The PVS Proof Checker: A Reference Manual (Beta Release)

N. Shankar, S. Owre and J. M. Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025
Phone: (415) 859-5272
{Shankar,Owre,Rushby}@csl.sri.com

March 31, 1993

Contents

1 Introduction

PVS stands for "Prototype Verification System," and as the name suggests, it is a prototype for a system for specification and verification based on higher-order logic.[1] The present document is a reference manual for the commands employed in constructing proofs using the PVS proof checker. The *PVS User Guide* [OSR93b] should be consulted for information on how to use the system to develop specifications and proofs. The *PVS Language Reference* [OSR93a] describes the language of PVS. The *PVS Tutorial* [SOR93] provides an introduction to the entire system.

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.

One notable aspect of PVS is that it combines a rich expressive specification language and an effective theorem prover. Most effectively mechanized theorem provers have generally been associated with the most restricted specification notations, and vice-versa. For example, the very powerful Boyer-Moore theorem prover [BM88] uses a constructive, untyped, quantifier-free first order logic as its specification lan-

[1]The development of PVS was funded by SRI International. Significant inputs were provided by Friedrich von Henke, David Cyrluk, Carl Witty, Steven Phillips, Judy Crow, Patrick Lincoln, and Mandayam Srivas, among many others. Decision procedures for equality and linear arithmetic that were originally due to Rob Shostak, form a significant component of the PVS inference mechanisms.

guage. Conversely, more attractive and expressive specification notations have generally lacked effective mechanical support—for example, "proof assistants" for Z and VDM (such as *Mural* [BR91]) provide attractive mouse-and-windows interfaces for natural deduction but lack deductive machinery. To support the combination of an expressive logic and a powerful theorem proving capability, PVS features careful integration between the typechecker and the proof checker. The typechecker makes use of the deductive power of the proof checker to automatically discharge proof obligations that come up as type correctness conditions corresponding to predicate subtypes during typechecking. 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's combination of direct control by the user for the higher levels of proof development, and powerful automation for the lower, is also somewhat unusual. On the whole, PVS provides more automation than a low-level proof checker (such as LCF [GMW79], HOL [Gor88], Nuprl [C$^+$86], Automath [dB80]), and more control than a highly automatic theorem prover (such as Otter [McC90] or Nqthm [BM79, BM88]). Compared with thoroughly automated theorem provers such as that of Boyer and Moore, 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 colleages [BB74].

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 does yield a rather productive verification environment. There are a number of examples, both big and small, that support this observation. PVS has been used to verify small examples such as the correctness of ordered binary tree insertion, the Boyer-Moore majority algorithm, a real-time mutual exclusion algorithm, and the Oral Messages algorithm for Byzantine agreement. These verifications have taken from a couple of hours to a day or so to undertake in PVS, from writing the specification to completing the proofs. Larger verifications include the correctness of pipelined processors, distributed agreement under a hybrid fault model, and a real-time railroad crossing controller. These examples typically take less than a week of total human effort to complete. This verification process can be further sped up by systematizing the specification format and customizing the proof checker with specialized proof strategies.

1.1 PVS Proof Display and Construction

The prover is interactive, although it supports a batch mode in which proofs can easily be 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*. 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

```
{-1}   A1
{-2}   A2
[-3]   A3
        ⋮
|-------
{1}    B1
[2]    B2
{3}    B3
        ⋮
```

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.[2] 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 should imply 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. 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 are all 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. Numbers in braces, as opposed to brackets (*e.g.*, {2} in the example above), serve to highlight those formulas that are changed 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 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

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

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 all of 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 those 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, a significant attribute of the PVS prover is that its proof steps are extremely sophisticated and employ arithmetic and equality decision procedures. Aspects of the PVS language, such as 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, several of these commands are quite powerful and can be used in many different ways. It is advisable to experiment with all of the commands in order to more thoroughly understand how they work. As with any automated reasoning system, the form of the specification can significantly impact the ease or difficulty of the proof. The specifier must demonstrate good taste in writing abstract specifications, using definitions to name useful concepts, employing types, subtypes, and abstract datatypes appropriately, and 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 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.

