A Tutorial on Specification and Verification Using PVS (Beta Release)

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

March 31, 1993

Abstract

PVS stands for "Prototype Verification System." It consists of a specification language integrated with support tools and a theorem prover. PVS tries to provide the mechanization needed to apply formal methods both rigorously and productively.

This tutorial serves to introduce PVS. In the first section, we briefly sketch the purposes for which PVS is intended and the rationale behind its design, mention some of the uses that we and others are making of it, and explain how to get a copy of the system. In Section 2, we use a simple example to briefly introduce the major functions of PVS; Sections 3 and 4 then give more detail on the PVS language and theorem prover, respectively, also using examples. The PVS language, system, and theorem prover each have their own reference manuals, [1,2,3] which you will need to study in order to make productive use of the system. A pocket reference card, summarizing all the features of the PVS language, system, and prover is also available.

This tutorial does not introduce the general ideas of formal methods, nor explain how formal specification and verification can best be applied to various problem domains; rather, its purpose is to introduce some of the more unusual and powerful capabilities that are provided by PVS. Consequently, this document, and the examples we use, are somewhat technical and are most suitable for those who already have some experience with formal methods and wish to understand how PVS provides mechanized support for some of the more challenging aspects of formal methods.

---

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

[2] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

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

Contents

1 Introducing PVS

PVS stands for "Prototype Verification System."[4] It consists of a specification language integrated with support tools and a theorem prover. PVS tries to provide the mechanization needed to apply formal methods both rigorously and productively.

The specification language of PVS is a higher-order logic with a rich type-system, and is quite expressive; we have found that most of the mathematical and computational concepts we wish to describe can be formulated very directly and naturally in PVS. Its theorem prover, or proof checker (we use either term, though the latter is more correct), is both interactive and highly mechanized: the user chooses each step that is to be applied and PVS performs it, displays the result, and then waits for the next command. PVS differs from most other interactive theorem provers in the power of its basic steps: these can invoke decision procedures for arithmetic, automatic rewriting, induction, and other relatively large units of deduction; it differs from other highly automated theorem provers in being directly controlled by the user. We have been able to perform some significant new verifications quite economically using PVS; we have also repeated some verifications first undertaken in other systems and have usually been able to complete them in a fraction of the original time (of course, these are previously solved problems, which makes them much easier for us than for the original developers).

PVS is the most recent in a line of specification languages, theorem provers, and verification systems developed at SRI, dating back over 20 years. That line includes the Jovial Verification System [EGMS79], the Hierarchical Development Methodology (HDM) [RL76, RLS79], STP [SSMS82], and EHDM [MSR85, RvHO91]. We call PVS a "Prototype Verification System," because it was built partly as a lightweight prototype to explore "next generation" technology for EHDM, our main, heavyweight, verification system. Another goal for PVS was that it should be freely available, require no costly licenses, and be relatively easy to install, maintain, and use. Development of PVS was funded entirely by SRI International

In the rest of this introduction, we briefly sketch the purposes for which PVS is intended and the rationale behind its design, mention some of the uses that we and others are making of it, and explain how to get a copy of the system. In Section 2, we use a simple example to briefly introduce the major functions of PVS; Sections 3 and 4 then give more detail on the PVS language and theorem prover, respectively, also using examples. The PVS language, system, and theorem prover each have their own reference manuals [OSR93a, SOR93, OSR93b], which you will need to study in order to make productive use of the system. A pocket reference card, summarizing all the features of the PVS language, system, and prover is also available.

The purpose of this tutorial is not to introduce the general ideas of formal methods, nor to explain how formal specification and verification can best be applied to various problem domains; rather, its purpose is to introduce some of the more unusual and powerful

[4]A number of people have contributed significantly to the design and implementation of PVS. They include David Cyrluk, Friedrich von Henke, Pat Lincoln, Steven Phillips, Mandayam Srivas, and Carl Witty. We also thank Mark Moriconi, Director of the SRI Computer Science Laboratory, for his support and encouragement.

capabilities that are provided by PVS. Consequently, this document, and the examples we use, are somewhat technical and are most suitable for those who already have some experience with formal methods and wish to understand how PVS provides mechanized support for some of the more challenging aspects of formal methods.

1.1 Design Goals for PVS

PVS provides mechanized support for Formal Methods in Computer Science. "Formal Methods" refers to the use of concepts and techniques from logic and discrete mathematics in the development of computer systems, and we assume that you already have some familiarity with this topic.

Formal methods can be undertaken for many different purposes, in many different ways and styles, and with varying degrees of rigor. The earliest formal methods were concerned with proving programs "correct": a detailed specification was assumed to be available and assumed to be correct, and the concern was to show that a program in some concrete programming language satisfied the specification. If this kind of program verification is your interest, then PVS is not for you. You will probably be better served by a verification system built around a programming language, such as Penelope [Pra92] (for Ada), or by some member of the Larch family [GHW85]. Similarly, if your interests are gate-level hardware designs, you will probably do best to consider model-checking and automatic procedures based on BDDs [BCM$^+$90].

The design of PVS was shaped by our experience in doing or contemplating early-lifecycle applications of formal methods. Many of the larger examples we have done concern algorithms and architectures for fault-tolerance (see [ORSvH93] for an overview). We found that many of the published proofs that we attempted to check were in fact, incorrect, as was one of the important algorithms. We have also found that many of our own specifications are subtly flawed when first written. For these reasons, PVS is designed to help in the detection of errors as well as in the confirmation of "correctness." One way it supports early error detection is by having a very rich type-system and correspondingly rigorous typechecking. A great deal of specification can be embedded in PVS types (for example, the invariant to be maintained by a state-machine can be expressed as a type constraint), and typechecking can generate proof obligations that amount to a very strong consistency check on some aspects of the specification.[5]

Another way PVS helps eliminate certain kinds of errors is by providing very rich mechanisms for conservative extension—that is, definitional forms that are guaranteed to preserve consistency. Axiomatic specifications can very effective for certain kinds of problem (e.g., for stating assumptions about the environment), but axioms can also introduce inconsistencies—and our experience has been that this does happen rather more often than one would wish. Definitional constructs avoid this problem, but a limited repertoire of such constructs (e.g., requiring everything to be specified as a recursive function) can lead to excessively constructive specifications: specifications that say "how" rather than "what." PVS provides both the freedom of axiomatic specifications, and the safety of a generous

[5]As a way to further strengthen error checking, we are thinking of adding dimensions and dimensional analysis to the PVS type system and typechecker.

collection of definitional and constructive forms, so that users may choose the style of specification most appropriate to their problems.[6]

The third way that PVS supports error detection is by providing an effective theorem prover. Our experience has been that the act of trying to prove properties about specifications is the most effective way to truly understand their content and to identify errors. This can come about incidentally, while attempting to prove a "real" theorem, such as that an algorithm achieves its purpose, or it can be done deliberately through the process of "challenging" specifications as part of a validation process. A challenge has the form "if this specification is right, then the following ought to follow"—it is a test case posed as a putative theorem; we "execute" the specification by proving theorems about it.[7]

1.2 Uses of PVS

PVS has so far been applied to several small demonstration examples, and a growing number of significant verifications. The smaller examples include the specification and verification of ordered binary tree insertion, the Boyer-Moore majority algorithm [BM91], an abstract pipelined processor [SGGH91], Fischer's real-time mutual exclusion protocol, and the Oral messages protocol for Byzantine agreement. Examples of this scale can typically be completed within a day. More substantial examples include the correspondence between the programmer and RTL level of a simple hardware processor, the correctness of a real-time railroad crossing controller, a variant of the Schröder-Bernstein theorem, and the correctness of a distributed agreement protocol for a hybrid fault model consisting of Byzantine, symmetric, and crash faults. These harder examples can take from several days to a week.

Currently, PVS is being applied to the requirements specification of selected aspects of the control software for NASA's space shuttle project.

1.3 Getting and Using PVS

At the moment, PVS is readily available only for Sun SPARC workstations, although versions of the system do exist for the IBM Risc 6000 (under AIX) and DECSystems (under Ultrix). PVS is implemented in Common Lisp (with CLOS), and has been ported to Lucid, Allegro, AKCL, and CMULISP. All versions of PVS require Gnu Emacs, which must be obtained separately. It is not particular about the window system, as long as it supports Gnu Emacs. In addition, LaTeX and an appropriate viewer are needed to support certain optional features of PVS.

PVS is quite large, requiring about 30 megabytes of disk space. In addition, any system on which it is to be run should have a minimum of 40 megabytes of swap space and 12 megabytes of real memory (more is better).

[6] Unlike Ehdm, PVS does not provide special facilities for demonstrating the consistency of axiomatic specifications. We do expect to provide these in a later release, but using a different approach than Ehdm.

[7] Directly executable specification languages (e.g., [AJ90, HI88]) support validation of specifications by running conventional test cases. We think there can be merit in this approach, but that it should not compromise the effectiveness of the specification language as a tool for deductive analysis; we are considering supporting an executable subset within PVS.

