the appropriate instance of measure induction using `measure-induct+`, skolemizes the resulting induction conclusion, and expands the definitions listed in the *expand* argument to then generate the corresponding cases. The resulting subgoals are then simplified and the induction hypothesis is instantiated by heuristic matching and each subgoal is subject to further propositional splitting and simplification based on rewriting and decision procedures. This command is very similar to `induct-and-simplify` but employs measure induction and uses the *expand* argument to guide the case analysis. If multiple instances of the induction hypothesis are needed, the *if-match* argument can be given as `first*` to obtain all instances of the first matching template in the quantified formula, or `all` to obtain all matches for all templates.

usage: `(measure-induct-and-simplify "length(x) + length(y)" ("x" "y") :expand "merge" :theories "merge_sort")`: This command could for instance be used to try to prove that the merge of two ordered lists is ordered. It invokes measure induction on the sum of the lengths of the two lists, then expands the definition of `merge`, and then repeatedly simplifies, instantiates, and rewrites (using the theory `merge_sort`.

`(measure-induct-and-simplify "length(x) + length(y)" ("x" "y") :expand "quicksort" :if-match first* :theories "quicksort")`: Since there are multiple recursive calls in the recursive case of `quicksort`, the `first*` option to *if-match* is used.

4.11.8 `name-induct-and-rewrite/$`: Induction on a named scheme followed by simplification and rewriting

syntax: `(name-induct-and-rewrite` *`var`* &OPTIONAL *`fnum[1] name`* &REST *`rewrites`*`)`

effect: Subsumed by `induct-and-simplify`. This command was a variant of a `induct-and-rewrite` that could be told to use a particular induction scheme using the *name* argument.

4.11.9 `rule-induct/$`: Induction on an inductive relation

syntax: `(rule-induct` *`rel`* &OPTIONAL *`fnum[+] name`*`)`

effect: Applies the induction scheme given by an inductive relation *rel* to a sequent of the form:

$$\ldots \vdash (\forall x_1, \ldots, x_n \colon rel(x_1, \ldots, x_n) \supset \ldots)$$

or, alternately, the form:

$$\ldots, rel(c_1, \ldots, c_n), \ldots \vdash \ldots$$

The induction schemes corresponding to an inductive relation *rel* are `rel_weak_induction` and `rel_induction`.

This command applies repeated skolemization and flattening to the specified *fnum* (or the first positive, skolemizable consequent) before invoking `rule-induct-step` on the resulting subgoal. The strategy uses the weak induction scheme by default but can be told to use strong induction by supplying `rel_induction` as the *name* argument.

4.11.10 `rule-induct-step/$`: Induction on an inductive relation

syntax: `(rule-induct-step` *`rel`* &OPTIONAL *`fnum[-] name`*`)`

effect: Subsumed by `rule-induct`. Applies the induction scheme given by an inductive relation *rel* to a sequent of the form:

$$\ldots, rel(c_1, \ldots, c_n), \ldots \vdash \ldots$$

Searches for an antecedent formula of the form $rel(c_1, \ldots, c_n)$ but this formula can also be given explicitly as *fnum*. The induction predicate is formulated using all the sequent formulas containing the constants c_1 to c_n. The strategy uses the weak induction scheme by default but can be told to use strong induction by supplying `rel_induction` as the *name* argument.

4.11.11 `simple-induct/$`: Introduce induction scheme instance

syntax: `(simple-induct` *`var fmla`* &OPTIONAL *`name`*`)`

effect: Subsumed by `rule-induct`. Applies the induction scheme given by an inductive relation *rel* to a sequent of the form:

$$\ldots, rel(c_1, \ldots, c_n), \ldots \vdash \ldots$$

Searches for an antecedent formula of the form $rel(c_1, \ldots, c_n)$ but this formula can also be given explicitly as *fnum*. The induction predicate is formulated using all the sequent formulas containing the constants c_1 to c_n. The strategy uses the weak induction scheme by default but can be told to use strong induction by supplying `rel_induction` as the *induction* argument.

4.11.12 `simple-measure-induct/$`: Introduce induction scheme instance

syntax: `(simple-measure-induct measure vars` &OPTIONAL *`fnum[1] order`*`)`

effect: Selects and insert as an antecedent, an instance of measure induction with measure *measure* containing only free variables from *vars* using formula *fnum* to formulate an induction predicate. Uses *order* as the well-founded relation. If the order is not specified, it defaults to the `<` relation on nats or ordinals.

usage: `(simple-measure-induct "i+j" ("i" "j"))`: Inserts measure induction on the measure `i + j` in the variables `i` and `j`.

4.12 Simplification with Decision Procedures and Rewriting

`assert/$`	*defined*	uses decision procedures to assert sequent formulas
`bash/$`	*defined*	`assert`, `bddsimp`, `inst?`, `skolem-typepred`, `flatten`, `lift-if`
`both-sides/$`	*defined*	Applies operation to both sides of inequality chain
`do-rewrite/$`	*defined*	applies auto-rewrites
`grind/$`	*defined*	installs rewrites and repeatedly simplifies
`ground/$`	*defined*	propositional and ground simplification
`record/$`	*defined*	records assumptions for the decision procedures
`reduce/$`	*defined*	Repeated simplification
`simplify`	*primitive*	does Boolean simplification using decision procedures
`simplify-with-rewrites/$`	*defined*	installs rewrites, simplifies, removes rewrites
`smash/$`	*defined*	`ground` with lift-if

Version 2.3 features a new implementation of the ground decision procedures. The new implementation overcomes several deficiencies of the old ground decision procedures originally implemented by Rob Shostak. The original decision procedures are also included in Version 2.3 for the purposes of backward compatibility which might be necessary in some rare cases. For the most part, the new decision procedures should subsume the old ones. The PVS Emacs command

`M-x new-decision-procedures` switches from the old to the new decision procedures and `M-x old-decision-procedures` can be used to switch back to the old ones.

4.12.1 `assert`: Simplifying Using the Decision Procedures

syntax: (`assert` &OPTIONAL *fnums[*] rewrite-flag flush? linear? cases-rewrite? type-constraints?[T] ignore-prover-output?*)

effect: The `assert` rule is a combination of `record`, `simplify`, `beta`, and `do-rewrite`. The use of decision procedures for equalities and linear inequalities is perhaps the most significant and pervasive part of PVS. These procedures are invoked to prove trivial theorems, to simplify complex expressions (particularly definitions), and even to perform matching. These decision procedures, originally due to Shostak, employ congruence closure for equality reasoning, and they also perform linear arithmetic reasoning over the natural numbers and reals. They deal solely with *ground* formulas, namely those that contain no quantifiers. While they primarily deal with linear arithmetic, *i.e.*, expressions of the form `2*x + 3*y <= 4*z`, there are some modest extensions for dealing with expressions involving nonlinear subterms using simplifications such as `(x + y)*(x - y) = (x*x - y*y)` and simplifications involving division such as `x*(y/x) = y`.

The assert rule employs the decision procedures to either simplify formulas or to assert the formula to the data-structures employed by the decision procedures. The assert rule can have one of three effects for a given formula named in *fnums*:

1. It can have no visible effect on the formula but could have asserted the formula to the congruence closure data-structures employed by the decision procedures. Antecedent formulas are recorded as being true, and consequent formulas are recorded as being false. only those formulas that do not contain branching (`if` or `cases`) or propositional structure (unless they are within a quantifier or a `lambda` binding) are asserted into the database since such structures are likely to need further simplification before they can be processed by the decision procedures. Unless the *type-constraints?* is set to `NIL`, any subtype constraints on subexpressions of the formulas processed by `assert` are also automatically recorded by the decision procedures for the commands `assert`, `record`, `simplify`, and `do-rewrite`. The new assertions in the data-structures remain valid for any descendant proof node of the current sequent and are automatically employed when assert is invoked at the lower nodes.
2. If the decision procedures succeed in demonstrating a contradiction from the formula as asserted, then the entire sequent is regarded as being proved.

If there are no assertable or simplifiable formulas among those listed in *fnums*, then the rule behaves as a `(skip)`. In every remaining case, a subgoal is generated.

3. It can simplify the formula by carrying out boolean simplification, simplify `if`-expressions by attempting to reducing the condition part to TRUE or to FALSE, and rewrite expressions using the rewrite rules provided by the `auto-rewrite` and `auto-rewrite-theory` rules below (see `do-rewrite`). In order for any automatic rewrites to take effect, it must be the case that the conditions of the instance of the rewrite rule should all simplify to TRUE, as should any type correctness conditions generated by typechecking an instantiating term with respect to the type of the variable that it instantiates. As a simple check to prevent such rewrites from looping, if a rewrite rule rewrites l to r and r is either a `cases` or an `if` expression, then the top-most conditional of these expressions is treated as if it were another condition in a conditional rewrite rule. In other words, if r is of the form `if a then b else c endif`, then a must simplify to `true` or `false`, and similarly, if r has the form `cases e of` p_1 `:` c_1`, ...endcases` then e should match one of the patterns p_i. The simplifications carried out by `assert` also include various obvious datatype simplifications and all of the beta-reductions. the resulting simplified formula, if suitable, is asserted to the data-structures.