Section 2 contains a small example proof to illustrate how the PVS proof checker is used. Section 3 gives a brief overview of the logical underpinnings of PVS. Section 4 describes the syntax of the proof commands. Section 5 is a guide to the primitive PVS proof rules. Section 6 is a guide to the PVS proof strategy language and also gives examples of proof strategies and derived inferences rules.

2 An Example Proof

We consider the example of the associativity of conjunction. This is stated simply in the theory below.

```
assoc : THEORY
  BEGIN

  A, B, C: VAR bool

  assoc: LEMMA ((A AND B) AND C) IMPLIES (A AND (B AND C))

  END assoc
```

The proof below uses three PVS proof commands:

(skolem! *fnum***)**: Introduces new Skolem constants[3] for the bound variables in the formula *fnum*, if this formula is either an antecedent formula of the form $(\exists x_1, \ldots, x_m \colon A)$ or a consequent formula of the form $(\forall x_1, \ldots, x_m \colon A)$. If the generated *names* are $a_1, \ldots, a_m$, then the formula numbered *fnum* is replaced by the formula $A[a_1/x_1, \ldots, a_m/x_m]$ which results from replacing each free occurrence of variable x_i in A by the corresponding a_i. The *fnum* argument to `skolem!` is optional; the prover searches for the first suitable sequent formula if this argument is omitted.

(flatten): Disjunctively simplifies all the sequent formulas so that the resulting sequent contains no antecedent formulas of the form $A \wedge B$ or $\neg A$, and no consequent formulas of the form $A \vee B$, $A \supset B$, or $\neg A$.

(split *fnum***)**: Causes a conjunctive splitting corresponding to a chosen antecedent formula of the form $A \vee B$ or $A \supset B$, or a consequent formula of the form $A \wedge B$. As with `skolem!`, the *fnum* argument can be omitted.

A proof session is initiated by 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 queried as to whether the proof attempt should proceed. If a proof or partial proof for the relevant formula already exists, then the user is further queried as to whether this proof should be rerun.

In the script 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

[3]The prover automatically generates names of the form `x!`n, where `x` is the name of the quantified variable, and n is a number guaranteed to make the new constant unique. The names may be directly specified by using the `skolem` command described later.

attempt can be abandoned by typing `q` or `quit` at the prompt. The user is queried as to whether the resulting partial proof should be saved. After the interactive proof is completed and the user is queried as to whether the new proof should be saved. (This query only comes up when an old proof exists.) The system then presents the option of having a brief version of the proof printed. The timing and storage characteristics are displayed at the end.

```
assoc :

  |-------
{1}   (FORALL (A, B, C: bool): ((A AND B) AND C) IMPLIES (A AND (B AND C)))

Rule? (skolem!)
For the top quantifier in 1, we introduce Skolem constants: (a!1 b!1 c!1)
this simplifies to:
assoc :

  |-------
{1}   ((a!1 AND b!1) AND c!1) IMPLIES (a!1 AND (b!1 AND c!1))

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
assoc :

{-1}   a!1
{-2}   b!1
{-3}   c!1
  |-------
{1}   (a!1 AND (b!1 AND c!1))

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

[-1]   a!1
[-2]   b!1
[-3]   c!1
  |-------
{1}   a!1

which is trivially true.

This completes the proof of assoc.1.

assoc.2 :

[-1]   a!1
[-2]   b!1
[-3]   c!1
  |-------
{1}   b!1

which is trivially true.

This completes the proof of assoc.2.

assoc.3 :
```

```
[-1]   a!1
[-2]   b!1
[-3]   c!1
  |-------
{1}   c!1

which is trivially true.

This completes the proof of assoc.3.

Q.E.D.


Run time  = 0.60 secs.
Real time = 13.51 secs.
```

The reader should be clear about the above model of interaction in using the system. The PVS Prover Tutorial provides more examples of PVS proofs. 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.

