A Very Brief Introduction to PVS

Tamarah Arons
tamarah@wisdom.weizmann.ac.il

May 29, 2002

The PVS system is an extensive, well-documented, deductive verification system. It is impossible to summarize all its features in a short document. The purpose of this document is to briefly introduce the user to the PVS system and some of the more frequently used commands. The full PVS documentation is available at ~verify/PVS2-4.1/Doc/ .

In Section 1 we give an overview of the sequent calculus which forms the theoretical basis for the PVS proof system. In Section 2 we define some basic concepts in the PVS system, in Section 3 overview the specification language, and in Section 4 explain how to run PVS. Section 5 details some useful prover features. Appendices A and B summarize the most frequently used system and prover commands, respectively.

1 The Logic of PVS

In this section we examine the theoretical basis underlying the PVS logic.

The PVS 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 the leaves are recognized as true. Each node of the proof tree is a *proof goal*, from which its offspring nodes follow by a *proof step*.

Each proof goal is a *sequent* consisting of a sequence of *antecedent* formulas, and a sequence of *consequent* formulas. Together, the antecedents and the consequents are called the *sequent formulas*.

We will let Γ and Δ represent (finite) sequences of formulas, and p q, p_i and q_i, represent individual formulas.

Sequents are represented in the form $p_1, p_2, \ldots, p_n \vdash q_1, q_2, \ldots, q_m$, where the formulas preceding the turnstile are the antecedents, and the formulas after the turnstile are the consequents. The intuitive interpretation of a sequent is that

$$\forall free : (p_1 \wedge p_2 \wedge \ldots \wedge p_n) \rightarrow (q_1 \vee q_2 \vee \ldots \vee q_m)$$

where *free* denotes the free (unbound) variables.

The system uses *backwards reasoning*. That is, each proof step results in sequent(s) that are at least as strong as the previous one. The root of the tree is the sequent $\vdash q$, where q is the theorem to be proved.

1.1 Axioms

Three axioms are used to recognize leaf sequents as true[1]. They are:

$$\begin{array}{lllll} \text{A1}: & \Gamma, p & \vdash & \Delta, p \\ \text{A2}: & \Gamma & \vdash & \Delta, \text{T} \\ \text{A3}: & \Gamma, \text{F} & \vdash & \Delta \end{array}$$

That is, A1 asserts that a sequent is true if any antecedent is the same as any consequent. (Clearly, it is always the case that $\Gamma \wedge p \rightarrow \Delta \vee p$). Axiom A2 states that a sequent is true if any of its consequents are true and axiom A3 that a sequent is true if any of its antecedents are false.

1.2 Proof Rules

In this section we present some of the proof rules used to add subtrees to the proof tree.

1.2.1 Propositional Rules

We consider rules for the basic operators of conjunction, disjunction and negation, noting that implications can be converted to these operators.

We first present four *non-expanding* propositional rules. These are rules in which a single sequent is derived from a parent[2].

$$\frac{\Gamma \vdash \Delta, p \vee q}{\Gamma \vdash \Delta, p, q} \qquad \frac{\Gamma, p \wedge q \vdash \Delta}{\Gamma, p, q \vdash \Delta}$$

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

The first two rules allow consequent disjuncts and antecedent conjuncts to be flattened, so that two formulas are obtained from one.

The second two rules allow negated antecedent or consequent formulas to become non-negated consequent or antecedent formulas, respectively.

We next present two *expanding* propositional rules[3]. These rules derive a subtree of two sequents.

$$\frac{\Gamma, p \vee q \vdash \Delta}{\Gamma, p \vdash \Delta \qquad \Gamma, q \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, p \wedge q}{\Gamma \vdash \Delta, p \qquad \Gamma \vdash \Delta, q}$$

That is, the first rule states that in order to prove that $\Gamma \wedge (p \vee q) \rightarrow \Delta$ it is sufficient to prove both that $\Gamma \wedge p \rightarrow \Delta$ and that $\Gamma \wedge q \rightarrow \Delta$.

[1]The application of these three axioms corresponds roughly to the PVS `assert` command.
[2]The non-expanding propositional rules correspond to the PVS `flatten` command.
[3]The expanding propositional rules correspond to the `split` command.

1.2.2 Quantifier Rules

The notation $p\{x \leftarrow t\}$ represents the result of substituting the term t for all free occurrences of x in p.