To obtain the PVS system, send a request to `pvs-request@csl.sri.com`, and we will provide further instructions for obtaining a tape or for getting the system by FTP. All installations of PVS must be licensed by SRI. A nominal distribution fee is charged for tapes; there is no charge for obtaining PVS by FTP.

2 A Brief Tour of PVS

In this section we introduce the system by developing a theory and doing a simple proof. This will introduce the most useful commands and provide a glimpse into the philosophy behind PVS. You will get the most out of this section if you are sitting in front of a workstation (or terminal) with PVS installed. In the following we assume familiarity with Sun Unix and GNU EMACS.

Start by going to a UNIX shell window and creating a working directory (using `mkdir`). Next, connect (`cd`) to that working directory and start up PVS by typing `pvs`.[8] This command executes a shell script which runs GNU EMACS, loads the necessary PVS EMACS extensions, and starts the PVS lisp image as a subprocess.[9] After a few moments, you should see the welcome screen indicating the version of PVS being run, the current directory, and instructions for getting help. You may be asked whether you want to create a new context in the directory; answer `yes` unless it is the wrong directory or you don't have write permission there, in which case you should answer `no` and provide an alternative directory when prompted.

PVS uses EMACS as its interface by extending EMACS with PVS functions, but all the underlying capabilities of EMACS are available. Thus the user can read mail and news, edit nonPVS files, or execute commands in a shell buffer in the usual way.

In the following, PVS EMACS commands are given first in their long form, followed by an alternative abbreviation and/or key binding in parentheses. For example, the command for proving in PVS is given as `M-x prove` (`M-x pr`, `C-c p`). This command can be entered by typing the `Escape` key, then an `x`[10] followed by `prove` (or `pr`) and the `Return` key. Alternatively, hold the `Control` key down while typing a `c`, then let go and type a `p`. The `Return` key does not need to be pressed when giving the key binding form. In PVS all commands and abbreviations are preceded by a `M-x`; everything else is a key-binding. In later sections we will refer to commands by their long form name, without the `M-x` prefix. Some of the commands prompt for a theory or PVS file name and specify a default; if the default is the desired theory or file, you can simply type the `Return` key.

To begin, type `M-x pvs-help` (`C-h p`) for an overview of the commands available in PVS (type `q` to exit the help buffer). To exit PVS, use `M-x exit-pvs` (`C-x C-c`).

[8] You may need to include a pathname, depending on where and how PVS is installed.

[9] All the GNU EMACS (and X-Windows or Emacstool) command line flags can be added to the `pvs` command and passed through as appropriate; the `-q` flag inhibits loading of the user's `.emacs` initialization file, and should be used if difficulties are encountered starting PVS or if there appear to be conflicts in keybindings. Do *not* report errors to us unless they can be reproduced when the `-q` flag is used.

[10] Many keyboards provide a `Meta` key (hence the `M-` prefix), and this may be used instead. On the SUN3, the `Meta` key is normally labeled `Left` and on the SUN4 (SPARC), it is labeled ◇. The `Meta` key is like the shift key; to use it simply hold the `Meta` key down while typing another key.

PVS specifications consist of a number of files, each of which contains one or more theories. Theories may import other theories; imported theories must either be part of the prelude (the standard collection of theories built-in to PVS), or the files containing them must be in the same directory.[11] Specification files in PVS all have a `.pvs` extension. As specifications are developed, their proofs are kept in files of the same name with `.prf` extensions. The specification and proof files in a given directory constitute a PVS *context*; PVS maintains the state of a specification between sessions by means of the `.pvscontext` file. The `.pvscontext` and `.prf` files are not meant to be modified by the user. Other files used or created by the system will be described as needed. You may move to a different context (*i.e.*, directory) using the `M-x change-context` command, which is analogous to the UNIX `cd` command.

Now let's develop a small specification:

```
sum: THEORY
 BEGIN
  n: VAR nat
  sum(n): RECURSIVE nat =
   (IF n = 0 THEN 0 ELSE n + sum(n - 1) ENDIF)
   MEASURE (LAMBDA n: n)
  closed_form: THEOREM sum(n) = (n * (n + 1))/2
 END sum
```

This is a specification for summation of the first n natural numbers

This simple theory has no parameters and contains three declarations. The first declares `n` to be a variable of type `nat`, the built-in type of natural numbers. The next declaration is a recursive definition of the function `sum(n)`, whose value is the sum of the first `n` natural numbers. Associated with this definition is a *measure* function, following the `MEASURE` keyword, which will be explained below. The final declaration is a formula which gives the closed form of the sum.

2.1 Creating the Specification

The `sum` theory may be introduced to the system in a number of ways, all of which create a file with a `.pvs` extension,[12] which can be done by

1. using the `M-x new-pvs-file` command (`M-x nf`) to create a new PVS file, and typing `sum` when prompted. Then type in the `sum` specification.

2. Since the file is included on the distribution tape in the `Examples/tutorial` subdirectory of the main PVS directory, it can be imported with the `M-x import-pvs-file` command (`M-x imf`). Use the `M-x whereis-pvs` command to find the path of the main PVS directory.

3. Finally, any external means of introducing a file with extension `.pvs` into the current directory will make it available to the system; for example, using `vi` to type it in, or `cp` to copy it from the `Examples/tutorial` subdirectory.

[11] PVS does support soft links, thus supporting a limited capability for reusing theories.

[12] The file does not have to be named `sum.pvs`, it simply needs the `.pvs` extension.

The first two alternatives display the specification in a buffer. The third option requires an explicit request such as a built-in GNU EMACS file command (like `M-x find-file`, `C-x C-f`), or the `M-x find-pvs-file` (`M-x ff` or `C-c C-f`) command. The latter is more useful when there are multiple specification files, as it supports completion on just the specification files, ignoring other files that you or the system have created in the directory.

2.2 Parsing

Once the `sum` specification is displayed, it can be parsed with the `M-x parse` (`M-x pa`) command, which creates the internal abstract representation for the theory described by the specification. If the system finds an error during parsing, an error window will pop up with an error message, and the cursor will be placed in the vicinity of the error. If you didn't get an error, introduce one (say by misspelling the `VAR` keyword), then move the cursor somewhere else and parse the file again (note that the buffer is automatically saved). Fix the error and parse once more. In practice, the parse command is rarely used, as the system automatically parses the specification when it needs to.

2.3 Typechecking

The next step is to typecheck the file by typing `M-x typecheck` (`M-x tc`, `C-c t`), which checks for semantic errors, such as undeclared names and ambiguous types. Typechecking may build new files or internal structures such as TCCs. When `sum` has been typechecked, a message is displayed in the minibuffer indicating that three TCCs were generated. These TCCs represent *proof obligations* that must be discharged before the `sum` theory can be considered typechecked. The proofs of the TCCs may be postponed indefinitely, though it is a good idea to view them to see if they are provable. TCCs can be viewed using the `M-x show-tccs` command, the results of which are shown in Figure 1 below.

```
% Subtype TCC generated for n - 1
  % unproved
sum_TCC1: OBLIGATION (FORALL n: NOT n = 0 IMPLIES n - 1 >= 0)

% Termination TCC generated for sum
  % unproved
sum_TCC2: OBLIGATION (FORALL n: NOT n = 0 IMPLIES n - 1 < n)

% Subtype TCC generated for 2
  % unproved
closed_form_TCC1: OBLIGATION 2 /= 0
```

Figure 1: TCCs for Theory `sum`

The first TCC is due to the fact that `sum` takes an argument of type `nat`, but the type of the argument in the recursive call to `sum` is integer, since `nat` is not closed under subtraction. Note that the TCC includes the condition `NOT n = 0`, which holds in the branch of the `IF-THEN-ELSE` in which the expression `n - 1` occurs.

The second TCC is needed to ensure that the function `sum` is total, *i.e.*, terminates. PVS does not directly support partial functions, although its powerful subtyping mechanism allows PVS to express many operations that are traditionally regarded as partial. The measure function is used to show that recursive definitions are total by requiring the measure to decrease with each recursive call.

The final TCC is due to the fact that division in PVS is declared to be a function of type `[number, nonzero_number -> number]`, so the denominator must be nonzero.

These TCCs are trivial, and in fact can be discharged automatically by using the `M-x typecheck-prove` (`M-x tcp`) command, which attempts to prove all TCCs that have been generated. (Try it).

2.4 Proving

We are now ready to try to prove the main theorem. Place the cursor on the line containing the `closed_form` theorem, and type `M-x prove` (`M-x pr` or `C-c p`). A new buffer will pop up, the formula will be displayed, and the cursor will appear at the `Rule?` prompt, indicating that the user can interact with the prover. The commands needed to prove this theorem constitute only a very small subset of the commands available to the prover; more details can be found in the prover guide [SOR93].

First, notice the display (reproduced below), which consists of a single formula (labeled `{1}`) under a dashed line. This is a *sequent*; formulas above the dashed lines are called *antecedents* and those below are called *succedents*. The interpretation of a sequent is that the conjunction of the antecedents implies the disjunction of the succedents. Either or both of the antecedents and succedents may be empty.[13] In our case, we are trying to prove a single succedent.