When the *rewrite-flag* is `lr`, only the right-hand side of any equality formula is simplified since simplifying the left-hand side of a formula to be used by `replace` in the process of rewriting (see strategies `rewrite` and `rewrite-lemma` below) could cause the replace to fail. correspondingly, when the *rewrite-flag* is `rl`, only the left-hand side of an equality formula is rewritten. If the *flush?* argument is `t`, then the existing database used by the decision procedures is flushed. This database can get fairly large in the course of a proof thereby decreasing the efficiency of the assert rule. The *flush?* flag should be used with caution. The decision procedures apply a modest amount of non-linear arithmetic simplification to multiplication and division expressions. This can sometimes get in the way. the *linear?* argument can be set to `t` in order to prevent such simplifications.

The `cases-rewrite?` must be set to `T` in order for simplification to occur within a branch of a CASES-expression.

The `type-constraints?` flag must be set to `NIL` to avoid asserting subtype constraints of subexpressions occurring in the sequent. If there are several large expressions with subtype constraints, this phase of simplification can be quite slow.

The ground prover when confronted with a non-convex assertion returns a disjunction of assertions that are equivalent to the input assertion. The simplification commands examine these outputs to see if every path through them yields a contradiction. This step can in some cases be expensive. The `ignore-prover-output?` flag can be set to NIL in order to avoid the examination of the ground prover outputs.

usage: `(assert)` : Same as `(assert *)`. proves, simplifies, or asserts all of the formulas in the sequent employing the decision procedures.

`(assert -1 lr)` : Simplifies the right-hand side of the first antecedent formula, since the *rewrite-flag* is set to `lr` (meaning "left-to-right").

`(assert (-1 3 4))` : Proves, simplifies, or asserts the formulas numbered `-1`, `3`, and `4` in the sequent.

`(assert + :flush? t)` : Flushes the existing database of assertions and asserts the consequent formulas in the current sequent.

errors: No error messages are generated.

notes: One significant point about `assert` is that it can be sensitive to the order in which the formulas are asserted. It is sometimes necessary to apply `assert` more than once in order to obtain the desired effect. For example, if the formula a is asserted before the formula b, but b is used to simplify a, then `assert` would have to be reapplied in order to effect this simplification. Another reason that `assert` might need to be repeated is that the subtype constraints on subexpressions of a formula are silently collected recorded after `assert` has processed the formula. Simplifications that rely on these constraints can be missed in the first pass of `assert`.

Another point about the `assert` rule is that is that from the point of view of the control mechanism used for strategies, `assert` almost always succeeds so that a strategy like `(repeat (assert))` is certain to get into an infinite loop. The typical way to get around this is to have `assert` follow some other step *step1* that is guaranteed to not repeat indefinitely, as in:

```
(repeat (try step1 (assert) (skip)))
```

The `try` strategy ensures that the `assert` step is invoked on the subgoals generated by *step1*.

4.12.2 bash/$: Assert, bddsimp, inst?, skolem-typepred, flatten, and lift-if

syntax: `(bash` &OPTIONAL *`if-match`* `[T]` *`updates?`*`[T]` *`polarity?`*`)`

effect: This command executes `assert`, `bddsimp`, `inst?`, `skolem-typepred`, `flatten`, and `lift-if`, in that order. This command is the core of the `reduce` command which in turn is the workhorse of `grind`.

As in the `inst?` command, the *if-match* option can be `NIL`, `T`, `ALL`, or `BEST` for no, some, all, or the best instantiation, respectively. Note that `inst?` precedes `skolem-typepred`, so that matches that are in the original sequent are preferred to those that are exposed by Skolemization. This has the drawback that quite often, it is better to instantiate following Skolemization. To avoid eager instantiation, `bash` should be invoked with *if-match* set to `NIL`.

If the *updates?* option is `NIL`, update applications are not if-lifted.

When the *polarity?* flag is `T`, the `inst?` command matches templates against complementary subexpressions.

4.12.3 `both-sides/$`: Applies Operation to Both Sides of Inequality Chain

syntax: `(both-sides` *op term* &OPTIONAL *fnum*[1]`)`

effect: Here *fnum* is used to find a chained conjunction of inequalities of the form `e1 <= e2 AND e2 <= e3 AND e3 <= e4`. If the *term* argument is `t`, the command replaces the above chain with `e1 op t <= e2 op t AND e2 op t <= e3 op t AND e3 op t <= e4 op t`. If the equivalence between this chain and the previous one doesn't simplify to `TRUE` using `assert` and `do-rewrite` with respect to the prelude theory `real_props`, then a proof obligation is generated.

4.12.4 `do-rewrite/$`: Applying auto-rewrites

syntax: `(do-rewrite` &OPTIONAL *fnums[*] rewrite-flag flush? linear? cases-rewrite? type-constraints?[T]*`)`

effect: This command is another fragment of `assert`. It is used to automatically carry out the rewrites specified by `auto-rewrite` and `auto-rewrite-theory`. This command uses simplification as defined by `assert` to discharge the hypotheses of conditional rewrite rules and any type correctness proof obligations that arise from the use of a rewrite rule, and also to simplify the rewritten result. The other arguments are as in `assert`.

usage: `(do-rewrite)` : Applies the rewrite rules to all the formulas in the sequent.

`(do-rewrite (-1 -3 2))` : Applies rewrite rules to the formulas `-1`, `-3`, and `2`.

notes: No new information is recorded into the data structures used by the decision procedures, except for the subtype constraints on subexpressions processed by `do-rewrite`.

4.12.5 `grind/$`: Install Rewrites and Repeatedly Simplify

syntax: `(grind` &OPTIONAL *defs* `[!]` *theories rewrites exclude if-match* `[T]` *updates?*`[T]` *polarity?*`)`

effect: This is a catch-all strategy that is frequently used to automatically complete a proof branch or to apply all the obvious simplifications till they no longer apply. The strategy first applies `install-rewrites` to install the given theories and rewrite rules along with all the relevant definitions in the given subgoal. It then applies `bddsimp` followed by `assert` (similar to `ground`) to carry out the first level of simplification. This is followed by `replace*` to carry out all the equality replacements. This is followed by `reduce` which repeatedly applies `bash` (which invokes `assert`, `bddsimp`, `inst?`, `skolem-typepred`, `flatten`, and `lift-if`) followed by `replace*`.

The options to `grind` can be used to carefully guide its behavior.

The *defs*, *theories*, *rewrites*, and *exclude* arguments are as in `install-rewrites`.

The *if-match* and *polarity?* are used to control heuristic instantiation using `inst?`. Note that by setting *if-match* to `NIL`, one can avoid the eager instantiation behavior of `grind`. A second `grind` can then be used to pick up the instantiations exposed by the first instantiation-free `grind`.

The *updates?* option is as in `bash` and `reduce`.

4.12.6 `ground/$`: Propositional and Ground Simplification

syntax: `(ground)`

effect: This command invokes propositional simplification followed by `assert`. It is useful in obtaining simplified forms of the cases arising from propositional simplification. These simplifications include those given by `assert`, namely, the various boolean, datatype, and arithmetic simplifications, beta-reduction, simplification using ground decision procedures, and rewriting with respect to the installed rewrite rules.

4.12.7 `record/$`: Recording assumptions for the decision procedures

syntax: (record &OPTIONAL *fnums[*] rewrite-flag flush? linear? cases-rewrite? type-constraints?[T] ignore-prover-output?*)

effect: The decision procedures maintain efficient data structures where the assumptions that are true in the current context are recorded. The record command is used to add more assumptions to these data structures. The only assumptions that can be recorded are those that do not contain any embedded `if`, `cases`, or boolean structure. The assumptions are antecedent formulas and negations of consequent formulas. It is possible for a `record` command to prove the sequent if the assumptions are found to be contradictory. The other arguments are as in `assert`.

usage: `(record)` : records all the assumption formulas in the sequent into the data structures used by the PVS decision procedures.

`(record (-1 -3 2))` : records the assumptions from formulas `-1`, `-3`, and `2`.

notes: a formula is simplified before it is recorded (see `simplify` and `assert` below) so that it is possible for `record` to record an assumption to contain `if`, `cases` or boolean structure that is eliminated by simplification. the command `assert` subsumes `record` but its behavior is more difficult to explain.

4.12.8 `reduce/$`: Repeated Simplification

syntax: (reduce &OPTIONAL *if-match* [T] *updates?*[T] *polarity?*)

effect: This command is the main workhorse of the `grind` command. It applies `bash` followed by `replace*` in a loop until neither command has any effect.

The *if-match* and *polarity?* options are as in `bash` and control the instantiation.

The *updates?* option is also as in the `bash` command and must be set to `NIL` in order avoid the automatic if-lifting of update applications.

4.12.9 `smash/$`: Propositional, Ground Simplification with IF-lifting

syntax: (smash &OPTIONAL *updates?* [T])

effect: This command is a more powerful version of `ground`. It is essentially an iterated application of `bddsimp` (whereas the propositional simplification in `ground` is similar to that of `prop`), `assert`, and `lift-if`.

The *updates?* flag can be set to `NIL` in order to avoid if-lifting update applications (see page 43).