The following two rules represent the process of *skolemization*[4]. They require that t be a new constant that does not occur in the sequent.

$$\frac{\Gamma, (\exists x : p) \vdash \Delta}{\Gamma, p\{x \leftarrow t\} \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, (\forall x : p)}{\Gamma \vdash \Delta, p\{x \leftarrow t\}}$$

The next two rules represent the effect of *instantiation*[5].(There is no requirement that t be a new constant.)

$$\frac{\Gamma, (\forall x : p) \vdash \Delta}{\Gamma, (\forall x : p), p\{x \leftarrow t\} \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, (\exists x : p)}{\Gamma \vdash \Delta, (\exists x : p), p\{x \leftarrow t\}}$$

1.2.3 Strengthening Rules

The above rules neither strengthened or weakened the sequent. We present two rules which allow a stronger sequent to be derived from a weaker one by removing formulas[6].

$$\frac{\Gamma, p \vdash \Delta}{\Gamma \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, p}{\Gamma \vdash \Delta}$$

2 The PVS Verification System : Basic Definitions

Specification files are ordinary text files, written in the PVS specification language, generally prepared using the PVS emacs editor interface. They have a `.pvs` extension. All such files must have the structure

```
theory-name: THEORY
 BEGIN


 END theory-name
```

Where theory-name is the name of the theory, and should preferably match the file name. The body of the theory is placed between the begin and end statements.

Proof files (`.prf` extension) save proofs that have been composed. They are also text files, but it is not advisable to try to edit them in any way.

A *context* is a set of specification and proof files found in one directory. The binary `.pvscontext` file saves the state of the verification from one verification attempt to another. For this reason, all files relating to one proof should be in one directory. Files relating to unrelated proofs are best kept in a separate directory with a separate context.

The PVS *interface* is through an emacs editor. In different buffers in the editor one can write specifications and run proofs. PVS commands are entered in the emacs mini-buffer window and are preceded by `M-x` (alt-x or esc-x).

[4]Skolemization is effected in PVS by `skolem!` and related commands.

[5]The form of instantiation presented here is equivalent to the `inst-cp` command.

[6]The PVS `delete` and `hide` commands explicitly delete and hide formulas, respectively. Other commands, such as `inst`, may also hide formulas.

3 The Specification Language

The PVS specification is built on higher order logic.

Variables and constants have types – some types are inbuilt, and the user can build their own (including arrays, records, etc). PVS also allows the user to use uninterpreted types.

Frequently used inbuilt types include `nat` , `int` , `real` , `bool` .

Constants are declared as being elements of a type e.g.

```
z :  nat
```

declares constant z of type natural.

To declare a *variable* one must simply precede the type with the `VAR` keyword e.g.

```
x :  VAR nat
```

There are many different possibilities in declaring types. One of the most basic, and most useful, are *tuples* e.g.

```
T1 :  TYPE = [nat, nat]
```

Records are tuples with labeled fields e.g.

```
T2 :  TYPE = [# first, second :  nat #]
```

Both T1 and T2 contain two natural numbers. In the first case the fields are unlabeled, in the second they are labeled.

Another useful type is a *function* type. This is a mapping from a domain to a range. e.g.

```
A1 :  TYPE = [upto[10] -> T1]
```

is an array A1[0..10] of elements of type [nat, nat].

The basic logical constructs are `OR` (also written \/), `AND` (also &, /\), `NOT` (~), =, \= (disequality), `IMPLIES` (=>) and `IFF` (<=>). The universal and existential quantifiers are `FORALL` and `EXISTS` , respectively.

3.1 Example - Hotel Reservations

As a very simple example we consider a hotel reservation system. For every room, for every date, the name of the person reserving the room is stored in the register. If a room is free, then it is recorded simply that it is registered to the constant name `free` .

We define this as

```
reservation: THEORY
 BEGIN

  room: TYPE
  date: TYPE
  name: TYPE
  free: name
  reservations: TYPE = [room, date → name]

 END reservation
```

The room, date and name types are all uninterpreted. The reservations type is a mapping from room and date to name. `free` is a constant of type name.