3 The Logic of PVS

While using the PVS proof checker, it is useful to be aware of the rules underlying the PVS logic. Though the rules presented in this section are not the ones that are directly implemented in PVS, they do form the theoretical basis for the implementation. There is, of course, a great deal more to building an effective proof checker than merely codifying the 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 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`, `ALL`
$\exists$	`EXISTS`, `SOME`
λ	`LAMBDA`, `LAM`

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

PVS employs a sequent calculus. This means that a *judgement* is expressed as a *sequent*. A sequent is of the form $\Gamma \vdash \Delta$, where Γ is a (possibly empty) *list* of formulas $\{A_1, \ldots, A_m\}$, and Δ is a (possibly empty) *list* of *formulas* $\{B_1, \ldots, B_n\}$. A formula is just a term with type `bool`. In a sequent, $\Gamma \vdash \Delta$, the formulas in Γ are called the *antecedent* formulas, and the formulas in Δ are called the *consequent* formulas. Together, the formulas in Γ and Δ are called the *sequent formulas*. The intuitive reading of such a sequent is $A_1 \wedge \ldots \wedge A_m \supset B_1 \vee \ldots \vee B_n$, *i.e.*, the *conjunction* of the formulas in Γ implies the *disjunction* of the formulas in Δ.

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.

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

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

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

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 $(\forall x, y{:}\,(\lambda z{:}\, f(y,z))(x) < y) \equiv (\forall z, x{:}\,(\lambda y{:}\, f(x,y))(z) < x)$ 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 *free* occurrences of e in the formula A, *i.e.*, where there are no free variable occurrences in e that appear bound in $A[e]$. The notation $\Delta[e]$ similarly highlights occurrences of e in Δ where no free occurrences of variables of e appear bound in $\Delta[e]$.

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

3.5 The Quantifier Rules

The quantifier rules are stated below. In the $\vdash \forall$ and $\exists \vdash$ rules, a must be a new constant.

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

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

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 $[\texttt{bool}, \alpha, \alpha \rightarrow \alpha]$. The transformation of $A[e]$ to $A[b]$ represents the replacement of the highlighted occurrences of e in A by b. 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.

4 The Proof Commands

The sequent calculus inference rules displayed in Section 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 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. There are three kinds of proof commands: primitive rules, defined rules, and strategies. The primitive rules correspond to the core logic and they cannot be changed. The primitive rules include a language for defining new proof strategies. Both the defined rules and strategies expand to primitive proof steps. The main difference between rules and strategies is that rules are atomic, *i.e.*, a rule when applied to a goal reduces it to zero or more subgoals in a single step, whereas a strategy generates a tree of rule-applications while reducing a goal to subgoals.

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:

$$(\langle \mathit{rulename} \rangle\ \langle \mathit{required} \rangle^*\ \&\text{OPTIONAL}\ \langle \mathit{optional[default]} \rangle^*\ \&\text{REST}\ \langle \mathit{argument} \rangle)$$

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.[4] 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.

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

[4] 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.

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 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*.

It is important to realize that 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 should respond with the prompt, and then type the corrected command. 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 a `:a` in the Lucid version, and a `:reset` in the Allegro version of PVS.) 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`*]* `)`
- `(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)`

The description of each proof rule below contains its syntax, its effect, examples, error messages, and notes. It should be noted that each rule is invoked at some leaf node of a partial proof tree. We call this leaf node the *focus* of the partial proof. A rule invocation can either have no effect, change the focus of the proof, add subgoal nodes to the current focus and shift the focus to the first such subgoal, complete the current branch of the proof, propagate a failure to the parent goal, or undo the proof to some ancestor node.

If the user invokes a rule or strategy that is not known to the system, the error message:

```
Ill-formed rule/strategy: ...
```

is printed and there is no change to the state of the proof.