4.12.10 `simplify`: Simplifying using decision procedures

syntax: `(simplify` &OPTIONAL *fnums[*] record? rewrite? rewrite-flag flush? linear? cases-rewrite? type-constraints?[T] ignore-prover-output?*`)`

effect: The commands `assert`, `do-rewrite`, `record`, and `simplify` used to all be primitive commands in PVS 1.x. In PVS 2, `simplify` is the only primitive command and it is used in the definition of `assert`, `do-rewrite`, and `record`. Thus `simplify` now includes the arguments to `assert` along with two flags: *record?*, and *rewrite?*. For the `assert` command, *record?* and *rewrite?* must be `t`. To get the `do-rewrite` command, *record?* must be `nil` and *rewrite?* must be `t`. To get the `record` command, *record?* must be `t`, and *rewrite?* must be `nil`. When *record?* and *rewrite?* are both `nil`, one gets the effect of the `simplify` command of PVS 1.x.

The other flags have already been documented with the `assert` command.

Simplification works by maintaining database of currently recorded information which is then used to simplify and record further information. The ground decision procedures can be used to decide if a given formula (that is, a boolean expression) is true or false (or not known to be either) with respect to the current database and relative to theories such as those of equality over uninterpreted function symbols and linear arithmetic. In a sequent of the form $a_1 \ldots a_m \vdash b_1 \ldots b_n$, the a_i are simplified and recorded as being true, and the b_i are simplified and recorded as being false. The simplifications are described below. The recording process can yield a refutation in which case the sequent has been proved.

The theories handled by the ground decision procedures include:

1. The theory of equality with uninterpreted functions symbols. This would enable it to prove a sequent of the form `x = f(x)` $\vdash$ `f(f(f(x))) = x`.
2. Quantifier-free linear arithmetic equalities and inequalities, *e.g.*,, `x < 2*y`, `y < 3*z` $\vdash$ `3*x < 18*z`. Note that `x`, `y`, and `z` are implicitly universally quantified.

3. Quantifier-free integer linear arithmetic, *e.g.*,, `i > 1, 2*i < 5` ⊢ `i = 2`. This procedure is incomplete since the decision problem for this theory is not known to be polynomial.
4. Arrays and functions with updates. Examples include:
   (a) ⊢ `f with [(s) := f(s)] = f`,
   (b) ⊢ `(f with [(s) := x])(s) = x`, and
   (c) `r/=s` ⊢ `(f with [(s) := x])(s) = f(s)`

The simplifications carried out by `simplify` are represented by means of ⟶, and include:

1. **beta reduction:** Examples of such redexes and the corresponding reductions are:
   - lambda redex: `(lambda x : x * x)(2)` ⟶ 2 * 2
   - record redex : `b((# a:= 1, b:= 2, c:= 3 #))` ⟶ `2`
   - tuple redex : `proj_2((1, 2, 3))` ⟶ `2`
   - function update redex: For function `f`,

     ```
     (f WITH [(i) := 3])(i)  ==>  3
     (f WITH [(0) := 3])(1)  ==>  f(1)
     ```

   - record update redex: For record `r`,

     ```
     a(r WITH [(a) := 3])  ==>  3
     a(r WITH [(b) := 2])  ==>  a(r)
     ```

   - datatype redex: `car(cons(1, null))` ⟹ `1`
   - recognizer redex: `cons?(null)` ⟹ `FALSE` and `cons?(cons(1, null))` ⟹ `TRUE`
   - subtype redex: `even?(i)` ⟹ `TRUE`, if `even?` is one of the subtype predicates in the type of `i`.
2. **Arithmetic simplifications:** If `a`, `b`, `c` are arbitrary arithmetic expressions, the following are examples of simplifications that are carried out by the `simplify` command :

   ```
           a + 0  ==>  a
         a + 2*a  ==>  3*a
       1 + a + 3  ==>  a + 4
       a*(b + c)  ==>  a*b + a*c
           0 * a  ==>  0
           1 * a  ==>  a
   a + b = b + c  ==>  a = c
   ```

In addition, sums and products are ordered into a canonical form. This ordering will affect certain PVS 1.x proofs since the forms of certain expressions in the sequent will now be different following simplification.

3. **Conditional simplification:** If `A` is a formula and `a` and `b` are expressions, the following are examples of simplifications applied by `simplify`:

```
 (IF TRUE THEN a ELSE b ENDIF)   ==>  a
(IF FALSE THEN a ELSE b ENDIF)   ==>  b
    (IF A THEN b ELSE b ENDIF)   ==>  b
    (IF A THEN a ELSE b ENDIF)   ==>  (IF A' THEN a' ELSE b')
```

   where: $\mathtt{A} \Longrightarrow \mathtt{A}'$
   $\mathtt{a} \Longrightarrow \mathtt{a}'$ assuming $\mathtt{A}'$
   $\mathtt{b} \Longrightarrow \mathtt{b}'$ assuming $\mathtt{NOT\ A}'$

```
(CASES null OF null: a, cons(x,y): b ENDCASES)  ==>  a
```

4. Datatype simplifications:

$$\mathtt{cons?(a)} \Longrightarrow \mathtt{TRUE},\ \ \mathtt{if}\ \ \mathtt{null?(a)} \Longrightarrow \mathtt{FALSE}$$
$$\mathtt{cons?(a)} \Longrightarrow \mathtt{FALSE},\ \ \mathtt{if}\ \ \mathtt{null?(a)} \Longrightarrow \mathtt{TRUE}$$

   The datatype simplifications might look circular but the simplifier uses the decision procedures to check for each recognizer applied to an expression having a datatype as a type, whether the result is true or false or unknown.

5. Boolean simplification: If the decision procedures determine a boolean expression to be `TRUE` or `FALSE` in the logical context in which the expression occurs, then it is simplified accordingly. If `A` is a boolean expression, the following are examples of other simplifications carried out by `simplify`:

```
        A AND TRUE   ==>  A
       A AND FALSE   ==>  FALSE
         A OR TRUE   ==>  TRUE
        A OR FALSE   ==>  A
    TRUE IMPLIES A   ==>  A
   FALSE IMPLIES A   ==>  TRUE
       NOT (NOT A)   ==>  A
          NOT TRUE   ==>  FALSE
         NOT FALSE   ==>  TRUE
             a = a   ==>  TRUE
  (FORALL x: TRUE)   ==>  TRUE
 (EXISTS x: FALSE)   ==>  FALSE
```

6. Quantifier simplifications: some quantified expressions are now simplified, including the following examples.

```
      (EXISTS x: x = 5)  ==>  TRUE
 (EXISTS x, y, z: x = y + z AND f(x, y, z))
                          ==>  (EXISTS y, z: f(y + z, y, z))
  (EXISTS (x: T): TRUE)   ==>  TRUE
 (FORALL (x: T): FALSE)   ==>  FALSE
```

   The last two simplifications only happen when the type `T` is know to be nonempty.

7. Rewriting: The simplifications include conditional rewriting with respect to the rewrite rules installed by means of `auto-rewrite`, `auto-rewrite!`, `auto-rewrite-theory`, `auto-rewrite-theories`, etc. The current set of rewrite rules can be viewed using the PVS Emacs command `M-x show-auto-rewrites`. Rewriting only occurs when the `rewrite` flag for the `simplify` command is `T`. If A and B are boolean expressions, and a and b are expressions, then rewrite rules can be of one of the following forms:

   (a) $a = b$
   (b) A `IMPLIES` $a = b$
   (c) A `IMPLIES` B

   In cases 1 and 2, the left-hand side (lhs) is a and the right-hand side (rhs) is b. In cases 2 and 3, the condition of the rewrite rule is A, whereas for case 1, the condition is `TRUE`. In case 3, the lhs is B and the rhs is `TRUE`.

   If a' is an instance of the lhs of a rewrite rule so that b' is the corresponding instance of the rhs and A' is an instance of the condition, then rewriting simplifies a' to b'' provided A' simplifies to `TRUE` and b' simplifies to b''. If b' is of the form `(IF` B `THEN` c `ELSE` d `ENDIF)`, then B must simplify to either `TRUE` or `FALSE` for the rewrite rule to be applicable, unless the rewrite rule has been installed with the `always?` flag set to `T`. The same constraint also applies if b' is a `CASES` expression. In applying a rewrite rule, the lhs of the rewrite rule, say a, is matched against the expression to be simplified. If the match succeeds, the typechecking of the instantiation could generate TCCs. The simplification process is applied to the TCCs which must simplify to `TRUE` before the rewrite rule is applied.

usage: `(simplify)` : Tries to simplify all the formulas in the sequent using the PVS decision procedures, beta-reduction, boolean simplification, datatype simplification, arithmetic simplification, and rewriting.

`(simplify (-1 -3 2))` : Simplifies the formulas `-1`, `-3`, and `2`.

`(simplify (-1 -3 2) :flush? T)` : Flushes the decision procedure database and then simplifies the formulas `-1`, `-3`, and `2`. This database can sometimes contain information that interferes with the expected simplification and it helps to flush the database.

`(simplify (-1 -3 2) :linear? T)` : The PVS decision procedures have a modest ability to handle nonlinear multiplication and division. This invocation of `simplify` causes simplification to occur with this capability turned off.

`(simplify -1 :rewrite-flag RL)` : The rewrite flag can take on values LR ("left-to-right") or RL ("right-to-left") to indicate that left-hand side (for LR) or right-hand side (for RL) of the indicated formula should be left undisturbed by simplification. This is needed in case the next step involves a syntactic replacement.

notes: Unlike PVS 1.x, the `simplify` commands and their variants can be placed within `repeat` strategies without the danger of creating an infinitely looping strategy.