It is also useful to declare interpreted functions. For example, we can define a function which given a register, adds a booking and returns the updated register:

```
reserve(r: room, d: date, n: name, register: reservations): reservations =
    register WITH [(r, d) := n]
```

This demonstrates the use of the very important `WITH` expression. `WITH` is an override expression used to modify the contents of a function, tuple, or record. `register WITH [(r, d) := n]` is the same as `register` except that the argument at value (r, d) is replaced with n.

Instead of declaring the types when using variables (e.g. r, n, d above) we can define them as global variables once, and then use them without stating the types every time. Thus, we define the function cancel which cancels the reservation for a room, and the predicate reserved which returns the true if a room is reserved.

```
r: VAR room
d: VAR date
n: VAR name
register: VAR reservations

cancel(r, d, register): reservations = register WITH [(r, d) := free]

reserved(r, d, register): bool = register(r, d) ≠ free
```

4 Running PVS

To run PVS make and enter the directory you plan to work from. You can then open PVS by calling the executables at ~verify/PVS2-4.1/pvs for sun machines, or ~verify/PVS2-4.1/pvs_linux for linux machines.

This should open up an emacs window. You will be asked whether to create a new context - answer yes.

We will continue with the reservation example. The .pvs file can be copied from ~verify/Course02a/, or you can type it in.

Open the desired file (emacs command `C-x, C-f`).

It is important to *typecheck* the file. Typechecking parses the file, and checks for semantic errors (e.g. undeclared names). We typecheck the file by typing `M-x tc` . PVS reports that one type correctness condition, TCC, was generated, and that it is unproved. To view the TCCs type `M-x show-tccs` . In this case the typecheck is trivial, and PVS can discharge it if you ask it to prove the typechecks – `M-x tcp` . When PVS cannot discharge the typecheck automatically, it must be done manually. This can be done by placing the cursor on the TCC definition in the buffer generated by `M-x show-tccs` , and proving it as you would any other formula.

We prove a simple lemma : if a reservation for a room is canceled then the room is not reserved:

```
canceled_not_reserved: LEMMA
  ∀ r, d, register: ¬ reserved(r, d, cancel(r, d, register))
```

To prove this lemma put the cursor on the definition of the lemma and type `M-x pr` .

You will get a new buffer, labeled *pvs*, containing the sequent:

```
canceled_not_reserved :

  |-------
{1}   FORALL r, d, register: NOT reserved(r, d, cancel(r, d, register))
```

PVS presents sequents as a list of negatively numbered antecedents above above a turnstile symbol `|-------`, and a list of positively numbered consequent formulas below it.

The `Rule?` prompt indicates that PVS is waiting for a new command to be entered. Prover commands use lisp-like syntax, and are always enclosed in round brackets.

We can *skolemize* and simplify by typing `(skosimp*)`. This removes quantified variables and replaces them by skolem constants. Skolem constants have "!1" added to the variable name.

We can now expand out the definitions of reserved and cancel by entering `(expand "reserved")` and then `(expand "cancel")`. The proof is complete.

In fact, the `(grind)` command can complete this proof in one step. `(grind)` expands out expressions, skolemizes, instantiates and simplifies. It is often useful in very small proofs, or towards the ends of large proofs. However, when it does not complete a proof, it can be counterproductive, generating many similar subgoals.

We consider a second lemma,

```
  is_reserved: LEMMA
    ∀ r, d, n, register: reserved(r, d, reserve(r, d, n, register))
```

The lemma is intended to show that after reserving, a room is reserved. The reader is invited to try to prove it. It will soon be apparent that this cannot be done – in fact the premise is false. The reader is encouraged to try to understand why this is the case and how it could be rectified.

4.1 Example : Queues

As a second example, we consider a simple queue structure modeled as an array[nat] of entries. The queue has a head pointer, pointing to its oldest element. It also has a size field, indicating how many entries are currently occupied. The occupied entries are stored in an array entries, from positions head to head + size - 1, inclusive.

That is, consider a queue with 2 entries, A and B, and its head at position 4. entry[4] = A and entry[5] = B. The queue size is 2. The contents of entry[x] is irrelevant for any $x \neq 4, 5$.

This theory, presented in Fig. 1, can also be downloaded from ~verify/Course02a/

A few notes on the data structures :

- `QUEUE_ENTRY` is an uninterpreted type. We know nothing about it.
- `QUEUE_TYPE` is a record with three fields, `head`, `size` and `entry`.