5 The Primitive Proof Commands

The primitive proof commands correspond to the core logic of PVS and they cannot be changed. The primitive rules include a language for defining proof strategies. Both the defined rules and proof-steps expand to primitive proof commands. The main difference between defined rules and proof-steps is that rules are atomic, *i.e.*, a rule when applied to a goal reduces it to zero or more subgoals in a single step, whereas a proof-step generates a tree of rule-applications while reducing a goal to subgoals.

The primitive rules essentially define the logic underlying PVS. There is considerable overlap between the functionalities of these proof rules. The primitive rules are a fixed set of these rules that cannot be changed, and every PVS proof follows from some combination of these primitive rules. Each primitive proof rule 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

The primitive proof rules can be classified as:

1. Basic rules
2. Structural rules
3. Propositional rules
4. Quantifier rules
5. Equality rules
6. Other rules

5.1 The Basic Rules

`help`	provides brief documentation
`skip`	has no effect but is useful in defining larger strategies.
`postpone`	causes the current goal to be left pending
`fail`	has the effect of signalling a failure
`undo`	undoes proof steps along a branch of the proof

`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 `M-x help-prover` command, which puts the display in a buffer that can be referenced as the proof is developed.

`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.

`postpone`: Go to Next Remaining Goal

syntax: `(postpone)`

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.

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.

`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 48 for examples of the use of `fail`.

`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.

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

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

`(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.

5.2 The Structural Rules

`delete`	deletes selected formulas from a goal sequent
`hide`	temporarily hides selected formulas from the displayed goal
`reveal`	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 Section 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 the formula numbers. The Contraction rule only appears in a limited form in PVS; 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 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` displays hidden formulas along with their numbers.

`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.

`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 `quant` rule and the `inst` rule automatically hide the quantified formula so that quantifiers can be reinstantiated further down in 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.

`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.

5.3 The Propositional Rules

`propax`	the propositional axiom rule
`flatten`	the disjunctive simplification rule
`split`	the conjunctive splitting rule
`lift-if`	the `IF`-lifting rule
`case`	introduces case splits (the Cut rule)
`iff`	converts boolean equalities to *if and only if* form

`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 Section 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.

`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 Section 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, although it may be difficult later to tell where a particular subformula came from.

`split`: Conjunctive Splitting

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

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)`.

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.

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 $\vdash \wedge$, $\vee \vdash$, $\supset\vdash$, $\texttt{IF} \vdash$, and $\vdash \texttt{IF}$ inference rules given in Section 3. These are all the inference rules which cause branching.
- To get the effect of repeatedly applying `flatten` and `split`, use the `prop` command.

`lift-if`: Lifting Embedded IF Connectives

syntax: `(lift-if` &REST *fnums[*]* `)`

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` expression. These `IF` and `CASES` branches could occur embedded within the formula. In order to carry out the corresponding case-split in the proof, 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 $f(\mathtt{IF}(A, B, \mathtt{IF}(C, D, E)))$ to $\mathtt{IF}(A, f(B), \mathtt{IF}(C, f(D), f(E)))$. On the other hand, $f(\mathtt{IF}(\mathtt{IF}(A, B, C), D, E))$ is transformed by `lift-if` to $\mathtt{IF}(A, f(\mathtt{IF}(B, D, E)), f(\mathtt{IF}(C, D, E)))$, reflecting the selection of the conditionals of the leftmost-innermost `IF`-expression. Note that $f(\mathtt{IF}(A, \mathtt{IF}(B, C, D), \mathtt{IF}(E, F, G)))$ is transformed to $\mathtt{IF}(A, \mathtt{IF}(B, f(C), f(D)), \mathtt{IF}(E, f(F), f(G)))$ 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.

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.

`(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)` : Same as above.

errors: No error messages are generated.

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

`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 $A_1, \ldots, A_n, \Gamma \vdash \Delta$, $A_1, \ldots, A_{n-1}, \Gamma \vdash A_n, \Delta$, $\ldots$, $A_1, \Gamma \vdash A_2, \Delta$, and $\Gamma \vdash A_1, \Delta$. 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`.