The command `track-rewrite` can be used to obtain diagnostic information about the progress of rewriting, and the command `untrack-rewrite` can be used to turn off the printing of this information.

4.13 Installing and Removing Rewrite Rules

`auto-rewrite`	*primitive*	declares a list of formulas as rewrite rules
`auto-rewrite!`	*defined*	declares a list of formulas as eager rewrite rules
`auto-rewrite!!`	*defined*	declares a list of formulas as macro rewrite rules
`auto-rewrite-defs`	*defined*	declares relevant definitions as rewrite rules
`auto-rewrite-explicit`	*defined*	declares relevant definitions as eager rewrite rules
`auto-rewrite-theories`	*defined*	declares theories to be used as rewrite rules
`auto-rewrite-theory`	*defined*	declares a theory to be used as rewrite rules
`auto-rewrite-theory-with-importings`	*defined*	declares a theory and its importings to be used as rewrite rules
`install-rewrites`	*defined*	declares rewrite rules and rewrite theories
`stop-rewrite`	*primitive*	disables auto-rewrite rules
`stop-rewrite-theory`	*defined*	turns off auto-rewrites from a theory

4.13.1 `auto-rewrite`: Installing Automatic Rewrite Rules

syntax: `(auto-rewrite` &REST *names*`)`

effect: The definitions, lemmas, and antecedent formulas named (or numbered) in *names* are made available for the descendant nodes of the current proof node as automatic rewrite rules to be used for simplification by the `assert` rule. Only formulas of a certain special form can be considered as automatic rewrite rules. An automatic rewrite rule is either:

- An *equality rewrite rule* of the form $l = r$ or $l \iff r$ or an atomic Boolean proposition, where l is any expression that is not an individual variable, and the set of free variables in r is a subset of the free variables of l
- An unquantified conditional rewrite rule of the form $A \supset B$ where B is either an equality or an unquantified conditional rewrite rule and the set of free variables in A is a subset of the free variables in B, or
- A quantified rewrite rule of the form $(\forall x_1, \ldots, x_n \colon A)$, where A has the form of a rewrite rule.

In addition, a generic rewrite rule that is installed without actual theory parameters must be such that all the formal parameters occur in the left-hand side expression l.

All rewrite rules have a left-hand side l, a right-hand side r, and a condition H so that any matching instance l' of the left-hand side is replaced by the corresponding instance r' of the right-hand side provided the corresponding instance H' of the condition can be simplified to `TRUE`. (H is usually the left-hand side of an implication, e.g. $H \supset l = r$, or the condition of an IF or $CASES$ expression.)

There are three kinds of automatic rewrite rules: *lazy*, *eager*, and *macros*. At a given sequent, each installed rewrite rule can belong to at most one of these classes.

There are two forms of the *names* argument. In the preferred form, each name is a *rewrite-name-or-fnum*:

rewrite-name-or-fnum	::=	*fnum* | *rewrite-name*
fnum	::=	[`-`] *Number* [`!` [`!`]]
rewrite-name	::=	*Name* [`!` [`!`]] [`:` { *TypeExpr* | *FormulaName* }]

Here a lazy rule has no exclamation marks, an eager rule has one, and a macro has two. For a rewrite name, a type expr or formula name allows a specific rewrite to be specified in the presence of overloading.

In the second form of *names* argument some of the names may be parenthesized, and the depth of parenthesization indicates whether it is lazy, eager, or a macro. If the name appears as unparenthesized as `"assoc"`, then it is lazy. If it is singly parenthesized, as in `("assoc")`, then it is eager. If it is doubly parenthesized, as in `(("assoc"))`, then it is a macro. The single and doubly parenthesized forms can include multiple unparenthesized names. If a given name is overloaded in a theory, there is no way in this form to indicate a specific declaration.

For example, for the command

```
(auto-rewrite "A" "B!" "1!" "C" "-3!!" "D!!")
```

`A` and `C` are lazy rewrites, `B` and formula number `1` are eager rewrites, and formula number `-3` and `D` are macro rewrites. In the deprecated form, this may be given as

```
(auto-rewrite "A" ("B" "1") "C" (("-3" "D")))
```

The two forms may not be mixed, and the parenthesized form may be disallowed in the future.

Lazy rules are the default where if the right hand side is a conditional or `CASES` expression, the rewrite rule is not triggered unless the top-level condition simplifies to `TRUE` or `FALSE` or the top-level `CASES` expression is resolved. All recursive definitions can only be lazy rewrite rules since there is a possibility that the rewriting might loop following the recursion. In an eager rewrite rule, the rewrite rule is applied regardless of the consequence of simplifications on the right-hand side instance. When rewriting with function definitions, both lazy and eager rewrite rules work with left-hand sides that, when curried, are in their fully applied form. That is, if a function definition allows the left-hand side forms f, $f(x, y)$, $f(x, y)(z)$, then $f(x, y)(z)$ is the only valid left-hand side to a lazy or eager rewrite rule form of this definition. Macros on the other hand always rewrite any occurrence of f so that f is rewritten to $(\lambda(x, y) \colon (\lambda z \colon \ldots))$ and $f(a, b)$ is rewritten to $(\lambda z \colon \ldots)[a/x, b/y]$.

It is preferable that the names of lemmas from imported theories be completely specified, *i.e.*, with all the actual theory parameters explicitly given. Otherwise, the rewrite rule is known as *generic* and the actual parameters are instantiated when the left-hand side is matched. Not all rewrite rules contain instances of all the actuals, and such rewrite rules are not installed in their generic form.

For rewrite rules installed from antecedent formulas, an internal name is generated that can be used for turning off the rewrite rule using `stop-rewrite` or one of its variants.

It is important to note that `auto-rewrite` has no visible effect on the sequent, but it affects the subsequent behavior of `assert` in any lower branch of the proof. The scope of an `auto-rewrite` declaration is restricted to the branch of the proof below it.

usage: `(auto-rewrite "append[nat]" "append[int]" "length[nat]")`

errors: Due to the presence of actual parameters, `auto-rewrite` can generate parsing and typechecking errors, in addition to those listed below.

No resolutions for ...: The system was unable to find declarations corresponding to the given names.

Can't rewrite using ...: LHS key ... is bad. Rewrite rules are arranged by a key which in the case of an application is the left-most operator name or expression type such as a record constructor, tuple constructor, update expression, tuple projection, record field access, cases expression, and lambda, forall, and exists expressions. Note that having too many rewrite rules attached to a single key can slow down rewrite lookup.

RHS free variables must be contained in the LHS free variables: Since the matching for rewriting is done using the left-hand side of the rewrite rule, there should be no free variables left in the formula that are not instantiated during such a match.

Hypothesis free variables must be contained in the LHS free variables: See explanation above.

Theory ... is generic; No actuals given; Free parameters in the LHS of rewrite must contain all theory formals.: Rewrite rules from generic theories are allowed as long as it is possible to extract the actual parameters by matching against the left-hand side.

notes: See the rule `stop-rewrite` for turning off an automatic rewrite rule, `auto-rewrite-theory` for turning an entire theory into rewrite rules, and `stop-rewrite-theory` to turn off theory rewriting. The Emacs command `M-x show-auto-rewrites` displays the currently active rewrite rules in a separate buffer. The `track-rewrite` command can be used to explain why rewrite rules did not perform the expected rewriting. A rewrite rule that is already installed is not affected by `M-x modify-declaration` and might therefore need to be reinstalled.

4.13.2 `auto-rewrite!/$`: Installing Eager Automatic Rewrite Rules

syntax: `(auto-rewrite!` &REST *names*`)`

effect: This is just a macro for the eager case of `auto-rewrite`. The convenience it offers over `auto-rewrite` is that the arguments can be given in &REST form.

4.13.3 `auto-rewrite!!`: Installing Automatic Macros

syntax: `(auto-rewrite!!` &REST *names*`)`

effect: This is just an abbreviation for the macro case of `auto-rewrite` where the arguments can be given in &REST form.

4.13.4 `auto-rewrite-defs/$`: Installing Relevant Definitions as Rewrites

syntax: `(auto-rewrite-defs` &OPTIONAL *explicit? always? exclude-theories*`)`

effect: Installs all the definitions used directly or indirectly in the current sequent as auto-rewrite rules. If the *explicit?* flag is T, the recursive definitions are not installed and only the explicit definitions are. If *always?* is `!!`, the explicit definitions are installed as macros. If this flag is `T`, then the explicit definitions are installed as eager rewrite rules. Otherwise, all definitions are installed as lazy rewrite rules. (See `auto-rewrites`.)

The *exclude-theories* takes a list of theories and the definitions in these theories will not be expanded.

The `install-rewrites` command should always be preferred over any of the specialized rewrite installation commands.

4.13.5 `auto-rewrite-explicit/$`: Installing Relevant Definitions as Rewrites

syntax: `(auto-rewrite-explicit` &OPTIONAL *always?*`)`

effect: Installs all and only the explicit definitions used directly or indirectly in the current sequent as auto-rewrite rules. If *always?* is `!!`, the explicit definitions are installed as macros. If this flag is `T`, then the explicit definitions are installed as eager rewrite rules. Otherwise, all definitions are installed as lazy rewrite rules. (See `auto-rewrite`.)

The `install-rewrites` command should always be preferred over any of the specialized rewrite installation commands.