  There are two means of accessing record fields – using a ‘, and using brackets. So, `queue‘size` and `size(queue)` both return the size field of record queue.

- `pushQentry` , `empty` , `popQentry` , `occupied` and `inQ` are all functions.

  Functions cannot modify their arguments.

  Procedures are functions that return a boolean value.

We examine how the lemmas can be proved.

We start with `empty_no_occupied` . This lemma states that a queue is empty if and only if none of its buffers are occupied.

We first skolemize `(skosimp*)` to remove the quantification. We then expand out the terms : `(expand "empty")` , `(expand "occupied")` , arriving at the sequent

```
  |-------
{1}   size(queue!1) = 0 IFF
       (FORALL qPoint:
          NOT (qPoint >= queue!1`head AND
                qPoint < queue!1`head + queue!1`size))
```

We use the `(split)` command to split the `IFF` statement into its two directions, generating two subgoals, one for the "if", and one for the "only if".

You can view the second subgoal by typing `(postpone`), or Tab Shift-p.

The first subgoal,

```
  |-------
{1}   size(queue!1) = 0 IMPLIES
       (FORALL qPoint:
          NOT (qPoint >= queue!1`head AND
                qPoint < queue!1`head + queue!1`size))
```

can be flattened `(flatten)` , into a sequent where `size(queue!1) = 0` is assumed:

```
{-1}  size(queue!1) = 0
  |-------
{1}   FORALL qPoint:
        NOT (qPoint >= queue!1`head AND
              qPoint < queue!1`head + queue!1`size)
```

After skolemizing `(skosimp*)` , an `(assert)` completes the proof.

PVS now returns to the second goal. We again flatten it, generating the sequent

```
{-1}  FORALL qPoint:
        NOT (qPoint >= queue!1`head AND
              qPoint < queue!1`head + queue!1`size)
  |-------
{1}   size(queue!1) = 0
```

The universal quantifier must be instantiated, and we ask PVS to do so by typing `(inst?)` . Unfortunately, PVS guesses the instantiation incorrectly, so we must instantiate manually.

We undo the instantiation by typing `(undo)` or Tab u. The correct instantiation is queue!1'head. (The antecedent -1 asserts that it is NOT the case that qPoint >= queue!1'head and qPoint < queue!1'head + queue!1'size. In other words, qPoint < queue!1'head or qPoint >= queue!1'head + queue!1'size. Setting qPoint = queue!1'head therefore implies that queue!1'size = 0.)

We instantiate with the command `(inst - "queue!1‘head")`. The first parameter to the `inst` command is the sequent number, the second the value to be instantiated. We could have used "-1" as the sequent number. By typing "-" we tell PVS to substitute into the first matching antecedent. In general, this is more robust than specifying exact line numbers (should the specification or proof later be changed, the line numbers may change and the proof might no longer work if sequent numbers are fully specified.)

The proof is now completed with `(assert)`.

We consider the second lemma, `pushed_entry_in_queue`. We first try to complete it using `grind`. Unfortunately, this does not work – `grind` instantiates the existential quantifier incorrectly.

We try `grind` without instantiation by typing `(grind :if-match nil)`. The parameter :if-match determines under which conditions `grind` will do instantiations. By setting it to nil, we prevent `grind` from instantiating.

The resulting formula has all the expressions expanded. (The same effect could have been obtained using the `expand` command).

```
  |-------
{1}   EXISTS (index: nat):
        (index >= queue!1‘head AND index < 1 + queue!1‘head + queue!1‘size)
         AND
         entry(queue!1) WITH [(head(queue!1) + size(queue!1)) := qEntry!1]
             (index)
          = qEntry!1
```

The correct instantiation is `(inst + "queue!1‘head + queue!1‘size")`. The assert command will now complete the proof.