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:

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

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 `case` rule is disallowed from introducing such 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.

`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 $A \iff B$.[5] Treating all boolean equalities as equivalences is not a good idea since that leads to a combinatorial explosion when the propositional steps are applied. In many cases, the boolean equality 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.

5.4 The Quantifier Rules

`skolem`	introduces *Skolem* constants for quantifiers of universal strength
`quant`	instantiates (essentially) existentially quantified variables

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

`skolem`: Replacing Universal Quantification with Constants

syntax: (`skolem` *fnum constants*)

effect: If the formula in the current sequent indicated by *fnum* is either an antecedent formula of the form $(\exists x_1, \ldots, x_n: A)$ or a consequent formula of the form $(\forall x_1, \ldots, x_n: A)$, 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 $A[c_1/x_1, \ldots, c_n/x_n]$.[6] 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.

usage: `(skolem * ("a3" "b3" "c3"))` : The first suitable formula in the sequent with the form $(\forall/\exists x_1, x_2, x_3: A)$ 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 $(\exists x_1, x_2, x_3: A)$ with $(\exists x_2: A)[\texttt{a1}/x_1, \texttt{a3}/x_3]$.

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.

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*.

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

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!` strategy described later 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.

`quant`: Instantiating Existential Quantifiers

syntax: `(quant` *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 `quant` 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: `(quant * ("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 $A[\mathtt{x} + \mathtt{3}/x_1, \mathtt{y} - \mathtt{z}/x_2]$.

`(quant - ("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 `quant`.

`(quant -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 $(\forall x_1, x_2 \colon A)$.

`(quant -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*.

Formula ... is not skolemizable: The formula explicitly indicated by *fnum*.

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 `quant` 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 `quant` rule supplies a substitution for y, it must also include a substitution for x.

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

notes: The defined rule `inst` is a non-copying form of `quant`, and the rule `inst-cp` is a copying form of `quant`. In the related `quant?` 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 `quant?` does not always succeed with the right match. It typically succeeds when there are very few possible matches. The `quant` rule captures the $\forall \vdash$ and $\vdash \exists$ sequent rules.

5.5 The Equality Rules

`lemma`	introduces instances of axioms, lemmas, and definitions
`expand`	expands (and simplifies) function definitions
`replace`	replaces LHS for RHS (or vice-versa) using equality formula
`beta`	beta-reduces λ-, record/tuple access, and datatype exprs
`extensionality`	introduces extensionality axioms
`name`	introduces a name for a complex expression

`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 `boolprops` 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 "boolprops.assoc")` : Adds the formula

$$(\forall (x, y, z\colon \text{BOOL})\colon ((x \wedge y) \wedge z) = (x \wedge (y \wedge z)))$$

to the antecedent of the current sequent.

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

$$(\forall (y\colon \text{BOOL})\colon ((\text{TRUE} \wedge y) \wedge \text{FALSE}) = (\text{TRUE} \wedge (y \wedge \text{FALSE})))$$

to the antecedent of the current sequent.

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

$$((\text{TRUE} \wedge A) \wedge \text{FALSE}) = (\text{TRUE} \wedge (A \wedge \text{FALSE}))$$

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` rule 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.

`expand:` Expanding Definitions

syntax: `(expand` *name* &OPTIONAL *fnum[*]* *occurrence* *if-simplifies* *beta-reduce[*`t`*]*)

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 n^{th} occurrence of the function symbol counting from the left. 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. If *beta-reduce* is `t`, then the resulting expression is beta-reduce.

usage: `(expand "sum")` : Expands the definition of `sum` throughout the current sequent, whether they simplify or not. The resulting expressions are all beta-reduced.

`(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" :beta-reduce nil)` : Expands `sum`, but does not beta-reduce the result.

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* to be specified.
- `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.

`replace:` Rewriting Using Equalities

syntax: (`replace` *fnum* &OPTIONAL *fnums[*]* *direction[*`LR`*]*)

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`.