4.13.6 `auto-rewrite-theories/$`: Automatic Rewriting Relative to Theories

syntax: `(auto-rewrite-theories` &REST *`theories`*`)`

effect: Applies the `auto-rewrite-theory` command to each of the theories in the list *theories*. Each entry in the list of theories can be a theory name with or without actuals or a list of arguments in the form accepted by `auto-rewrite-theory`.

This command is subsumed by `install-rewrites`.

4.13.7 `auto-rewrite-theory/$`: Automatic Rewriting Relative to a Theory

syntax: `(auto-rewrite-theory` *`name`* &OPTIONAL *`exclude defs always? tccs?`*`)`

effect: Installs an entire theory or only (explicit) definitions if *defs* is `T` (`explicit`) as auto-rewrites. In the case of a parametric theory, unless the *defs* flag is `T` or `explicit`, the actual parameters must be given. If *always?* is `T` the rewrites are installed so that any rewrite other than a recursive definition always takes effect (see `auto-rewrite!`). If *always?* is set to `!!`, then the non-recursive definitions are always rewritten even when only a few of the curried arguments have been provided.) Declarations named in *exclude* are not introduced and any current rewrite rules in the *exclude* list are disabled. By default, TCCs in the theory are excluded but they can be included when the *tcc?* flag is `T`.

usage: `(auto-rewrite-theory "sets[nat]" "sets[rational]" "list_adt[nat]")` : Declares all those definitions and formula from the listed modules that can be viewed as rewrite rules to be automatic rewrite rules.

`(auto-rewrite-theory "agreement")` : If we are proving a lemma names `main` in the theory `agreement`, then all the rewrite rules preceding `main` are declared as automatic rewrite rules to be used by `assert`.

errors: Apart from parsing and typechecking errors, the following error messages are possible:

Could not find theory ...: The named theory does not appear in the current context. Add the theory to the `USING` list if needed.

... is not a fully instantiated theory: Provide the relevant actual parameters.

4.13.8 `auto-rewrite-theory-with-importings/$`: Automatic Rewriting Relative to a Theory and Its Importings

syntax: `(auto-rewrite-theory-with-importings` *`name`* &OPTIONAL *`exclude-theories importchain? exclude defs always? tccs?`*`)`

effect: Installs rewrites in theory *name* along with any theories imported by *name*. The full import chain of theories can be installed by supplying the *importchain?* flag as `T`. Theories named in *exclude-theories* are ignored. The other arguments are similar to those of `auto-rewrite-theory` and apply uniformly to each of the theories to be installed.

errors: Same as for `auto-rewrite-theories`

4.13.9 `install-rewrites/$`: Install Rewrite Rules from Names and Theories

syntax: `(install-rewrites` &OPTIONAL *`defs theories rewrites exclude-theories exclude`*`)`

effect: This is the most powerful way to install rewrite rules and essentially subsumes all the other ways of installing rewrite rules.

The *defs* argument can be

- `NIL`: To avoid installing the definitions relevant to the current sequent.
- `T`, `!`, or `!!`: To install all definitions as lazy, eager, or macro rewrites, respectively.
- `explicit`, `explicit!`, or `explicit!!`: Only the explicit definitions are installed as lazy, eager, or macro rewrites, respectively.

The *theories* argument is a list of theories whose declarations are meant to be used as rewrite rules. Each entry in the list is either a theory name, with or without actual parameters, or an argument list in the format expected by `auto-rewrite-theory`.

The *rewrites* argument is a list of names of rewrite rules to be installed.

The *exclude-theories* argument is a list of theories that have to be excluded when installing rewrite rules from theories.

The *exclude* argument is a list of names of rewrite rules that are removed from the list of installed rewrite rules.

4.13.10 stop-rewrite: Turning Off Rewrite Rules

syntax: (stop-rewrite &REST *names*)

effect: Turns off those automatic rewrite rules named in *names* that were turned on by either auto-rewrite or auto-rewrite-theory.

usage: (stop-rewrite "append[nat]" "append[int]" "length[nat]") : Turns off automatic rewriting of these specific rewrite rules regardless of whether they were added using auto-rewrite or auto-rewrite-theory.

errors: . . . is not an auto-rewrite: This is a helpful message rather than an error.

4.13.11 stop-rewrite-theory/$: Turning Off Rewrite Rules from a Theory

syntax: (stop-rewrite-theory &REST *names*)

effect: Turns off any rewrite rules from the theories named in *names* that were turned on by either auto-rewrite or auto-rewrite-theory. Any theory listed in *names* must either be fully instantiated, or name a defined constant, or be the theory for the current proof. Note that for a rewrite rule to be turned off, the name should be given in the same form to stop-rewrite as in the corresponding auto-rewrite. command

usage: (stop-rewrite-theory "sets[nat]" "sets[rational]" "list_adt[nat]") : Turns off any rewrite rules in the listed theories. As with auto-rewrite-theory, the theory names should be fully instantiated.

errors: Same as auto-rewrite-theory.

4.14 Making Type Constraints Explicit

typepred	*primitive*	introduces type constraints on expressions
typepred!	*primitive*	minor variant of typepred

4.14.1 typepred: Making Implicit Type Constraints Explicit

syntax: (typepred &REST *exprs*)

effect: If *exprs* is a list of expressions $e_1, \ldots e_n$, then for each e_i, and for each type-constraint predicate p in the type of e_i, an antecedent formula of the form $p(e_i)$ is introduced. A predicate p is a type-constraint predicate in a type $\{x\colon \tau | q(x)\}$ if either $p \equiv q$ or p is a type-constraint predicate in τ.

usage: `(typepred "i+j")` : If `i` and `j` are natural numbers, then the antecedent formula `i + j >= 0` is added to the current sequent along with the assertions `integer_pred(i + j)`, `rational_pred(i + j)`, and `real_pred(i + j)`.

`(typepred "cons(i, null)")` : Adds the antecedent formula `cons?[nat](cons(i, null))` to the current sequent.

errors: Apart from parsing and typechecking errors, `typepred` can generate two errors:

Given expression does not typecheck uniquely: This means that there was some type ambiguity in the given expression that can be resolved by providing theory names, actuals, and/or explicit coercions.

Irrelevant free variables in . . . : As with many other rules, no free variables can be introduced into a sequent in a PVS proof.

notes: Type predicates are automatically made available to the decision procedures in most cases; this command is only occasionally needed.

4.14.2 `typepred!`: Making Implicit Type Constraints Explicit

syntax: `(typepred!` *`exprs`* &OPTIONAL *`all?`*`)`

effect: The only difference between `typepred` and `typepred!` is that the *exprs* argument is not given in &REST form and the `all?` flag can be set to `T` to get all the type predicates for subtypes of number. The `typepred` command only returns the type predicates up to the natural number constraint.

4.15 Abstraction and Model Checking

`abstract-and-mc`	*defined*	Boolean/data abstraction followed by model-checking
`abstract`	*defined*	Boolean abstraction of PVS expressions
`abs-simp`	*primitive*	Boolean abstraction
`model-check`	*defined*	CTL model checking
`musimp`	*primitive*	mu-calculus model checker

4.15.1 `abstract-and-mc/$`: Boolean abstraction followed by model-checking

syntax: (abstract-and-mc *cstate astate amap* &OPTIONAL *theories rewrites exclude strategy* [(assert)] *feasible verbose?*)

effect: This command constructs an abstraction of the mu-calculus formulas in the given goal using `install-rewrites` to install various rewrites, `assert` to apply these rewrites and other simplifications, and `abs-simp` to actually construct the abstraction.

The mu-calculus formulas in the goal that contain quantification over the concrete state type *cstate* are abstracted with respect to both predicate and data abstraction maps given in *amap*. The result is a corresponding mu-calculus formula over the abstract state type *astate*. Both *cstate* and *astate* are expected to be type expressions, and *amap* is a list of pairs of field-name (from the record type *astate*) and a function from the concrete state type *cstate* to the type (which currently must be a boolean or a scalar type) corresponding to the field-name in the type *astate*. The optional arguments *theories*, *rewrites*, and *exclude* are as in `install-rewrites`. The *strategy* argument takes a proof command that is used to discharge the proof obligations that arise in the construction of the abstraction. The default strategy is `(assert)`. Various forms of `grind` are also suitable though significantly more expensive in terms of time. The `feasible` argument takes a predicate in the abstract state *astate* that characterizes the feasible abstract states. This is needed when the abstracted formulas contain quantifiers of existential strength. Each abstract state corresponds to a set of concrete states, but the latter set might be empty leading to an existential formula that is satisfiable at the abstract level but not at the concrete level. Finally, the *verbose?* flag prints out an extensive listing of the proof obligations generated during abstraction and the success or failure of the proof effort.

4.15.2 `abstract/$`: Abstraction of PVS expressions

syntax: (abstract *cstate astate amap* &OPTIONAL *theories rewrites exclude strategy* [(assert)] *feasible verbose?*)

effect: This command invokes the command `abstract` to construct an abstraction of the mu-calculus formulas in the given goal.