The basic objective of the proof is to generate a *proof tree* in which all of the leaves are trivially true. The nodes of the proof tree are sequents, and while in the prover you will always be looking at an unproved leaf of the tree. The *current* branch of a proof is the branch leading back to the root from the current sequent. When a given branch is complete (*i.e.*, ends in a true leaf), the prover automatically moves on to the next unproved branch, or, if there are no more unproven branches, notifies you that the proof is complete.

Now back to the proof. We will prove this formula by induction on `n`. To do this, type `(induct "n")`.[14] This is not an EMACS command, rather it is typed directly at the prompt, including the parentheses. This generates two subgoals; the one displayed is the base case, where `n` is `0`. To see the inductive step, type `(postpone)`, which postpones the current subgoal and moves on to the next unproved one. Type `(postpone)` a second time to cycle back to the original subgoal (labeled `closed_form.1`).[15]

[13] An empty antecedent is equivalent to `true`, and an empty succedent is equivalent to `false`, so if both are empty the sequent is unprovable.

[14] PVS expressions are case-sensitive, and must be put in double quotes when they appear as arguments in prover commands.

[15] Three extremely useful EMACS key sequences to know here are `M-p`, `M-n`, and `M-s`. `M-p` gets the last input typed to the prover; further uses of `M-p` cycle back in the input history. `M-n` works in the opposite direction. To use `M-s`, type the beginning of a command that was previously input, and type `M-s`. This will get the previous input that matches the partial input; further uses of `M-s` will find earlier matches. Try these key sequences out; they are easier to use than to explain.

To prove the base case, we need to expand the definition of `sum`, which is done by typing `(expand "sum")`. After expanding the definition of `sum`, the prover applies the underlying decision procedures to simplify the succedent to `TRUE`, which completes the proof of this subgoal, and the system moves on to the next subgoal, which is the inductive step.

The first thing to do here is to eliminate the `FORALL` quantifier. This can most easily be done with the `skolem!` command[16], which provides new constants for the bound variables. To invoke this command type `(skolem!)` at the prompt. The resulting formula may be simplified by typing `(flatten)`, which will break up the succedent into a new antecedent and succedent. The obvious thing to do now is to expand the definition of `sum` in the succedent. This again is done with the `expand` command, but this time we want to control where it is expanded, as expanding it in the antecedent will not help. So we type `(expand "sum" +)`, indicating that we want to expand `sum` in the succedent.[17]

The final step is to send the proof to the PVS decision procedures, which can automatically decide certain fragments of arithmetic. This is done by typing `(assert)`.[18] The proof is now complete, the system may ask whether to save the new proof, and whether to display a brief printout of the proof. You should answer `yes` to these questions just to see how they work. After responding to these questions, the buffer from which the `prove` command was issued is redisplayed if necessary, and the cursor is placed on the formula that was just proved. The entire proof transcript is shown below. Yours may be different, depending on your window size and the timings involved.

```
closed_form :

  |-------
{1}   (FORALL (n: nat): sum(n) = (n * (n + 1)) / 2)

Rule? (induct "n")
Inducting on n,
this yields  2 subgoals:
closed_form.1 :

  |-------
{1}   sum(0) = (0 * (0 + 1)) / 2

Rule? (postpone)
Which takes us to,
closed_form.2 :
```

[16]The exclamation point differentiates this command from the `skolem` command, where the new constants have to be provided by the user.

[17]We could also have specified the exact formula number (here 1), but including formula numbers in a proof tends to make it less robust in the face of changes. There is more discussion of this in the prover guide [SOR93].

[18]The `assert` command actually does a lot more than decide arithmetical formulas, performing three basic tasks:

- it tries to prove the subgoal using the decision procedures.
- it stores the subgoal information in an underlying database, allowing automatic use to be made of it later.
- it simplifies the subgoal, again utilizing the underlying decision procedures.

These arithmetic and equality procedures are the main workhorses to most PVS proofs. You should learn to use them effectively in a proof.

```
  |-------
{1}   (FORALL (j: nat):
         sum(j) = (j * (j + 1)) / 2
           IMPLIES sum(j + 1) = ((j + 1) * (j + 1 + 1)) / 2)

Rule? (postpone)
Which takes us to,
closed_form.1 :

  |-------
{1}   sum(0) = (0 * (0 + 1)) / 2

Rule? (expand "sum")
Expanding the definition of sum
this simplifies to:
closed_form.1 :

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

which is trivially true.

This completes the proof of closed_form.1.

closed_form.2 :

  |-------
{1}   (FORALL (j: nat):
         sum(j) = (j * (j + 1)) / 2
           IMPLIES sum(j + 1) = ((j + 1) * (j + 1 + 1)) / 2)

Rule? (skolem!)
For the top quantifier in 1, we introduce Skolem constants: (j!1)
this simplifies to:
closed_form.2 :

  |-------
{1}   sum(j!1) = (j!1 * (j!1 + 1)) / 2
        IMPLIES sum(j!1 + 1) = ((j!1 + 1) * (j!1 + 1 + 1)) / 2

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
closed_form.2 :

{-1}   sum(j!1) = (j!1 * (j!1 + 1)) / 2
  |-------
{1}   sum(j!1 + 1) = ((j!1 + 1) * (j!1 + 1 + 1)) / 2

Rule? (expand "sum" +)
Expanding the definition of sum
this simplifies to:
closed_form.2 :

[-1]   sum(j!1) = (j!1 * (j!1 + 1)) / 2
  |-------
{1}   (IF j!1 + 1 = 0 THEN 0 ELSE j!1 + 1 + sum(j!1) ENDIF)
        = (j!1 * j!1 + 2 * j!1 + (j!1 + 2)) / 2

Rule? (assert)
Invoking decision procedures,
this simplifies to:
closed_form.2 :
```

```
{-1}   sum(j!1) = (j!1 * j!1 + j!1) / 2
  |-------
{1}   TRUE

which is trivially true.

This completes the proof of closed_form.2.

Q.E.D.


Run time  = 5.62 secs.
Real time = 58.95 secs.
```

2.5 Status

Now type `M-x status-proof-theory` (`M-x spt`) and you will see a buffer which displays the formulas in `sum` (including the TCCs), along with an indication of their proof status. This command is useful to see which formulas and TCCs still require proofs. Another useful command is `M-x status-proofchain` (`M-x spc`), which analyzes a given proof to determine its dependencies. To use this, go to the `sum.pvs` buffer, place the cursor on the `closed_form` theorem, and enter the command. A buffer will pop up indicating whether the proof is complete, and that it depends on the TCCs and the `nat_induction` axiom.

2.6 Generating LaTeX

In order to try out this section, you must have access to LaTeX and a TeX previewer, such as `vitex` or `dvitool` (for SunView), or `xdvi` (for X-windows). Otherwise this section may be skipped.

Type `M-x latex-theory-view` (`M-x ltv`). You will be prompted for the theory name—type `sum`, or just `Return` if `sum` is the default. You will then be prompted for the TeX previewer name. Either the previewer must be in your path, or the entire pathname must be given. This information will only be prompted for once per session, after that PVS assumes that you want to use the same previewer.

sum: THEORY
  BEGIN
  n: VAR nat
  sum(n): RECURSIVE nat $=$
    (IF $n = 0$ THEN 0 ELSE $n + \mathrm{sum}(n - 1)$ ENDIF)
      MEASURE $(\lambda\ n\colon n)$
  closed_form: THEOREM $\mathrm{sum}(n) = (n \times (n + 1)) / 2$
  END sum

Figure 2: Theory `sum`

After a few moments the previewer will pop up displaying the `sum` theory, as shown in Figure 2. Note that `LAMBDA` has been translated as λ. This and other translations are built into PVS; the user may also specify translations for keywords and identifiers (and override those built-in) by providing a substitution file, `pvs-tex.sub`, which contains commands to customize the LaTeX output. For example, if the substitution file contains the three lines

```
THEORY key 7 {\large\bf Theory}
sum    id  2 {\sum_{i = 0}^{\star_1} i}
sum    1   2 {\sum_{i = 0}^{#1} i}
```

the output will look like Figure 3.

sum: **Theory**
BEGIN
n: VAR nat

$(\sum_{i=0}^{\star_1} i)(n)$: RECURSIVE nat $=$ (IF $n = 0$ THEN 0 ELSE $n + \sum_{i=0}^{n-1} i$ ENDIF)
MEASURE $(\lambda\ n:\ n)$

closed_form: THEOREM $\sum_{i=0}^{n} i = (n \times (n + 1)) / 2$
END sum

Figure 3: Theory `sum`

Finally, using the `M-x latex-proof` command, it is possible to generate a LaTeX file from a proof. A part of an example is shown below; details are in the PVS system manual.

Expanding the definition of sum
`closed_form.2:`

$\{-1\}\quad \sum_{i=0}^{j'} i = (j' \times (j' + 1)) / 2$

$\{1\}\quad$ (IF $j' + 1 = 0$ THEN 0 ELSE $j' + 1 + \sum_{i=0}^{j' + 1 - 1} i$ ENDIF) $= ((j' + 1) \times (j' + 1 + 1)) / 2$

3 The PVS Language

The specification language of PVS is a highly expressive language based on higher-order logic. The language was designed to describe computer systems, but concentrates on abstract descriptions rather than detailed prescriptions (*i.e.*, *what* rather than *how*). The language supports modularity and reuse by means of parameterized *theories*, and has a rich type system, including the notion of a *predicate subtype*. This makes typechecking undecidable, but provides a great deal of flexibility. In addition, there are type constructors for function, tuple, record, and abstract datatypes.