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**.

`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:

- $(\lambda x_1 \ldots x_n\colon e)(t_1, \ldots, t_n)$ reduces to $e[t_1/x_1, \ldots, t_n/x_n]$.
- $\texttt{proj}_i(t_1, \ldots, t_n)$ reduces to t_i.
- $label_i((\#label_1\colon=t_1, \ldots, label_n\colon=t_n\#))$ 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 got 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.

`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])")`[7] : 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.

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.

`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.

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

errors: **...is not a symbol:** The given *name* is not a symbol or a string that can be turned into a symbol.

...is already declared: The given *name* is already declared and therefore not new.

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

5.6 The Other Rules

`typepred`	introduces type constraints on expressions
`apply`	applies a proof strategy in a single atomic step
`record`	records assumptions for the decision procedures
`simplify`	does Boolean simplification using decision procedures
`assert`	uses decision procedures to assert sequent formulas
`auto-rewrite`	declares a list of formulas as rewrite rules
`auto-rewrite-theory`	declares a theory to be used as rewrite rules
`do-rewrite`	applies auto-rewrites
`stop-rewrite`	turns off auto-rewrite rules for the current subtree
`stop-rewrite-theory`	turns off auto-rewrites from a theory

`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.

`apply:` Making Proof Strategies Atomic

syntax: `(apply` *strategy* &OPTIONAL *comment*`)`

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.

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.

errors: No error messages are generated.

`record:` Recording Assumptions for the Decision Procedures

syntax: `(record` &OPTIONAL *fnums[*]* `)`

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.

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.

`simplify:` Simplifying Using Decision Procedures

syntax: `(simplify` &OPTIONAL *fnums[*]* `)`

effect: A sequent will frequently contain an `IF`-branching structure that can be simplified using a small amount of arithmetic and equality reasoning. The `simplify` command is used to carry out such simplification of `IF`, `CASES`, and propositional structure. It also simplifies arithmetic expressions, and eliminates record, tuple, `LAMBDA`, and datatype redexes.

usage: `(simplify)` : Tries to simplify all the formulas in the sequent using the PVS decision procedures.

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

notes: As with `record`, the `assert` command subsumes `simplify`. No new information is recorded into the data structures used by the decision procedures except the subtype constraints on subexpressions of formulas processed by the `simplify` command.

`assert:` Simplifying Using the Decision Procedures

syntax: `(assert` &OPTIONAL *fnums[*] rewrite-flag flush? linear?*`)`

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 merely 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. 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.

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*.

`auto-rewrite`: Adding Automatic Rewrite Rules

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

effect: The definitions and lemmas named 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. It is important to note that the names of lemmas should be completely specified, *i.e.*, all the actual theory parameters must be explicitly given, otherwise an error is signalled and that particular name is ignored by the rule. This is not the case for definitions. For curried definitions, only the most curried form of the left-hand side of the definition is rewritten so that if f has the definition $\lambda x\colon \lambda y\colon \lambda z\colon\ e$, then only expressions of the form $f(a)(b)(c)$ are automatically rewritten, and those of the form f, $f(a)$, or $f(a)(b)$ are not.

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 is an atomic Boolean proposition, where l is a *rewritable left-hand side* 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.

A rewritable left-hand side is either a constant c or an application of the form $f(a_1, \ldots, a_n)$, where f is a rewritable left hand side. The constant c is the `key` for the rewrite rule in the sense that it is the appearance of this constant in the leftmost position in an expression of the form $(\ldots c(a_1)\ldots(a_n))$ that triggers the application of the given rewrite rule.

It is important to note that `auto-rewrite` has no visible effect, 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.

The name ... must be fully instantiated with actual parameters: As indicated above, only fully instantiated rewrite rules can be used for automatic rewriting. This restriction might be weakened in future releases.

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.

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.