The mu-calculus formulas in the goal that contain quantification over the concrete state type *cstate* are abstracted with respect to both predicate and data abstraction maps given in *amap*. The result is a corresponding mu-calculus formula over the abstract state type *astate*. Both *cstate* and *astate* are expected to be type expressions, and *amap* is a list of pairs of field-name (from the record type *astate*) and a function from the concrete state type *cstate* to the type (which currently must be a boolean or a scalar type) corresponding to the field-name in the type *astate*. The optional arguments *theories*, *rewrites*, and *exclude* are as in `install-rewrites`. The *strategy* argument takes a proof command that is used to discharge the proof obligations that arise in the construction of the abstraction. The default strategy is `(assert)`. Various forms of `grind` are also suitable though significantly more expensive in terms of time. The `feasible` argument takes a predicate in the abstract state *astate* that characterizes the feasible abstract states. This is needed when the abstracted formulas contain quantifiers of existential strength. Each abstract state corresponds to a set of concrete states, but the latter set might be empty leading to an existential formula that is satisfiable at the abstract level but not at the concrete level. Finally, the *verbose?* flag prints out an extensive listing of the proof obligations generated during abstraction and the success or failure of the proof effort.

4.15.3 `abs-simp/$`: Boolean abstraction

syntax: `(abs-simp` *`cstate astate amap`* &OPTIONAL *`strategy`* `[(assert)]` *`feasible verbose?`* `)`

effect: This is the primitive proof command used to construct an abstraction of the mu-calculus formulas in the given goal.

The mu-calculus formulas in the goal that contain quantification over the concrete state type *cstate* are abstracted with respect to both predicate and data abstraction maps given in *amap*. The result is a corresponding mu-calculus formula over the abstract state type *astate*. Both *cstate* and *astate* are expected to be type expressions, and *amap* is a list of pairs of field-name (from the record type *astate*) and a function from the concrete state type *cstate* to the type (which currently must be a boolean or a scalar type) corresponding to the field-name in the type *astate*. The *strategy* argument takes a proof command that is used to discharge the proof obligations that arise in the construction of the abstraction. The default strategy is `(assert)`. Various forms of `grind` are also suitable though significantly more expensive in terms of time. The `feasible` argument takes a predicate in the abstract state *astate* that characterizes the feasible abstract states. This is needed when the abstracted formulas contain quantifiers of existential strength. Each abstract state corresponds to a set of concrete states, but the latter set might be empty leading to an existential formula that is satisfiable at the abstract level but not at the concrete level.

Finally, the *verbose?* flag prints out an extensive listing of the proof obligations generated during abstraction and the success or failure of the proof effort.

4.15.4 `model-check/$`: CTL Model Checking

syntax: `(model-check` &OPTIONAL *`dynamic-ordering?`*`[T]` *`cases-rewrite?`*`[T]` *`defs theories rewrites exclude irredundant?`*`)`

effect: This command is still quite experimental. It has the effect of rewriting with respect to the theories that define the CTL operators in terms of the mu-calculus, and then applying `musimp`, the mu-calculus model checker to the result.

The *dynamic-ordering?* flag can be set to `NIL` to turn off the dynamic reordering of variables in order to reduce BDD size.

The *cases-rewrite?* flag can be set to `NIL` to avoid rewriting and simplification within unresolved selections within a `CASES` expression for the sake of efficiency.

The commands *defs*, *theories*, *rewrites*, and *exclude*, are used exactly as in `install-rewrites` to set up rewrite rules for use in simplification prior to model checking.

The model checker invoked by `musimp` uses the same BDD package as the `bddsimp` command. The results of Boolean simplification and model checking are returned in sum-of-products form as a disjunction of conjunction of literals. Some of the disjuncts might be redundant but generating a minimal set of disjunctions is expensive. The flag *irredundant?* when set to `T` allows a less expensive redundant sum-of-products to be returned as the result.

The model checker either verifies the goal, returns a collection of subgoals that serve as counterexamples to the given mu-calculus assertion, or gives an indication that the result cannot be translated. The counterexamples correspond to the set of states for which the mu-calculus assertion fails.

usage: `(model-check :theories ("transitions" "properties") :irredundant? T)` : Translates the given sequent containing CTL or mu-calculus assertions into a Boolean mu-calculus, invokes a BDD-based symbolic model checker on this, and either proves the result or returns a collection of subgoals.

`(model-check :dynamic-ordering? NIL)` : Invokes CTL/mu-calculus model checking procedure on the subgoal but with dynamic reordering of BDDs disabled. The dynamic reordering tries to reduce the size of the BDDs but can be expensive in terms of time.

4.15.5 `musimp`: Mu-Calculus Model Checking

syntax: `(musimp` &OPTIONAL *fnums* `[*]` *dynamic-ordering?*`[NIL])`

effect: This command is primarily used in the `model-check` strategy to invoke the BDD-based, symbolic mu-calculus model checker. A glaring weakness of this model checker is that it does not generate a counterexample trace. The outcome of the command is usually a collection of subgoals corresponding to the states that violate the mu-calculus formula given by the sequent formulas selected using *fnums*.

4.16 Converting Strategies to Rules

`apply`	*primitive*	applies a proof strategy in a single atomic step

4.16.1 `apply`: Making Proof Strategies Atomic

syntax: `(apply` *strategy* &OPTIONAL *comment save? time?*`)`

effect: The `apply` rule takes an application of a proof *strategy* and applies it as a single atomic step that generates those subgoals left unproved by the proof strategy. The `apply` rule is frequently used when one wishes to employ a proof strategy but is not interested in the details of the intermediate steps. A number of defined rules employ `apply` to suppress trivial details. The optional *comment* field can be used to provide a format string to be used as commentary while printing out the proof. If the `save?` flag is set to `T`, the `apply` step is saved even if the applied strategy results in no change to the proof. This is useful if, for example, the command within the apply uses the `lisp` command to change a Lisp variable for use elsewhere in the proof. The `time?` flag when `T` is used to return timing information regarding the applied step.

usage: `(apply (then* (skolem 2 ("a4" "b5")) (beta) (flatten) "Skolemizing and beta-reducing")` : The `then*` strategy performs each of the steps given by its arguments in sequence. Wrapping this strategy in an `apply` ensures that the intermediate steps in the sequence are hidden. The given commentary string is printed out as part of the proof.

`(apply (try (skolem!) (flatten) (ground)))` : This applies a strategy that applies `(skolem!)` to the current goal, and if that "succeeds," applies `(flatten)` to the resulting subgoals, and otherwise it applies `(ground)` to the current goal. The above rule carries out this strategy in an atomic step and returns the resulting subgoals.

`(apply (grind) :save? T :time? T)` Applies the `grind` strategy but saves the step even when `grind` has no effect, and returns timing information.

errors: No error messages are generated by apply.

Chapter 5

Proof Strategies

We have so far described the primitive proof rules employed by PVS to construct proofs. Since it would be moderately tedious to construct proofs using only the primitive proof rules, there is a simple language for defining more powerful proof rules and proof strategies for combining proof rules. A *proof strategy* is intended to capture patterns of inference steps. A *defined proof rule* is a strategy that is applied in a single atomic step so that only the final effect of the strategy is visible and the intermediate steps are hidden from the user. There are four basic forms for constructing strategies or proof rules: recursion, let, backtracking, and the conditional form. Lisp expressions can be employed in constructing strategies as shown below. There is a special purpose interpreter for strategy expressions. An advanced user would need to study the interpreter. The crucial aspect of PVS commands is that the arguments are not evaluated. We use a substitution model of evaluation. Lisp code can only appear in the conditional of an `if` strategy and in the bindings of a `let` strategy.

5.1 Global Variables used in Strategies

The following global variables are kept current with each proof state and can be used within strategies.

`*ps*`	Current proof state
`*goal*`	Goal sequent of current proof state
`*label*`	Label of current proof state
`*par-ps*`	Current parent proof state
`*par-label*`	Label of current parent
`*par-goal*`	Goal sequent of current parent
`*+*`	Consequent sequent formulas
`*-*`	Antecedent sequent formulas
`*new-fmla-nums*`	Numbers of new formulas in current sequent
`*current-context*`	Current typecheck context
`*module-context*`	Context of current module
`*current-theory*`	Current theory

5.2 Data Structures

We now document the various operations on PVS data structures for terms, formulas, and proof goals that are needed for writing nontrivial PVS proof strategies. PVS data structures are defined as classes in the Common Lisp Object Sysem (CLOS). Each class is defined by indicating its slots. Classes can be defined as subclasses of one or more *superclasses* by introducing the additional slots. For example, the proof state that is the root node of a proof is defined as a subclass of an ordinary proof state that contains an extra slot for referring to the formula declaration corresponding to the proof. Data objects corresponding to a class are called *instances*. If a Lisp term t has instance v as its value, then `(show` t`)` displays the slot values of v. With PVS data structures, if value v is an instance of class `c`, then `c?` is the recognizer corresponding to the class so that `(c?` v`)` is `T`. Furthermore, if `c` is a subclass of class `b`, then `(b?` v`)` is also `T`. If `s` is a slot name in class `c`, then `(s` v`)` returns the corresponding slot value in v. A slot value is destructively updated by `(setf (s` v`)` u`)`, which sets the slot value of slot `s` in v to u. An instance can be nondestructively copied and updated by `(copy` v `'s1` u_1 `'s2` u_2`)`, which returns a copy of v with slot `s1` set to u_1 and `s2` set to u_2. There is a lazy form of copy where `(lcopy` v `'s1` u_1 `'s2` u_2`)` creates a new copy only when the updates actually change the slot values.