The third lemma is left as an exercise.

```
queue: THEORY
 BEGIN

  QUEUE_ENTRY: TYPE

  QUEUE_TYPE: TYPE = [# head: nat, size: nat, entry: [nat → QUEUE_ENTRY] #]

  queue: VAR QUEUE_TYPE
  qEntry: VAR QUEUE_ENTRY
  qPoint: VAR nat

  % Returns true if a queue is empty
  empty(queue): bool = size(queue) = 0

  % Pushes an entry onto a queue and returns the updated queue
  pushQentry(queue, qEntry): QUEUE_TYPE =
      queue WITH [size := queue‘size + 1,
                  entry := entry(queue) WITH [(head(queue) + size(queue)) := qEntry]]

  % Pops an entry from a queue. It returns the new queue.
  popQentry(queue): QUEUE_TYPE =
      IF ¬ empty(queue)
        THEN queue WITH [head := head(queue) + 1, size := size(queue) − 1]
      ELSE queue
      ENDIF

  % Returns true if entry qPoint of the queue is occupied,
  % i.e. lies between head and head + size
  occupied(queue, qPoint): bool =
      qPoint ≥ queue‘head ∧ qPoint < queue‘head + queue‘size

  % Checks whether there is an occupied entry in the queue with value
  inQ(queue, qEntry): bool =
      ∃ (index: nat): occupied(queue, index) ∧ queue‘entry(index) = qEntry

  % If a queue is empty, no entries are occupied
  empty_no_occupied: LEMMA
    ∀ queue: empty(queue) IFF (∀ qPoint: ¬ occupied(queue, qPoint))

  % After pushing an entry onto a queue, the entry is in the queue
  pushed_entry_in_queue: LEMMA
    ∀ queue, qEntry: inQ(pushQentry(queue, qEntry), qEntry)

  % If one pushes an entry onto an empty queue, and then pops one off,
  % the queue is again empty.
  empty_push_pop_empty: LEMMA
    ∀ queue, qEntry: empty(queue) ⊃ empty(popQentry(pushQentry(queue, qEntry)))

 END queue
```

Figure 1: Theory queue.pvs

5 Some Prover Features

In this Section we will discuss some useful features of the PVS environment.

5.1 Stepping Through a Proof

PVS allows you to walk through, or step through, a proof, viewing the effects of each command. The step through command `M-x step-proof` initiates a proof from the beginning, and also opens a Proof buffer in which the last saved proof is displayed.

The first command in the buffer is highlighted. Typing Tab 1 will cause it to be executed. Typing Tab n, for number n, will cause the next n commands to be implemented. If you put the cursor in the Proof buffer, Tab 1 will execute the next command after the cursor, and not the highlighted command.

To break a continuous execution generated by Tab n, type C-g. This will complete the currently executed command and highlight the next one.

At any point, you can enter commands at the Rule? prompt rather than from the proof buffer.

5.2 X-Displaying a Proof

One can generate an X-display of a proof. This is a window graphically showing the current proof as a tree. Every sequent in the proof is represented by a $\vdash$ symbol. The root, at the top, is the initial sequent. The proof commands used to create the proof are shown between the $\vdash$ symbols. To see the full text of a sequent click on the $\vdash$ symbol.

Status information about the proof is indicated by colors: blue indicates a completed branch, brown the current branch, purple the current sequent, and black any incomplete branch which is not the current branch.

The proof tree is automatically updated as the proof is run.

To initiate a proof with its X-display enter `M-x xpr` when the cursor is on the formula. Similarly, step through a proof with its X-display using `M-x x-step-proof` .

To open an X-display for a proof already in progress one must use the main PVS menu. From the PVS menu go to display-commands then show-current-proof.

Warning: The X-display slows the proof down significantly. Furthermore, trying to access the display (either to start it, or to view sequents etc) while PVS is running (i.e. (ILISP : run) displayed on the *pvs* buffer label line, rather than (ILISP : ready)) can cause PVS to be interrupted. The result of this is that PVS must be reset, and the proof restarted!

5.3 Hiding and Revealing Formulas

When a formula is instantiated, its uninstantiated copy is "hidden" by PVS. Hidden formulas can be viewed by typing `M-x show-hidden-formulas` .

The formula can then be revealed by typing `(reveal fnum)` where fnum is the sequent number of the desired formula.

Formulas which are not needed can be hidden by typing `(hide fnum)` .

(Both the hide and reveal commands can take lists of formulas).

5.4 Strategies

The PVS user can construct strategies which combine proof rules together into more powerful proof rules. Strategies are written in a lisp based language, and can include recursion, branching and backtracking. (C.f. Chapter 5 of *PVS Prover Guide*).

At this point we will discuss only two basic, and useful, constructs: `then` and `repeat` .