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

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

effect: Takes all the definitions and formulas declared in each of the theories listed in *names*, filters out those that meet the syntactic criteria for rewrite rules, and declares these to be automatic rewrite rules. Note that one cannot have a generic theory listed in *names*, *i.e.*, all the actuals for a theory must be supplied, unless it is the same theory as the proof in progress. When `auto-rewrite-theory` is applied to the current theory, all of the rewrite rules that precede the formula being proved in the theory are declared to be automatic.

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.

`do-rewrite`: Applying Auto-Rewrites

syntax: `(do-rewrite` &OPTIONAL *fnums[*]* `)`

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 `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.

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`.

`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.

`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`.

6 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.

6.1 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:

$$
\begin{array}{lcl}
\langle \mathit{step} \rangle & := & \langle \mathit{primitive\text{-}rule} \rangle \\
 & | & \langle \mathit{defined\text{-}rule} \rangle \\
 & | & \langle \mathit{defined\text{-}strategy} \rangle \\
 & | & (\texttt{quote}\ \langle \mathit{step} \rangle) \\
 & | & (\texttt{try}\ \langle \mathit{step} \rangle\ \langle \mathit{step} \rangle\ \langle \mathit{step} \rangle) \\
 & | & (\texttt{if}\ \langle \mathit{lisp\text{-}expression} \rangle\ \langle \mathit{step} \rangle\ \langle \mathit{step} \rangle) \\
 & | & (\texttt{let}\ (\{(\langle \mathit{symbol} \rangle\ \langle \mathit{lisp\text{-}expression} \rangle)\}^{+})\ \langle \mathit{step} \rangle)
\end{array}
$$

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 defined rule *name* and a strategy *name$*. For the present version of this document, we omit the details of how strategies are defined, particularly the use of Lisp expression to access the syntax of proof goals and the status of the proof. In the following, we describe some important strategies and defined proof rules.

`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.

`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.

`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)`.

`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)))))
        (if steps
           (if (cdr steps) (then x y) x)
           (skip)))
```

6.2 The Defined Rules

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. In the examples below, `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.

`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: The `prop` rule is defined using the `prop$` strategy described below and should almost always be preferred to it.

`ground:` Decision Procedures and Propositional Simplification

syntax: `(ground)`

effect: Carries out propositional simplification as with `prop` above, and applies the arithmetic decision procedures to any remaining subgoals. The decision procedures are fairly quick, but the same warning as `prop` applies with even greater strength to `ground` since `ground` is considerably more time-consuming. The recursive definition for `(ground)` is