The class `proofstate` of proof states consists of a number of slots. These slots include:

`label`	Displayed label of proof state
`current-goal`	Sequent part of proof state
`current-rule`	Rule being applied to current proof state
`alist`	Decision procedure data structures
`done-subgoals`	List of proof states of completed subgoals
`pending-subgoals`	List of processed but incomplete subgoals
`remaining-subgoals`	List of unprocesses subgoals
`current-subgoal`	Proof state currently being processed
`subgoalnum`	Number of current proof state as subgoal of parent
`context`	Current typecheck context
`parent-proofstate`	Parent proof state
`justification`	Proof of subtree
`current-auto-rewrites`	Current rewrite rules

Only a few of these are really relevant for writing strategies, and these are typically the ones that are already captured in global variables.

The global variable `*ps*` is always bound to the currently active proof goal. Each proof goal is an instance of class `proofstate`. The sequent corresponding to the proof goal is kept in the global variable `*goal*` and appears in the `current-goal` slot of the proofstate. The current sequent is an instance of the class `sequent` which has the slots:

`s-forms`	List of active sequent formulas
`hiddens-forms`	List of hidden sequent formulas

A sequent formula is of class `s-formula` and the main slot here is `formula` so that if `sf` is a sequent formula, `(formula sf)` is the expression corresponding to the formula. This expression is a negation in the case of an antecedent formula.

Typical formulas are either negations, disjunctions, conjunctions, implications, equalities, equivalences, conditional expressions, arithmetic inequalities, or universally or existentially quantified expressions. Quantified expressions are in the class `binding-expr` with slots `bindings` which returns the bound variables, and `expression`, which returns the body of the binding expression. The other forms are all instances of the `application` class consisting of a slot for the `operator` and one for the `argument`. The first or only argument of an application `expr` can be obtained by `(args1 expr)`. The second argument, if any, can be obtained by `(args2 expr)`. The predicates for recognizing the different connectives are summarized in the following table.

Connective	**Recognizer Form**
Negation	`(negation? expr)`
Disjunction	`(disjunction? expr)`
Conjunction	`(conjunction? expr)`
Implication	`(implication? expr)`
Equality	`(equality? expr)`
Equivalence/Equality	`(iff? expr)`
Conditional	`(branch? expr)`
Universal Formula	`(forall-expr? expr)`
Existential Formula	`(exists-expr? expr)`

5.3 Selecting Sequent Formulas

Several Lisp functions select sequent formulas given their labels or numbers, or collect the numbers of selected sequent formulas. Given a sequent `seq`, typically obtained by `(s-forms *goal*)` and a list of labels or formula numbers `fnums`, the Lisp expression `(select-seq seq fnums)` returns the list of sequent formulas in `seq` corresponding to the given `fnums`. The Lisp expression `(delete-seq seq fnums)` returns the list of sequent formulas in `seq` that are not selected by the given `fnum`. If we are interested in selecting the sequent formulas according to some predicate, then the Lisp expression `(gather-seq seq yes-fnums no-fnums pred)` returns the list of sequent formulas in `seq` that are selected by `yes-fnums` but not by `no-fnums` such that the formula part of the sequent formula satisfies the unary predicate given by `pred`. Given a sequent formula `sf` in `(s-forms *goal*)` or in the list returned by `gather-seq`, the Lisp expression `(formula sf)` returns the actual PVS term corresponding to the sequent formula. Note that the formula numbers input to `gather-seq` can also `be` `'*` (for all the formulas), `'+` (for the consequent formulas), and `'-` (for the antecedent formulas), and also formula labels.

Since many commands take formula numbers or lists of formula numbers as arguments, it is useful to select these numbers rather than the formulas themselves. The Lisp expression `(gather-fnums seq yes-fnums no-fnums pred)` returns the list of all the formula numbers of sequent formulas in `seq` corresponding to `fnums` that satisfy the predicate `pred`. Note that any reference to the actual PVS term representing the sequent formula `sf` in the predicate `pred` will have to be of the form `(formula sf)`.

Thus, the Lisp expression

```
(gather-seq (s-forms *goal*)
            '-
            nil
            #'(lambda (sf) (and (negation? (formula sf))
                                 (forall-expr? (args1 (formula sf))))))
```

collects the list of universally quantified antecedent formulas, and the Lisp expression

```
(gather-fnums (s-forms *goal*)
              '-
              nil
            #'(lambda (sf) (and (negation? (formula sf))
                              (forall-expr? (args1 (formula sf))))))
```

returns the corresponding list of formula numbers.

5.4 Strategy Expressions

We describe the more easily understood aspects of strategies below. Any strategy expression can be typed in at the `Rule?` prompt in a proof. The syntax for strategy expressions is as follows:

$$
\begin{array}{rcl}
\langle \mathit{step} \rangle & := & \langle \mathit{primitive\text{-}rule} \rangle \\
& | & \langle \mathit{defined\text{-}rule} \rangle \\
& | & \langle \mathit{defined\text{-}strategy} \rangle \\
& | & (\texttt{quote}\ \langle \mathit{step} \rangle) \\
& | & (\texttt{try}\ \langle \mathit{step} \rangle\ \langle \mathit{step} \rangle\ \langle \mathit{step} \rangle) \\
& | & (\texttt{if}\ \langle \mathit{lisp\text{-}expression} \rangle\ \langle \mathit{step} \rangle\ \langle \mathit{step} \rangle) \\
& | & (\texttt{let}\ \ (\{(\langle \mathit{symbol} \rangle\ \langle \mathit{lisp\text{-}expression} \rangle)\}^{+})\ \langle \mathit{step} \rangle)
\end{array}
$$

5.5 Defining Strategies

User-defined strategies should be saved in a file called `pvs-strategies`. PVS loads strategies from files of this name from both the user's home directory and the current context directory. A strategy definition has the form:

```
(defstep name
         (required-parameters
           &optional optional-parameters
           &rest parameter)
    strategy-expression
    documentation-string
    format-string)
```

This generates both a (blackbox) defined rule *name* and a (glassbox) strategy *name£*. There are two other definition forms that are essentially similar to `defstep`. These are `defstrat` and `defhelper`. The `defhelper` version is identical to `defstep` but is used for defining strategies that are only meant to be used in the definition of other strategies and are not likely to be invoked directly by the user. The `defstrat` version defines only a glassbox strategy called *name* and the rule or blackbox version is not defined. The `defstrat` form does not take the final format-string argument given to `defstep`. The differences between a defined rule and a strategy are:

1. A defined rule like a primitive rule is atomic, whereas a strategy could expand to the application of several atomic rules.

2. Only the expanded form of a strategy is saved to be rerun, whereas a rule is saved and rerun in its unexpanded form.

3. A defined rule merely returns the unproved subgoals, whereas a strategy returns the expanded proof tree. For example, `prop$` and `ground` are strategies and `prop` and `ground` are their corresponding rule versions. The former are glass boxes in that their internal behavior is visible to the user, whereas the latter are black boxes.

Otherwise, defined rules and strategies are very similar. Both can be recursive and can involve the application of a number of primitive proof steps to achieve their effect.

In the following, we describe some important strategies used for defining new proof rules.

5.6 The Basic Strategies

`quote`	Identity strategy
`try`	Subgoaling and Backtracking
`if`	Conditional strategy
`let`	Evaluate/bind Lisp expressions/values

5.6.1 `quote`: The Identity Strategy

syntax: `(quote` *step*`)`

effect: The strategy expression `(quote` *step*`)` simply evaluates to *step* itself. The reason it is needed is that it is quite typical to construct a Lisp expression corresponding to a strategy but this then needs to be unquoted to be used as a strategy. For example the strategy `(then* &REST steps)` is given the recursive definition

```
(let ((x (when steps (car steps)))
      (y (when steps (cons 'then* (cdr steps)))))
  (if steps (if (cdr steps)(then x y) x) (skip)))
```

In evaluating the body of this definition, the quoted forms of the values of the bindings for `x`, `y` and `z` are substituted for these variables into the body of the let-expression. When any of these values are to be evaluated as strategies, they are unquoted by the definition for the `quote` strategy given above.

5.6.2 `try`: Strategy for Subgoaling and Backtracking

syntax: `(try` *`step1 step2 step3`*`)`

effect: This is the basic control strategy. It applies *step1* to the current goal. If *step1* succeeds and generates subgoals, then *step2* is applied to those subgoals. If *step1* does nothing, *i.e.*, behaves as a `(skip)` step, then *step3* is applied to the current goal. The `try` step thus provides a backtracking mechanism for proof search that can be controlled by appropriately signalling failure. The following identities describe the behavior of the `try` strategy:

1. `(try (skip) (assert) (split)) = (split)`
2. `(try (try (skip) (assert) (fail)) (split) (flatten)) = (flatten)`
3. `(try (try (lemma "assoc") (fail) (assert)) (split) (flatten)) = (flatten)`
4. `(try (try (lemma "assoc") (assert) (fail)) (split) (flatten)) = (flatten)`

The important thing to note is that if step A succeeds on the current goal, then applying `(try` A B C`)` causes the alternative C to be closed. Then `(fail)` backtracks to the last open (*i.e.*, not closed) alternative.

usage: `(try (flatten) (propax) (split))` : Applies the disjunctive simplification step to the current goal. If the goal does disjunctively simplify, then the `(propax)` step is applied to the resulting subgoal. Otherwise the conjunctive splitting step is applied to the current goal.