A theory consists of a sequence of *declarations*, which provide names for types, constants, (logical) variables, axioms, and formulas. These names may be overloaded; *e.g.*, `+` may be declared to operate on a newly declared type, and still be available for integer addition. There is a large body of theories built into PVS, collectively referred to as the *prelude*.

In the following sections we will describe the language by means of a series of examples. These examples were chosen to exemplify various aspects of the language, and do not necessarily reflect the best style. The PVS language is described in detail in [OSR93a].

3.1 A Simple Example: The Rational Numbers

The rational numbers are built into PVS, but for the sake of illustration we attempt to develop a partial axiomatization. The examples in this section illustrate some simple syntactic and semantic aspects of PVS. They show how theories are defined containing declarations of types, variables, and constants. They also illustrate the definition of types, subtypes, and constants, the declaration of axioms and formulas, and the consequences of typechecking in the presence of subtypes. We start with the following theory introducing a type `rat`, a constant `zero` of type `rat`, and a binary function `/`. These form the signature of the theory `rats`.

```
rats: THEORY
 BEGIN
  rat : TYPE
  zero : rat
  / : [rat, rat -> rat]
 END rats
```

The type `rat` is uninterpreted; the only assumption made by the system is that it is nonempty. Here the division function '`/`' takes two arguments of type `rat` and returns a value of type `rat`.[19]

The theory presented so far says little about the rational numbers; just that there is a constant and a binary function defined on the type. The rationals are a model for this theory, but so are the booleans, the integers, etc.[20] The next thing to do is to introduce axioms and definitions that further constrain the possible interpretations of the theory. We can augment the `rats` theory above as follows:

```
rats : THEORY
 BEGIN
  rat : TYPE
  zero : rat
  / : [rat, rat -> rat]
  * : [rat, rat -> rat]
  x, y: VAR rat
  left_cancellation : AXIOM x * (y/x)  = y
  zero_times : AXIOM zero * x = zero
 END rats
```

In this augmented theory, we have introduced the declaration for the multiplication operation '`*`,' the identifiers `x` and `y` have been declared as logical variables that range over the type `rat`, and we have added two axioms asserting properties of multiplication and division.

[19] The division symbol '`/`' has already been declared as an infix symbol in the PVS grammar.

[20] For example, we can interpret `rat` as the boolean type, `zero` as FALSE, and `/` as `AND`.

Though the `left_cancellation` axiom looks plausible, there is a problem with it. A reasonable "challenge" for our theory is the conjecture `(EXISTS y : y /= zero)`.[21] Unfortunately, we can easily prove `zero = y` for any `y`, by substituting `zero` for `x` in `left_cancellation`, and applying `zero_times` with `(y/x)` substituted for `x`. The conclusion `zero = y` is clearly not intended for a model of the rational numbers. One way to repair the problem is to qualify the `left_cancellation` axiom so that it reads `x /= zero IMPLIES x * (y/x) = y`. In this way, the axioms make no restrictions on the value returned by division when given a zero denominator. This technique of having division by zero return some unspecified value is the traditional approach used in logic and mathematics, but can lead to problems in specifications since most implementations prefer to treat this as an error condition, rather than returning an arbitrary value.

To circumvent this problem, PVS makes it possible to specify division so that it is an error to apply it when the denominator is zero. Here is an improved PVS specification:

```
rats : THEORY
 BEGIN
  rat : TYPE
  x, y : VAR rat
  zero : rat
  nonzero : TYPE = {x | x /= zero}
  / : [rat, nonzero -> rat]
  * : [rat, rat -> rat]
  left_cancellation : AXIOM zero /= x IMPLIES x * (y/x)  = y
  zero_times : AXIOM zero * x = zero
 END rats
```

Here the type `nonzero` is defined to be the type of elements of `rat` that are different from `zero`. We call `nonzero` a *predicate subtype* of `rat`, since it consists of the elements of `rat` satisfying a given predicate. In this new specification, the denominator argument may only range over the nonzero elements of the type `rat`. In typechecking the occurrence of division in `left_cancellation`, there is a *type correctness condition* (TCC) generated by the typechecker that is added to the theory as a declaration:

```
left_cancellation_TCC1: OBLIGATION
      (FORALL (x: rat): zero /= x IMPLIES x /= zero)
```

Notice that the logical antecedent governing the occurrence of division is included as an antecedent of `left_cancellation_TCC1`. This TCC is of course easily proved. In fact, if the antecedent were written in the more natural form `x /= zero`, the TCC would not have been generated. TCCs such as this are *obligations*, and they must be proved in order to show that the theory is type correct. PVS allows the proof of a TCC to be deferred, but until it has been discharged any proofs involving the theory `rats` directly or indirectly will be considered incomplete.

For a more slightly more sophisticated example, we can introduce the "less-than" relation and the subtraction operation, along with a statement asserting a property of subtraction, division, and the less-than ordering.

```
< : [rat, rat -> bool]
- : [rat, rat -> rat]
div_test: FORMULA x /= y IMPLIES (y-x)/(x-y) < zero
```

[21] In PVS, "not equal" is written as "`/=`".

Notice here that the relation `<` is declared as a function with range type `bool` representing the booleans[22] Typechecking the formula `div_test` generates the TCC

```
div_test_TCC1: OBLIGATION
    (FORALL (x, y: rat): x /= y IMPLIES (x - y) /= zero)
```

An alternative notation for predicate subtypes is illustrated by another example. If the predicate `non_neg?` (we often use a question mark in a name to indicate predicates) is defined as

```
non_neg?(x): bool = NOT (x < zero)
```

then the expression `(non_neg?)` denotes a subtype with the same meaning as the type expression {`x | NOT (x < zero)`}. The function returning the absolute value of a rational can now be specified as

```
abs(x): (non_neg?) = (IF (x < zero) THEN -x ELSE x ENDIF)
```

When typechecked, this definition generates the TCCs

```
abs_TCC1: FORMULA
  (FORALL (x: rat): (x < zero) IMPLIES non_neg?(-x))
abs_TCC2: FORMULA
  (FORALL (x: rat): NOT (x < zero) IMPLIES non_neg?(x))
```

Note that the type of `abs` is `[rat -> (non_neg?)]` which is more informative than the type of `[rat -> rat]`. The advantage of this is that whenever `abs` occurs as an argument to a function requiring a `non_neg?` argument, no new obligations are generated. For example, a square root function `sqrt` may be defined with type `[(non_neg?) -> (non_neg?)]` and freely applied to the result of `abs` without incurring any new obligations.[23]

Clearly, we still have an inadequate specification of rational numbers since we do not have the axioms required to prove the various TCCs that were generated. We will not embark on the full axiomatization here, as no new features of the language are involved. The development of the rational numbers is described in an appendix to the Language Reference [OSR93a]. It is important to note that the PVS proof checker has an underlying decision procedure that automatically proves many of the properties of the rational numbers.

Summarizing, we have illustrated how predicate subtypes can be used to circumscribe the domain of partially defined operations such as division, and to more usefully delineate the range of functions such as `abs`. We also examined how the use of predicate subtypes in an expression could require certain type correctness conditions to be proved before the expression is regarded as well-typed.

3.2 A More Sophisticated Example: Stacks

Perhaps the most hackneyed specification example is that of stacks. It is interesting therefore to examine if PVS can contribute anything novel to this well-worn example. For starters, let us try to capture the signature of the stack operations.

[22]Functions with range type `bool` are often called *predicates* in PVS, and are also used to represents sets. Some convenient notation for these interpretations is introduced later.

[23]This is one of the advantages of having predicate subtypes; in a logic of partial terms we would be forced to show that the term involving `sqrt` is well-defined every time it occurs, or to somehow cache the information.

```
stacks : THEORY
 BEGIN
  stack : TYPE
  empty : stack
  push : [nat, stack -> stack]
  top : [stack -> nat]
  pop : [stack -> stack]
 END stacks
```

In the theory `stacks` above, `nat` is the built-in type for natural numbers, `stack` is declared as an uninterpreted type, and the rest are uninterpreted constants of type `stack`. The function `push` is declared to take a natural number and a stack and return a stack. The `top` function takes a stack and returns a natural number. The `pop` function takes a stack and returns a stack.

One immediate objection is that the above declaration only specifies the signature for stacks of natural numbers and is therefore not sufficiently generic. PVS supports this by providing parameterization at the theory level.[24] With the `stacks` theory appropriately parameterized, we get

```
stacks [t : TYPE] : THEORY
 BEGIN
  stack : TYPE
  empty : stack
  push : [t, stack -> stack]
  top : [stack -> t]
  pop : [stack -> stack]
 END stacks
```

This declares a *schema* of stacks, one for each type. Within the `stacks` theory `t` is treated as a fixed uninterpreted type. When the `stacks` theory is used by another theory it must be instantiated. For example, the theory of stacks of natural number is just `stacks[nat]`. It is important to note that each instantiation of a theory yields a new signature; thus instantiating with `int` and `nat` yields two different `empty` constants.