The then keyword is used in the format (**`then`** step1, rest-steps). Step1 is applied to the current goal and then rest-steps to each of the sequents generated. For example, `(then (split -1)(assert))` will split formula -1 and then apply `assert` to each of the sequents generated.

Iteration can be effected by using (**`repeat*`** step). Rule step is replied iteratively along all subgoals until its application has no effect e.g. `(repeat* split)` would repeatedly split until there is nothing more to split.

A Summary of Some Often-Used System Commands

We first list some PVS system commands. These commands are generally entered while the .pvs specification is the current buffer. Command relating to a single formula require that the cursor be on the formula. More information can be found in the *PVS System Guide.*

Command	Alias	Comments
		Exiting and Interrupting PVS
M-x exit-pvs	C-x C-c	Exit PVS
	C-c C-c	Interrupt PVS process (useful if PVS is taking unreasonably long). Entering `(restore)` allows you to resume the proof.
M-x reset-pvs	C-z C-g	Abort PVS and resynchronize. You will have to restart the proof from the beginning
		Initiating and Stepping Through Proofs
M-x prove, M-x pr	C-c p	Prove formula pointed to by cursor
M-x step-proof	C-c C-p	Step through an existing poof. (C.f. Section 5.1)
M-x x-prove	M-x xpr	Prove with X-display. (C.f. Section 5.2)
M-x x-step-proof		Step through with X-Display
M-x prove-theory	M-x prt	Reruns all proofs in the theory (non-interactively).
		Typechecking
M-x typecheck	M-x tc	Typecheck theories in current buffer
M-x typecheck-prove	M-x tcp	Typecheck theories and try to prove TCCs
M-x show-tccs	M-x tccs	Show the TCCs of the current theories. TCCs can be proved from the new buffer created
		Editing and Viewing Proofs
M-x edit-proof		Edit / view the proof of the formula
M-x install-proof	C-c C-i	Installs a proof on a formula The format must be like that obtained when editing proofs, not the format in the .prf file. You can highlight a proof buffer of one formula and install it on a second
		Proof Status
M-x status-proof	M-x sp	Status of formula at cursor. proved – complete = fully proved, proved – incomplete = formula is proved but depends on some unproved formula unchecked = changes since the proof succeeded may invalidate it. You should rerun the proof (M-x pr). untried = proof never attempted unfinished = proof attempted, but never completed.
M-x status-proof-theory	M-x spt	Status of all formulas in theory
M-x status-proof-importchain	M-x spi	Status of formulas on importchain
M-x status-proofchain	M-x spc	Displays proofchain of formula at cursor i.e. lists the formulas on which it is dependent
		Context and Prelude Commands
M-x change context	M-x cc	Switch to a new context (new directory)
M-x load-prelude-library		Allows the current context to use all theories in the loaded context (directory).

B Summary of Some Often-Used Prover Commands

This appendix lists a selection of the more frequently used prover commands. More information can be obtained from the *PVS Prover Guide*.

Note that PVS is case insensitive regarding prover commands i.e. you may type the command using any combination of upper and lower case letters.

B.1 Parameters

Proof commands take a list of zero or more required and optional parameters. Each optional parameter has an associated default value. In this appendix optional parameters are bracketed with their default values. Required parameters are not bracketed.

We have not listed here all parameters to the commands, but only those that are frequently used. When we have omitted some intermediate parameters, this is node by "..." in the parameter listing.

When invoking a proof command actuals are associated with formal parameters according to the order in which they are given. If there are fewer actuals than formals, then those parameters for which no actual was provided are bound with their default values. To give parameter values out of order, or without specifying a previous parameter in the list, one must name the formal when giving the actual.

For example, consider the proof command
`replace` Parameters : fnum, (fnums *), (dir LR), (hide? nil)

This command takes the formula at line fnum (it must be an equality) and replaces it in formulas fnums. The replacement is in direction LR (left to right), and the formula is hidden if hide? is set to t.

Consider, for example, a sequent with an antecedent

```
[-3] i!1 = j!1
```

Command `(replace -3)` will replace i!1 with j!1 in all sequents.

Command `(replace -3 (2, 3) :hide?  t)` will replace i!1 with j!1 only in sequents 2 and 3, and then hide -3.

Command `(replace -3 :dir rl :fnums +)` will replace j!1 with i!1 in all consequents (reversing the direction of replacement).