`(try (try (flatten) (fail) (skolem 1 ("a" "b"))) (postpone) (prop))` : If the current goal disjunctively simplifies, then backtrack and apply `(prop)`, otherwise introduce skolem constants and if that fails, try propositional simplification, otherwise postpone. Notice how `(fail)` is used to trigger backtracking from a subgoal.

5.6.3 `if`: Conditional Selection of Strategies

syntax: `(if` *condition step1 step2*`)`

effect: Here *condition* is some Lisp code that is evaluated against the current goal. If *condition* evaluates to `nil`, then *step2* is applied, else *step1* is applied.

usage: `(if (equal (get-goalnum *ps*) 1) (ground) (prop))` : If the current goal (`*ps*` is the current proofstate) is the first subgoal of its parent, the apply `(ground)`, else apply `(prop)`.

5.6.4 `let`: Using Lisp in Strategies

syntax: `(let ((`var_1 $lexpr_1$`)` ... `(`var_n $lexpr_n$`))` *step*`)`

effect: Here var_1 through var_n are symbols, and $lexpr_1$ through $lexpr_n$ are Lisp expressions. The `let`-form allows some values to be computed to be plugged into strategies. The scope of each let-binding extends over the later bindings and the body of the `let` strategy, so that it is similar to the `let*` construct of Lisp.

usage: A simplified definition of the basic querying strategy is shown below. Here, a defined Lisp function `qread` is used to generate the `"Rule?"` query and read the resulting user input.

```
(query*) = (let ((input (qread "Rule? ")))
             (try input (query*) (query*)))
```

The `then*` strategy has the definition shown below where it is defined by iterating the `then` strategy.

```
(then* &rest steps)
   = (let ((x (when steps (car steps)))
     (y (when steps (cons 'then (cdr steps))))
     (z steps))
(if steps (if (cdr steps) (try x y y) x) (skip)))
```

5.7 Strategies

We have already briefly discussed the differences between strategies and defined rules. In many of the cases, defined rules are analogous to the *tactics* of LCF, and strategies are like the *tacticals* which are used to combine rules in various ways.

5.7.1 `branch`: Assigning Strategies to Subgoals

syntax: `(branch` *`step steplist`*`)`

effect: Just like spread except that when *steplist* has only n elements and *step* generates more than n subgoals, the n'th element of `steplist` is also applied to the subgoals following the n'th one. This is useful when only the first few subgoals generated by *step* require special attention and the rest of the subgoals yield to some uniform strategy.

5.7.2 `else`: A Simple Backtracking Strategy

syntax: `(else` *`step1 step2`*`)`

effect: First applies *step1* and if that does nothing, then *step2* is applied to the present goal. The definition of `else` is just `(try` *`step1`* `(skip)` *`step2`*`)`.

5.7.3 `query*`: The Basic Interaction Strategy

syntax: `(query*)`

effect: This is the strategy that repeatedly queries the user for the current rule or strategy. PVS also invokes `query*` when all other options have been exhausted and the prover is being used in an interactive mode.

5.7.4 `checkpoint`: Checkpoint Handling

syntax: `(checkpoint)`

effect: A synonym for `query*`: inserting `(checkpoint)` into an edited proof and rerunning it causes the non-checkpointed subproofs to be installed (using `just-install-proof`) so that the proof is only run up to the checkpoint.

notes: This command is not meant to be used directly; see the User Guide for using it in editing proofs.

5.7.5 `just-install-proof`: Checkpoint Handling

syntax: `(just-install-proof` *`proof)`*

effect: Installs, the *proof* without actually checking it, and treats the current subgoal as proved, but marks the proof as unfinished. Used in conjunction with `checkpoint`.

notes: This command is not meant to be used directly; see the User Guide for how it is used in editing proofs.

5.7.6 `repeat`: Iterate Along Main Proof Branch

syntax: `(repeat` *`step`*`)`

effect: First applies *step* to the current goal. If this does nothing, then no further steps are indicated. If the application of *step* generates subgoals, then `(repeat` *`step`*`)` is recursively applied to the first of those subgoals (the main subgoal). See *repeat** below. The *repeat* strategy must be used cautiously. It can easily cause loops since it is only terminated when *step* does nothing. Some commands such as `assert` almost always take effect even when they seemingly do nothing, and wrapping these within a `repeat` can cause a loop.

5.7.7 `repeat*`: Iterate Along all Branches

syntax: `(repeat*` *`step`*`)`

effect: While *repeat* only repeats the step for the main branch of the proof, `repeat*` carries out the repetition along all the subgoals resulting from the first application of *step*. The repetition is continued along each branch until an application of *step* has no effect.

5.7.8 `rerun`: Rerun a Proof or Partial Proof

syntax: `(rerun` &OPTIONAL *`proof`*`)`

effect: This step can be used to rerun a partial or completed proof from a previous attempt or from another branch of the proof. This step is largely used automatically by the system when it queries as to whether the proof should be rerun. The *proof* argument can also be explicitly given by the user using either the `M-x edit-proof` or `M-x show-proof` commands to generate and edit such inputs. This step can be used to:

1. Restore a partial proof to the state when the proof was interrupted.
2. Recheck a completed proof.
3. Redo a proof following some changes to the specification. It is possible that the old proof only partially works for the changed specification. In this case, it is usually possible to clean up and complete the resulting partial proof.
4. Apply a partial or completed proof from one subgoal to some other subgoal in the proof attempt.

5.7.9 `spread`: Assigning Strategies to Subgoals

syntax: `(spread` *`step steplist`*`)`

effect: First applies *step1* and then applies the *i*'th element of *steplist* to the *i*'th subgoal. This is typically used when *step* splits the proof into multiple branches where a different strategy is required for each of the branches. See *branch* below.

5.7.10 `spread!`: Assigning Strategies to Subgoals

syntax: `(spread!` *`step steplist`*`)`

effect: Like `spread`, applies *step* and then pairs the steps in *steplist* with the subgoals, but generates an error and queries the user if the number of subgoals do not match the number of subproofs.

5.7.11 `spread@`: Assigning Strategies to Subgoals

syntax: `(spread@` *`step steplist`*`)`

effect: Like `spread`, applies *step* and then pairs the steps in *steplist* with the subgoals, but generates a warning if the number of subgoals do not match the number of subproofs.

5.7.12 `then`: A Sequencing Strategy

syntax: `(then` &REST *`steps`*`)`

effect: Let the list *steps* consists of the first element *step1* and the remaining steps *rest-steps*. This strategy first applies the *step1* to the current goal. If any subgoals are generated, then `(then :steps` *`rest-steps`*`)` is applied to each of these subgoals. If *step1* has no effect, then `(then :steps` *`rest-steps`*`)` is applied to the original goal. The definition of `then` is just `(try` *`step1 step2 step2`*`)`.

5.7.13 `then*`: Apply Steps in Sequence

syntax: `(then*` &REST *`steps`*`)`

effect: This command is subsumed by `then` and should now be considered obsolete. This is an iterated version of `then` so that `(then*` *`step1 ...stepn`*`)` is just `(then step1 (then ...`*`stepn`*`))`.

5.7.14 `then@`: Apply Steps in Sequence Along Main Branch

syntax: `(then@` &REST *`steps`*`)`

effect: This is a version of `then*` where the given steps are applied in sequence only along the main branch of the proof, *i.e.*, if the the given rule is `(then@` *`step`*$_1$ …*`step`*$_n$`)`, then $step_{i+1}$ is only applied to the first subgoal of $step_i$.

5.7.15 `time`: Time a Given Strategy

syntax: `(time` *`strategy`*`)`

effect: Executes the given rule or strategy as an atomic step (like the `apply` command) while printing out the run times at each of the leaf nodes. This command has no other effect on the proof. It only prints out timing information when there are leaf nodes generated and yields no information when the given strategy succeeds in proving the subgoal.

usage: `(time (then (lift-if)(prop)(skolem!)))`

errors: No error messages other than those generated by the given strategy.

5.7.16 `try-branch`: Branch or Backtrack

syntax: `(try-branch` *`step1 steplist step2`*`)`

effect: This is a combination of the `try` and `branch` strategies. It is like branch for *step1* and *steplist* except that when *step1* has no effect, then *step2* is attempted on the original subgoal.

Bibliography

[1] J. C. Bicarregui and B. Ritchie. Reasoning about VDM using the VDM support tool in Mural. In S. Prehn and W. J. Toetenel, editors, *VDM '91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*, pages 371–388, Noordwijkerhout, The Netherlands, October 1991. Springer-Verlag. Volume 1: Conference Contributions. 2

[2] W.W. Bledsoe and P. Bruell. A man-machine theorem-proving system. In *Advance Papers of Third International Joint Conference on Artificial Intelligence*. W.W. Bledsoe, 1974. 2

[3] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979. 2

[4] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, New York, NY, 1988. 2

[5] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, Englewood Cliffs, NJ, 1986. 2

[6] N. G. de Bruijn. A survey of the project Automath. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 589–606. Academic Press, 1980. 2

[7] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979. 2

[8] M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, Boston, MA, 1988. 2

[9] W. McCune. OTTER 2.0 users guide. Technical Report ANL-90/9, Argonne National Laboratory, 1990. 2

[10] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999. 1, 7

[11] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999. 1, 5, 7

Index