The new signature is still unsatisfactory since the signatures for `pop` and `top` permit expressions such as `pop(empty)` and `top(empty)`, for which it is difficult to ascribe a meaning. The obvious solution in PVS is to mimic what was done with division and constrain the domains of `pop` and `top`. We do this by introducing the predicate `nonemptystack?` in the new specification below. Note that the expression `[stack -> bool]` for the type of a predicate has been rewritten as the equivalent `PRED[stack]`. In addition, we add the usual stack axioms:

```
stacks [t: TYPE] : THEORY
 BEGIN
  stack : TYPE
  nonemptystack? : PRED[stack]
  s : VAR stack
  x, y : VAR t
  ns : VAR (nonemptystack?)
  empty : stack
```

[24] An alternative approach would allow type variables and type abstraction in the language, but in the presence of subtypes this greatly complicates the semantics.

```
   push : [t, stack -> (nonemptystack?)]
   top : [(nonemptystack?) -> t]
   pop : [(nonemptystack?) -> stack]
   pop_push : AXIOM pop(push(x, s)) = s
   top_push : AXIOM top(push(x, s)) = x
   push_pop_top : AXIOM  push(top(ns), pop(ns)) = ns
   push_empty : AXIOM empty /= push(x, s)
   nonempty_empty : THEOREM NOT nonemptystack?(empty)
   pop2_push2 : THEOREM pop(pop(push(x, push(y, s)))) = s
  END stacks
```

Now we can explore the consequence of this specification by examining the formula declaration `pop2_push2`. The left-hand side expression of this formula poses a problem for most type systems since the type of `pop(push(x, (push(y, s))))` (i.e., the argument to the outermost `pop`) is `stack`, whereas the domain type of `pop` is the more constrained `(nonemptystack?)`. In PVS, the typechecker generates the TCC

```
pop2_push2_TCC1: OBLIGATION
  (FORALL (x, y: t), (s: stack):
     nonemptystack?(pop(push(x, push(y, s)))))
```

This TCC is easily proved from the axiom `pop_push`.

We have now presented an abstract specification of stacks, showing how theories may be parameterized and further illustrating the subtype mechanism. In the following section we will explore different constructive definitions of stacks, and in Section 3.9 we will see how to define stacks as an abstract datatype.

3.3 Implementing Stacks

Having satisfied ourselves that stacks can be specified using the signature and axioms above, we might wish to introduce an alternative, more definitional specification of stacks. In this new specification, we implement a stack with two components: a counter for recording the number of elements in the stack, and an array containing the stack entries. One way to implement such a stack in PVS is to use the tuple type constructor and another is to employ records. We examine both approaches below.

```
newstacks [t: TYPE] : THEORY
 BEGIN
  i, j: VAR nat
  stack : TYPE = [ nat, ARRAY[nat -> t] ]
  s: VAR stack
  x, y: VAR t
  size(s): nat = proj_1(s)
  elements(s): ARRAY[nat -> t] = proj_2(s)
  e: t
  nonemptystack?(s) : bool = (size(s) > 0)
  empty: stack = (0, (LAMBDA j: e))
  push(x,s): (nonemptystack?) =
    (size(s)+1, elements(s) WITH [(size(s)):=x])
  ns: VAR (nonemptystack?)
  top(ns): t = elements(ns)(size(ns)-1)
```

```
  pop(ns): stack = (size(ns)-1, elements(ns))
 END newstacks
```

There are several points to note about the above specification. The 2-tuple implementing stacks is specified by the type `[nat, ARRAY [nat -> t]]` whose first component is a natural number and whose second component is an array with index type `nat` and element type `t`. The array type expression `ARRAY[nat -> t]` is identical to the function type expression `[nat -> t]` and the `ARRAY` keyword serves a descriptive rather than semantic purpose. The built-in family of `proj` functions serve to access tuple components. The function `size` is defined to return the size of stack `s` which is defined to be the first component of `s`, namely, `proj_1(s)`. The stack elements are "stored" in the array that is the second component of stack `s`, so that `elements(s)` is defined as `proj_2(s)`. The `size` function could also have been defined by the declaration

```
size: [stack -> nat] = proj_1
```

which is merely a consequence of applying the extensionality axiom of higher-order logic to the earlier definition of `size`. The next thing to note is that if the size of the stack is `i`, for `i > 0`, then the `i` stack values are stored in the indices `0` to `i-1`. A stack expression is constructed by means of a pair of comma-separated expressions enclosed in parentheses, so that the empty stack is constructed by the expression `(0, (LAMBDA j: e))` whose size component is `0`, and where the `elements` array is initialized to contain a default element `e`. The `push` operation applied to an element `x` and a stack `s` constructs a stack with size `size(s) + 1`, where the `size(s)` index of the `elements` array of `s` has been updated to be `x`. The *function update* is done by the `WITH` construct as used in `elements(s) WITH [(size(s)):=x]`. Correspondingly, the `pop` operation decrements the stack size by one, but leaves the `elements` array unchanged. Note that we have used predicate subtyping to ensure that `pop` is only applied to stacks of positive size, *i.e.*, nonempty stacks. The `top` operation applied to a nonempty stack `ns` returns the `(size(ns)-1)`th element of the array `elements(ns)`.

The above specification of stacks when implemented using the record type construction of PVS rather than tuples, has the following form.

```
newstacks [t: TYPE] : THEORY
 BEGIN
  i, j: VAR nat
  stack : TYPE = [# size: nat, elements : ARRAY[nat -> t] #]
  s: VAR stack
  x,y: VAR t
  e: t
  nonemptystack?(s) : bool = (size(s) > 0)
  empty : stack = (# size := 0, elements := (LAMBDA j: e) #)
  push(x,s): (nonemptystack?) =
     (# size:= size(s)+1,
        elements := elements(s) WITH [(size(s)):=x] #)
  ns: VAR (nonemptystack?)
  top(ns): t = (elements(ns)(size(ns)-1))
  pop(ns): stack = (ns WITH [size := size(ns) -1])
 END newstacks
```

In this new specification of stacks, the record type is constructed by the type expression `[# size: nat, elements : ARRAY[nat -> t] #]` with fields `size` and `elements`. The expression

`(# size := 0, elements := (LAMBDA j: e) #)` constructs the empty stack. This specification is slightly more pleasing than the one using tuples since the `size` and `elements` functions were automatically generated from the record labels, and record updating can be used to concisely define the `pop` operation.

There is a problem with both versions of `newstacks` that has to do with stack equality. If we consider `newstacks[nat]`, *i.e.*, stacks of natural numbers, then both `(# size := 0, elements := (LAMBDA i: 1) #)` and `(# size := 0, elements := (LAMBDA i: 2) #)` represent empty stacks, but they are unequal since they differ on their elements field. Similarly, with this representation, the formula `pop(push(x, s)) = s` fails to be a theorem, since the `elements` array may differ at elements beyond `size`. We defer the discussion of this problem and its solution to Section 3.8.

To summarize the discussion of the PVS language so far, we have examined theories, declarations and definitions of types and constants, declarations of axioms and formulas, predicate subtypes and the generation of type correctness conditions, and the definition, construction and use of tuple and record types. So far we have only made limited use of higher-order logic by using a function object to model the array that is used to contain the elements of a stack. We next examine the use of theories in PVS.

3.4 Using Theories: Partial and Total Orders

There are several reasons for structuring specifications into (parameterized) theories as is done in PVS. The primary ones are that it provides for modularization, and the use of parameters allows more generic specifications, as we saw with stacks. In this section, we focus on the use and parameterization of theories.

A preliminary example of theory is that of partial orders as transcribed in PVS below.[25]

```
partial_order [t: TYPE] : THEORY
 BEGIN
  <= : PRED[[t,t]]
  x, y, z: VAR t
  refl: AXIOM x <= x
  trans: AXIOM  x <= y AND y <= z IMPLIES x <= z
  antisym: AXIOM x <= y AND y <= x IMPLIES x = y
  < : PRED[[t,t]] = (LAMBDA x, y: x <= y AND x /= y)
 END partial_order
```

Note that the type of a binary relation, such as `<=`, can be given either as `[t, t -> bool]`, or as a predicate on the tuple `[t, t]`, as illustrated above. For any type `t`, the theory `partial_order` introduces a partial order relation ‘`<=`’ with the axioms of reflexivity, transitivity, and antisymmetry. It also introduces a strict partial order relation ‘`<`’ along with its definition.

The next example is the theory of total ordering which extends the original theory `partial_order`.