```
(try (flatten)(ground) (try (split)(ground)(assert)))
```

notes: As with `prop`, `ground` is almost always to be preferred over the `ground$` strategy described below. Note that as with `assert`, `ground` is never equivalent to a `skip`, so it should not be included within `repeat` loops in strategies without a termination check for the loop.

`rewrite-lemma:` Rewriting Using a Lemma

syntax: `(rewrite-lemma` *name 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 the name for a lemma or a definition that can be used as a rewrite rule. *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.

`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.

`inst:` Instantiation of Universal Strength Quantifiers Without Copying

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

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

`inst-cp:` Instantiation of Universal Strength Quantifiers With Copying

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

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

`quant?:` Instantiation of Universal Strength Quantifiers

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

effect: This rule extends the capabilities of the `quant` 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 rest of the sequent (or as constrained by the *where* argument). The first successful match is used to instantiate the quantified variables. If a partial substitution is given but no match is found, then if *if-match* is `nil`, then the rule goes ahead and instantiates using the given partial substitution. If *if-match* is `T`, then the instantiation only takes place if the matching process succeeds. 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.

`inst?`: Instantiation of Universal Strength Quantifiers

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

effect: This is simply a synonym for quant? that is provided to be uniform with the use of `inst` and `inst-cp`.

`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.

`replace-extensionality`: Replace Using Extensionality

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

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.

`apply-extensionality`: Apply Extensionality to an Equality

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

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.

`induct`: Automatically Employ Induction Scheme

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

effect: The formula indicated by *fnum* must be a universally quantified, consequent formula. The variable name *variable* must be quantified at the outermost level of this formula. As with the substitutions in `quant?`, it is sufficient to give that part of the variable name preceding the `$` mark, if there is one in the bound variable name. The bound variable must be of type `nat` or a PVS abstract datatype for the induction scheme to be selected automatically. The induction scheme corresponding to the type is then instantiated using 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.

`forward-chain`: Forward Chaining

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

effect: This rule is used to forward chain on the given lemma. If the given lemma has the form $A_1 \wedge \ldots \wedge A_n \supset C$, then this rule looks 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.

6.3 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.

`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.

`skolem!`: Automatic Introduction of Skolem Constants

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

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. This is the reason this rule is a strategy—so that it is the expanded `skolem` form of the rule that is actually saved and rerun. When the *fnum* argument is `*`, `+`, or `-`, the first Skolemizable formula in this range is selected. It should be noted that placing a `skolem!` step within a defined rule rather than a strategy is likely to make the proof less robust against even small changes.

`skosimp`: Skolemize then Flatten

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

effect: This is a short form for `(then (skolem` *fnum*`) (flatten))`. It is a strategy and not a defined rule for the same reason as `skolem!` above.

`skosimp*`: Repeatedly Skolemize then Flatten

syntax: `(skosimp*)`

effect: This is a short form for `(repeat (then (skolem) (flatten)))`.

`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*`)`.

`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.

`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.

`then:` A Sequencing Strategy

syntax: `(then` *step1 step2*`)`

effect: First applies *step1* and if any subgoals are generated, then *step2* is applied to each of these subgoals. If *step1* has no effect, then *step2* is applied to the original goal. The definition of `then` is just `(try` *step1 step2 step2*`)`.

`then*:` Apply Steps in Sequence

syntax: `(then*` &REST *steps*`)`

effect: This is an iterated version of `then` so that `(then*` *step1* ... *stepn*`)` is just `(then step1 (then` ... *stepn*`))`.

`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 $step_i$.

`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.

`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.

`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.

`prop$:` Propositional Simplification

syntax: `(prop$)`

effect: This is an observable, glass-box version of `prop`. It applies `(flatten)` and `(split)` in succession, and if any step succeeds and generates subgoals, then `(prop$)` is applied recursively to these subgoals. It's to be used when the goal sequent is to be propositionally simplified so that there are no axioms or top-level propositional connectives left among the resulting subgoal sequents. It should be used with care since a lot of subgoals can easily be generated.

`ground$:` Decision Procedures and Propositional Simplification

syntax: `(ground$)`

effect: This is similar to `prop$` and is a glass-box version of `ground`. It applies `(flatten)`, `(split)`, and `(assert)` in succession and recursively invokes `ground$` on any subgoal. The same precautions noted for `prop$` apply here with greater strength.

`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.

`tcc-strategy:` Automatically Prove Type Correctness Conditions

syntax: (tcc-strategy)

effect: This strategy automatically applies `skolem!`, `quant?`, `ground`, and `lift-if` in an attempt to automatically prove trivial type correctness conditions. This currently has the definition:

```
(repeat* (try (then* (repeat* (skolem!))
                     (repeat* (lift-if)))
              (apply (then (ground)(assert)))
              (fail)))
```

This definition can be modified by the user. Note how we have avoided using `assert` or `ground` in a way that could cause a nonterminating `repeat*` loop.

References

[BB74] 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.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.

[BM88] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, New York, NY, 1988.

[BR91] 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.

[C+86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[dB80] N. G. de Bruijn. A survey of the project AUTOMATH. In *Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 589–606. Academic Press, 1980.

[GMW79] 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.

[Gor88] 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.

[McC90] W. McCune. OTTER 2.0 users guide. Technical Report ANL-90/9, Argonne National Laboratory, 1990.

[OSR93a] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language (Draft)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[OSR93b] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System (Draft)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[SOR93] N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.