B.2 Formula and Truth Value Selection

Many commands have arguments taking the number(s) of the sequent formulas where the rule is to be applied. By convention, a single formula is called "fnum" (for "sequent formula number"), a list is "fnums".

The list of antecedent sequent formulas can be indicated by "-", the list of consequent sequent formulas by "+", and the list of all sequent formulas by "*".

Where truth values are requested, "t" stands for true, "nil" for false.

Command	Parameters	Comments
Control Rules		
quit		Terminates the proof attempt
postpone		Go to next remaining goal. Alias: Tab shift-p
undo	(to 1)	Undoes commands.
		If “to” is a number, n, the last n commands are undone
		If “to” is a proof rule, undoes the proof to the last
		occurrence of this proof rule
		(undo undo) Undoes the undo, if it was the last command executed.
		Alias : Tab u
Propositional Rules		
case	exprs	Introduces case split.
		On one branch exprs is assumed to be true, on another false.
		exprs must be in double-quotes e.g. (case “i!1 = j!1”)
split	(fnum *)	Conjunctive splitting.
		Splits one formula to generate 2 subgoals :
		Antecedent $A \vee B$ into antecedent A and antecedent B.
		Antecedent A implies B into antecedent B and consequent A
		Consequent $A \wedge B$ into consequent A and consequent B
		Antecedent $IF(A, B, C)$ into antecedents $A \wedge B$ and $\neg B \wedge C$
		Consequent $IF(A, B, C)$ into consequents A implies B and $\neg A$ implies C
flatten	(fnums *)	Disjunctive Simplifications. Generates 1 subgoal converting :
		Antecedent formula $\neg A$ into consequent A
		Antecedent formula $A \wedge B$ into 2 antecedent formulas, A and B.
		Consequent formula $\neg A$ into antecedent A
		Consequent formula $A \vee B$ into two consequent formulas, A and B.
iff	(fnums *)	Converts boolean equality (=) into equivalence (iff)
lift-if	(fnums *)	Lifts the left-most if or cases statement to the topmost level
Quantifier Rules		
skosimp*		Repeatedly skolemizes then flattens
inst	fnum, terms	Instantiates formula fnum with terms in terms
inst-cp	fnum, terms	Instantiates, retaining a copy of the uninstantiated formula
inst?	(fnums *),...,	PVS chooses the terms to instantiate into formula fnums.
	(copy? nil)	Retains a copy of uninstantiated formula if copy? is t.
Equality Rules		
replace	fnum,	Rewriting using equalities
	(fnums *),	Formula fnum must be of form $l = r$.
	(dir LR),	All occurrences of l in formulas fnums are replaced with r.
	(hide? nil),	Instead, replace r with l if dir RL specified.
		Formula fnum is hidden if hide? is t.
replace*	fnums	Replace formulas in fnums into all formulas.

Using Definitions and Lemmas		
expand	name, (fnum *)	Expands the definition of name in formulas fnum e.g. (expand ”empty” 1 -3) expands empty in formulas 1 and -3
expand*	names	Expands all occurrences of all expressions listed in names
lemma	name	Introduces an instance of lemma name. e.g. (lemma ”empty_no_occupied”) introduces forall queue: ...
Simplification with Decision Procedures		
simplify		Simplifies using decision procedures
assert		More aggressive simplification. (Calls simplify.)
grind	... (if-match t)	Expands all definitions, replaces, simplifies, skolemizes, splits etc. Can be used to automatically complete a proof. Performs instantiation unless if-match is set to nil.
ground		Invokes propositional simplification, splits and asserts.
Making Type Constraints Explicit		
typepred	exprs	If exprs is of type p, antecedent p(exprs) is introduced. E.g. (typepred ”head(queue!1)”) introduces head(queue!1) >= 0
Structural Rules – Hiding and Revealing Formulas		
hide	fnums	Moves formulas fnums to the Hidden buffer (C.f. Section 5.3)
hide-all-but	keep-fnums, (fnums *)	Hides all the formulas in fnums except those in keep-fnums
reveal	fnums	Copies formulas fnums from Hidden buffer to sequent.
Annotation Rules – Labeling		
label	string, fnums, (push? nil)	Attaches the label string to formulas fnums If push? is t, then any previous label is retained. Else, the label replaces previous labels
unlabel	(fnums *)	Removes all labels from formulas fnums