[25] This is built in to PVS in a different form; again, the development here is for pedagogical purposes.

```
total_order [t: TYPE] : THEORY
 EXPORTING ALL WITH partial_order[t]
 BEGIN
  IMPORTING partial_order[t]
  x, y: VAR t
  total: AXIOM  x <= y OR y <= x
 END total_order
```

There are several points to note with `total_order`. It *imports* the theory `partial_order[t]` with the `IMPORTING` construct. It would have also been acceptable to import the generic theory `partial_order` since the typechecker is able to resolve the type of the occurrences of `<=` in `total_order` as belonging to `partial_order[t]`. The `EXPORTING` clause that precedes the body of the theory (as marked by `BEGIN`) causes every type, constant, and formula declaration in `total_order` to be visible in any theory that imports `total_order`. In addition to the declarations in `total_order`, the `EXPORTING` clause also makes visible those declarations in `partial_order[t]` that are externally visible. When there is no `EXPORTING` clause, the default is that all declarations[26] are visible, including all instances of imported theories that were referenced.[27] Generic theories cannot be exported, *i.e.*, it is not possible to replace `partial_order[t]` in the `EXPORTING` clause with `partial_order`.

3.5 Using Theories: Sort

The next series of examples illustrate the use of the `partial_order` and `total_order` theories in several ways. These examples provide a generic specification of what it means for an array to be sorted.

```
sort [domain, range: TYPE] : THEORY
 BEGIN
  IMPORTING partial_order[range], total_order[domain]
  Array_type: TYPE = ARRAY[domain -> range]
  A, B, C: VAR Array_type
  sorted?(A): bool =
    (FORALL (x, y: domain): x < y IMPLIES NOT (A(y) <= A(x)))
 END sort
```

The above theory is parameterized with respect to the types `domain` and `range`. It imports the theories `partial_order[range]` and `total_order[domain]`. The predicate `sorted?` on arrays of element type `range` and index type `domain` is defined to check that the partial ordering on the elements never violates the total ordering on the indices. Note that the types of the predicates `<` and `<=` are potentially ambiguous since they could come from either `partial_order[range]` and `total_order[domain]` but the typechecker resolves their types from the context of their application. If it was not possible to resolve the ambiguity from the context, then it would have been necessary to write `<` as `total_order[domain].<` in order to distinguish it from `partial_order[domain].<`.

One immediate problem with the above specification of sortedness is that it is specified with respect to a fixed total ordering on the indices and a fixed partial order on the elements

[26] With the exception of variable declarations.

[27] When a generic theory is imported, the typechecker determines the instance for each reference to an entity declared in the generic theory—it is these instances that are exported.

of the array. It is therefore not sufficiently generic. The following revised specification of the theory `sort` fixes this problem. It does this by taking the domain and range orderings as parameters but then places restrictions that constrain the domain ordering to be total and the range ordering to be partial. These restrictions are listed in the `ASSUMING` part between the keywords `ASSUMING` and `ENDASSUMING`. The assuming part can only contain variable declarations and assumptions. These assumptions have to be discharged whenever the theory is instantiated with actual parameters. These proof obligations are automatically generated by the typechecker.

```
sorta [domain, range: type,
       d_order: PRED[domain, domain],
       r_order: PRED[range, range]] : THEORY
 BEGIN
  ASSUMING
   x, y, z: VAR domain
   u, v, w: VAR range
   d_refl: ASSUMPTION d_order(x,x)
   d_trans: ASSUMPTION  d_order(x, y) & d_order(y, z) => d_order(x, z)
   d_antisym: ASSUMPTION d_order(x, y) & d_order(y, x) => x = y
   d_total: ASSUMPTION  d_order(x, y) OR d_order(y, x)
   r_refl: ASSUMPTION r_order(u, v)
   r_trans: ASSUMPTION  r_order(u, v) & r_order(v, w) => r_order(u, w)
   r_antisym: ASSUMPTION r_order(u, v) & r_order(v, u) => u = v
  ENDASSUMING
  Array_type: TYPE = ARRAY[domain->range]
  A, B, C: VAR Array_type
  sorted?(A): bool =
    (FORALL (x, y: domain):
       (d_order(x, y) & x /= y) => NOT r_order(A(y), A(x)))
 END sorta
```

The above specification of sortedness might seem a little tedious given that we have already specified partial and total orderings. This points to difficulties in the original specifications for `partial_order` and `total_order`. In the first place, the constant '`<=`' is declared in the `partial_order` theory; in general there is already a relation at hand, and the desire is to check that it is a partial order. So we would like '`<=`' to be a parameter to the theory. But now the axioms are inappropriate; if the theory is parameterized with '`/=`', for example, then we have an inconsistency. There are two possible approaches to this. One is to only allow the theory to be parameterized with relations that satisfy the axioms. This is done by means of an `ASSUMING` part:

```
partial_order1 [t: TYPE, <=: PRED[[t,t]]] : THEORY
 BEGIN
  ASSUMING
   x, y, z: VAR t
   refl: ASSUMPTION x <= x
   trans: ASSUMPTION  x <= y AND y <= z IMPLIES x <= z
   antisym: ASSUMPTION x <= y AND y <= x IMPLIES x = y
  ENDASSUMING
  < : PRED[[t,t]] = (LAMBDA x, y: x <= y AND x /= y)
 END partial_order1
```

Now if the theory is instantiated with '`/=`', the typechecker will generate proof obligations which will be impossible to prove; thus such an instantiation is disallowed.

The alternative is to declare higher-order predicates instead of axioms:

```
partial_order2 [t: TYPE] : THEORY
 BEGIN
  <= : VAR PRED[[t,t]]
  x, y, z: VAR t
  reflexive?(<=): bool = (FORALL x: x <= x)
  transitive?(<=): bool =
    (FORALL x,y,z: x <= y AND y <= z IMPLIES x <= z
  antisymmetric?(<=): bool =
    (FORALL x,y: x <= y AND y <= x IMPLIES x = y
 END partial_order2
```

The advantage of this theory is that it allows us to test directly the properties of a given relation. The disadvantage is the the '<' relation must be defined outside. The real advantage comes in being able to combine properties, and use them as types. For example we can add the declaration

```
partial_order?(<=): bool =
  reflexive?(<=) AND transitive?(<=) AND antisymmetric?(<=)
```

and declare a relation `R` to be a variable ranging over partial orders on the type integer:

```
R: VAR (partial_order?[int])
```

We exploit this in the next theory specification, which describes the various ordering relations as predicates on relations. The `orderings` theory introduces an infix *variable* `<=` which is a reasonable thing to do in a higher-order logic. Now notice that the predicates `reflexive?`, `antisymmetric?`, `transitive?`, etc., are higher-order operations since they are predicates on predicates. The important concept of well-foundedness is also introduced in the theory below. A partial order `<=` is said to be well-founded on a set A if A contains no infinitely descending chain of elements $\ldots$`<=`x_n`<=`$\ldots$`<=`x_1`<=`x_0. Another way of stating this is that every nonempty subset of A must contain a minimal element with respect to `<=`. In terms of higher-order logic, for every predicate `qq` on `t` that holds somewhere (*i.e.*, is nonempty), there is a minimal element that satisfies `qq`.

```
orderings[t: TYPE] : theory
 BEGIN
  x, y, z: VAR t
  pp, qq: VAR PRED[t]
  <= : VAR PRED[t,t]
  reflexive?(<=) : bool =  (FORALL x: x <= x)
  antisymmetric?(<=) : bool =
       (FORALL x, y: x <= y AND y <= x IMPLIES x = y)
  transitive?(<=) : bool =
    (FORALL x, y, z: x <= y AND y <= z IMPLIES x <= z)
  partial_order?(<=) : bool =
    reflexive?(<=) AND antisymmetric?(<=) AND transitive?(<=)
  linear?(<=) : bool =
    (FORALL x, y : x <= y OR y <= x)
  total_order?(<=) : bool =
      partial_order?(<=) AND linear?(<=)
  well_founded?(<=): bool =
    (FORALL qq: (EXISTS y: qq(y))
```

```
        IMPLIES (EXISTS y: (FORALL x: qq(x) IMPLIES NOT x<=y)))
  END orderings
```

Now we can give yet another specification of sortedness. The interesting thing to note here is the occurrence of a `IMPORTING` clause in the parameter list of the theory to bring in the information that is needed to typecheck the remaining parameters.

```
sorto [domain, range: type,
       (IMPORTING orderings[range], orderings[domain])
       d_order: (total_order?),
       r_order: (partial_order?)] : THEORY
 BEGIN
  Array_type: TYPE = ARRAY[domain->range]
  A, B, C: VAR Array_type
  sorted?(A): bool =
    (FORALL (x, y: domain): (d_order(x, y) AND x/=y)
                    IMPLIES NOT r_order(A(y), A(x)))
 END sorto
```

Another thing one can do with the `orderings` theory is define well-founded induction which is a standard proof technique for proving properties of programs. The key idea is that if we are trying to prove `pp(x)` for all `x` of type `t`, and there is a well-founded ordering `<=` on `t`, then we can reason as follows. Consider the subset of elements `y` of `t` such that `NOT pp(y)` holds. Suppose that this set is nonempty, then by well-foundedness, this subset must contain a minimal element `z`, but the hypothesis of `wf_induction` can be used to derive `pp(z)` since `(FORALL y: y <= z AND y /= z IMPLIES pp(y))` holds trivially for a minimal element `z`. Thus both `pp(z)` and `NOT pp(z)` hold, contradicting the assumption that the set of `y` such that `NOT pp(y)` is nonempty.

```
wf_induction [t: type,
              (IMPORTING well_founded[t])
              <= : (well_founded?)] : THEORY
 BEGIN
  x, y, z: VAR t
  pp: VAR PRED[t]
  wf_induction: AXIOM
    (FORALL x: (FORALL y: y<=x AND y/= x IMPLIES pp(y))
               IMPLIES pp(x))
    IMPLIES (FORALL x: pp(x))
 END wf_induction
```

The `wf_induction` theory makes use of higher-order logic to assert the given ordering `<=` to be well-founded, and to state induction as an axiom for any predicate `pp`.

As we did in the case of ordering relations, we can build a theory defining various properties of functions in order to further illustrate the capabilities of higher-order logic as formalized by PVS.

```
functions [domain, range : type] : THEORY
 BEGIN
  fun : VAR [domain -> range]
  x, y : VAR domain
  injective?(fun) : bool =
     (FORALL x, y : fun(x) = fun(y) IMPLIES x=y)
  surjective?(fun) : bool =
```

```
      (FORALL (u : range) (EXISTS x : fun(x) = u))
    bijective?(fun) : bool = injective?(fun) AND surjective?(fun)
  END functions
```

If we then introduce a function as an injection, the PVS typechecker will require that we demonstrate that its definition indeed satisfies the predicate `injective?`. For example, `square: (injective?[int, nat]) = (LAMBDA x: x * x)` generates the (false) TCC `x*x = y*y IMPLIES x=y`, thereby revealing an error in this construction.

Summarizing, we have examined some more advanced capabilities of the language and logic of PVS. The parameterization and use of theories was illustrated in all the examples in this section. The theories `orderings`, `wf_induction`, and `functions` illustrate the higher-order aspects of the logic as well. We also noted the capability of the typechecker in resolving ambiguities in naming from the application context.

3.6 Sets in Higher-order Logic

In this section, we expand on the capabilities of higher-order logic with a naive encoding of the various set-theoretic operations. We stay consistent with the higher-order logic view of sets as predicates. In this case the theory `sets` is parameterized so that we are talking about predicates over a type `T`. The element `x` is a member of a set `a` if and only if `a(x)` is `TRUE`. The `union` and `add` operation is defined in terms of disjunction (`OR`), and the intersection and difference operations are defined in terms of conjunction (`AND`). The extensionality axiom asserts that if sets `a` and `b` have exactly the same members, then they are equal. Extensionality for sets can be proved from extensionality for functions so it is stated below as a lemma.

```
sets [T: TYPE] : THEORY
 BEGIN
  set: TYPE = SETOF[T]
  member(x:T,a:set): bool = a(x)
  union(a,b:set): set = {x:T | member(x,a) OR member(x,b)}
  intersection(a,b:set): set = {x:T | member(x,a) AND member(x,b)}
  difference(a,b:set) : set = {x:T | member(x,a) AND NOT member(x,b)}
  add(x:T,a:set) : set = {y:T | x = y OR member(y,a)}
  singleton(x:T) : set = {y:T | y = x}
  subset?(a,b:set) : bool = (FORALL (z:T) : member(z,a) => member(z,b))
  strict_subset?(a,b:set) : bool = subset?(a,b) AND a /= b
  empty?(a:set) : bool = (FORALL (x:T) : NOT member(x,a))
  emptyset: set = {x:T | FALSE}
  fullset: set = {x:T | TRUE}
  extensionality: LEMMA
    FORALL (a,b: set):
      (FORALL (x:T): member(x,a) = member(x,b)) => (a = b)
 END sets
```

Sequences provide yet another nice illustration of the power of the PVS higher-order logic. We can formalize infinite sequences of elements from some type `T` as functions of type `[nat -> T]`, where `nat` is the type of natural numbers. Then the `n`th element of a sequence `seq` is just `seq(n)`. The sequence that is obtained from `seq` by removing the first `n` elements is defined as `suffix(seq, n)`.

```
sequences[T: TYPE] : THEORY
 BEGIN
  sequence : TYPE = [nat->T]
  nth(seq: sequence, n: nat): T = seq(n)
  suffix(seq:sequence, n:nat): sequence =
    (LAMBDA (i:nat): seq(i+n))
  first(seq: sequence): T = nth(seq, 0)
  rest(seq: sequence): sequence = suffix(seq, 1)
 END sequences
```

Both `sets` and `sequences` are employed heavily in specification writing.

3.7 Recursion

In this section we discuss recursive declarations. We start with a simple example, the factorial function:

```
factorial(x:nat): RECURSIVE nat =
  IF x = 0 THEN 1 ELSE x * factorial(x - 1) ENDIF
  MEASURE (LAMBDA (x:nat): x)
```

This is similar to a constant declaration, except that the defining expression references `factorial`, which is the function being defined. In addition, there is a `MEASURE` function specified. In PVS, all definitions are total, and form a conservative extension.[28] In order to guarantee these conditions, a `MEASURE` function is required. This function has the same domain as the definition, but its range is `nat`.[29] The `MEASURE` function is used to show that the definition terminates, by generating an obligation that the `MEASURE` decreases with each call:

```
factorial_TCC2: OBLIGATION
  (FORALL (x: nat): NOT x = 0 IMPLIES x > x - 1)
```

Note that the context '`NOT x = 0`' is included in the *termination* TCC. PVS does not allow mutual recursion directly, although the same effect may be had by using axioms or by translating the mutually recursive forms to higher order, so this is not a real restriction.

3.8 Dependent Typing

In this section, we illustrate a more sophisticated form of typing that can involve dependencies between the components of a tuple or a record type, and also between the range and domain of a function type. As we have already seen, predicate subtyping makes it possible to express properties within the type language, and dependent typing significantly enhances this capability.

To explore dependent typing, we return to the example of `newstacks`. There the type `stack` was defined as the record type `[# size: nat, elements : ARRAY[nat -> t] #]`. We noted that this specification would distinguish between two empty stacks simply because they contained different `elements` arrays, even though the contents of the `elements` array are irrelevant when the the `size` field is `0`. What we would like is to specify a record with two fields, `size` and `elements`, where the type of the `elements` field varied according to the contents of the `size` field. We can in fact express such a record type in PVS so that the definition of the type `stack` becomes

[28] This means that (new) inconsistencies are not introduced as a result of adding a new definition.

[29] The range may be the constructive ordinals instead, but we will not be discussing that further here.

```
stack : TYPE =
    [# size: nat, elements : ARRAY[{i| i<size} -> t] #]
```

Note that the index type of the `elements` array has been restricted to the natural numbers below the contents of the `size` field. Such a record type is an instance of a *dependent type*. With this form of dependent typing, the `newstacks` specification can be written in PVS as follows.

```
newstacks [t: TYPE] : THEORY
 BEGIN
  i: VAR nat
  stack : TYPE = [# size: nat, elements: ARRAY[{i| i<size} -> t] #]
  s: VAR stack
  x,y: VAR t
  e: t
  nonemptystack?(s) : bool = (size(s) > 0)
  empty: stack =
    (# size := 0, elements := (LAMBDA (j: i|i<0): e) #)
  push(x,s): (nonemptystack?) =
    (# size:= size(s)+1,
       elements := elements(s) WITH [(size(s)):=x] #)
  ns: VAR (nonemptystack?)
  pop(x,ns): stack = (ns WITH [size := size(ns) -1])
  top(ns): t = (elements(ns)(size(ns)-1))
 END newstacks
```

There are a number of subtleties to the above specification. The `empty` stack contains an `elements` array with an empty index type. Now any two stacks with the `size` field set to zero will be equal since any element arrays will be treated as equal when compared over the empty index type.

The value returned by the `push` operation is a stack where the size field is one greater than that of the input stack. The type of the `elements` field of this stack is therefore different from that of the input stack. There is an additional index where the `elements` array must be defined, and the update operation (using the `WITH` construct) ensures that the `elements` field is in fact defined on this additional index.

The `pop` operation merely decrements the `size` component of the input stack by one to get the output stack and leaves the value of the `elements` field unchanged. Once again, this is acceptable since the elements field of the input stack is larger than the elements field required of the output stack. With this treatment, the formula `pop(push(x, s)) = s` is provable.

3.9 Abstract Datatypes: Stacks

In this section we describe one of the more powerful features of the PVS language: abstract datatypes. We will once again be using stacks for illustrative purposes.

The abstract `stacks` theory of Section 3.2 contains axioms providing the usual algebraic specification of stacks. However, PVS has a mechanism for automatically generating a complete axiomatization for such a theory from a very succinct description. Thus an alternative specification for stacks would be

```
stack [t: TYPE]: DATATYPE
 BEGIN
  empty: emptystack?
  push(top: t, pop: stack) : nonemptystack?
 END stack
```

Notice that the keyword `DATATYPE` distinguishes this from an ordinary `THEORY`. In this specification, `empty` and `push` are *constructors*, `top` and `pop` are *accessors*, and `emptystack?` and `nonemptystack?` are *recognizers* of the parameterized `stack` type. In addition to generating the signatures given in the previous `stacks` theory, this specification automatically generates (in the file and theory called `stack_adt`):

- Extensionality axioms for the constructors, *e.g.*,

```
push_extensionality: AXIOM
  (FORALL (nonemptystack?_var: (nonemptystack?)),
          (nonemptystack?_var2: (nonemptystack?)):
     top(nonemptystack?_var) = top(nonemptystack?_var2)
       AND pop(nonemptystack?_var) = pop(nonemptystack?_var2)
       IMPLIES nonemptystack?_var = nonemptystack?_var2)
```

- Accessor/constructor axioms, *e.g.*,

```
pop_push: AXIOM
  (FORALL (push1_var: t), (push2_var: stack):
    pop(push(push1_var, push2_var)) = push2_var)
```

- A disjointness axiom stating that no stack may be both empty and nonempty:

```
stack_disjoint: AXIOM
  (FORALL (stack_var: stack):
    NOT (emptystack?(stack_var) AND nonemptystack?(stack_var)))
```

- An inclusion axiom stating that every stack must be either empty or nonempty:

```
stack_inclusive: AXIOM
  (FORALL (stack_var: stack):
    emptystack?(stack_var) OR nonemptystack?(stack_var))
```

- An induction scheme:

```
stack_induction: AXIOM
  (FORALL (p: [stack -> boolean]):
     p(empty) AND
       (FORALL (push1_var: t), (push2_var: stack):
          p(push2_var) IMPLIES p(push(push1_var, push2_var)))
       IMPLIES (FORALL (stack_var: stack): p(stack_var)))
```

- A recursive combinator:

```
stack_nat_rec((emptystack?_fun: nat),
              (nonemptystack?_fun: [t, nat -> nat])):
 [stack -> nat] =
  LAMBDA (stack_var: stack):
   CASES stack_var OF
    empty: emptystack?_fun,
```

```
    push(push1_var, push2_var):
      nonemptystack?_fun(push1_var,
                         stack_nat_rec(emptystack?_fun,
                                       nonemptystack?_fun)
                           (push2_var))
  ENDCASES
```

The recursive combinator allows the specification of functions such as `length`[30]:

```
length(s:stack): nat =
  stack_nat_rec(0, (LAMBDA (x:t),(n:nat): n + 1))
```

- In addition to the recursive combinator used above (which is specialized to the construction of measure functions), a fully general recursive combinator is generated in a separate parameterized theory named `stack_rec_mod`.

3.10 Abstract Datatypes: Terms

More complicated examples of abstract datatypes may be given. For example, an abstract term structure may be defined as below

```
term [id, varid: TYPE] : DATATYPE
 BEGIN
  const(cid: id): const?
  variable(vid : varid) : var?
  lamb(bnd:(var?), body:term) : lamb?
  app(op: term, args: list[term]): app?
 END term
```

Note that the `args` accessor is of type `list[term]`. There is a restriction on the types of the accessors; in this case the only allowable types available for accessors involving the type `term` are: `term`, `list[term]`, `setof[term]`, `sequence[term]`, or `pred[term]`. There is no restriction on type expressions that do not reference `term`. Some of the axioms are modified accordingly; in particular, the induction axioms and recursive combinators are modified to handle the list argument.

4 The PVS Proof Checker

4.1 Introduction

Just as we execute programs to check if they return the desired result, we subject high-level functional descriptions of a system to challenges by demanding proofs of desirable properties. We call such challenges *putative theorems*. Here are some simple examples:

- If a function that reverses a list has been correctly specified, then we should be able to prove that we get the original list by reversing a list twice.

[30] In order to preserve soundness, PVS requires all user-defined recursive functions to include a *measure*, which is used to generate termination conditions. The primary use of the recursive combinator is to build measure functions for recursive definitions. (Measure functions themselves cannot usually be defined by recursive definitions, since those definitions require an existing measure function.)

- When a train is allowed into a railroad crossing the gates must be down.
- If the operational semantics is correct, then it should conform to the denotational semantics.

The point of these challenges is that the process of proving putative theorems quickly highlights the gaps, errors, and inadequacies in the functional description. In some cases, such a proof could also prove that the system as described meets its complete specification, in other words, that it is *correct*. But it is the rare proof that succeeds. The typical proof attempt fails, but such failure usually yields valuable insights that can be used to correct oversights in the specification or formulation of the challenge. A useful automated proof assistant must therefore play the role of an intelligent but implacable skeptic in rejecting any argument that is not entirely watertight. Furthermore, in rejecting these arguments, such a skeptic must pinpoint the source of the failure so that the argument can be corrected and the dialogue resumed. The PVS proof checker is intended to serve as the skeptical party in such a dialogue. The user supplies the steps in the argument and PVS applies them to the goal of the proof progressively breaking them into simpler subgoals or to obvious truths or falsehoods. If all of the subgoals are reduced to obvious truths, the proof attempt has succeeded. Otherwise, the proof attempts fails either because the argument or the conjecture is incorrect.

The central design assumptions in PVS are therefore that

- The purpose of an automated proof checker is not merely to prove theorems but also to provide useful feedback from failed and partial proofs by serving as a rigorous skeptic.
- The most straightforward mechanizable criterion for a rigorous argument is that of a formal proof.
- Automation serves to minimize the tedious aspects of formal reasoning while maintaining a high level of accuracy in the book-keeping and formal manipulations.
- Automation should also be used to capture repetitive patterns of argumentation.
- The end product of a proof attempt should be a proof that, with only a small amount of work, can be made humanly readable so that it can be subjected to the *social process* of mathematical scrutiny.

In following these design assumptions, the PVS proof checker is more automated than a low-level proof checker such as AUTOMATH [dB80], LCF [GMW79], Nuprl [Con86], Coq [CH85], and HOL [Gor88], but provides more user control over the structure of the proof than highly automated systems such as Nqthm [BM79, BM88] and Otter [McC90]. We feel that the low-level systems over-emphasize the formal correctness of proofs at the expense of their cogency, and the highly automated systems emphasize theorems at the expense of their proofs.

What is unusual about PVS is the extent to which aspects of the language, the typechecker, and proof checker are intertwined. The typechecker invokes the proof checker in order to discharge proof obligations that arise from typechecking expressions involving predicate subtypes or dependent types. The proof checker also makes heavy use of the typechecker to ensure that all expressions involved in a proof are well-typed. This use of the typechecker can also generate proof obligations that are either discharged automatically or are presented as additional subgoals. Several aspects of the language, particularly the type system, are built into the proof checker. These include the automatic use of type constraints by the decision procedures, the simplifications given by the abstract datatype axioms, and forms of beta-reduction and extensionality, Another less unusual aspect of PVS is the extent to which decision procedures involving equalities and linear arithmetic inequalities are employed.[31] The most direct consequence of this is that the trivial, obvious, or tedious parts of the proof are often entirely hidden from the displayed proof so that the user can focus on the intellectually demanding parts of the proof, and the resulting proof is also easier to read.

As with much else in PVS, the implementation philosophy of the proof checker has been guided by the 80-20 rule, namely that 80% of the functionality of a nearly perfect system can be built with 20% of the effort, and the remaining 20% of the functionality can take up the remaining 80% of the effort. PVS attempts to provide much of the 80% of the functionality that is easily implemented. Each PVS proof commands performs the function that, in our experience, is typically required of it. To some reasonable extent, the less typical functionality can be obtained by providing optional arguments to these proof commands. In atypical instances, the burden of carrying out some manipulation falls squarely on the user. Even in these instances, it is not too tedious to achieve one's ends with the existing proof commands in some fairly simple ways. The reader should let us know if any of our design decisions are found to be ill-considered.

In order to learn how to use the PVS proof checker, one must first understand the sequent representation used by PVS to represent proof goals, the commands used to move around and undo parts of the proof tree, and the commands used to get help. One must then understand the syntax and effects of proof commands used to build proofs. Many of these commands are extremely powerful even in their simplest usage. Several of these commands can be more carefully directed by supplying them with one or more optional arguments. The advanced user will also need to understand how to define proof strategies that capture repetitive patterns of proof commands, and commands used for displaying, editing, and replaying proofs.

Section 4.2 provides the basic information needed to get started with the PVS proof checker. The remaining sections give a collection of typical examples of how the proof checker is used. The PVS Proof Checker Reference Manual [SOR93] contains detailed descriptions of the PVS proof commands.

4.2 Preliminaries

Sequent Representation of Proof Goals. Each goal or subgoal in a PVS proof attempt is a sequent of the form $\Gamma \vdash \Delta$, where Γ is a sequence of *antecedent* formulas and Δ is a sequence of *consequent* formulas. The actual displayed form of a PVS sequent is

[31] The Ontic system [McA89] is a proof checker where decision procedures are ubiquitously used. Nqthm [BM79, BM88], Eves [PS89], and IMPS [FGT91] also rely heavily on the use of decision procedures.

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

where each A_i is an antecedent formula and each B_i is a consequent formula. The intuitive reading of such a sequent is as the formula

$$(A_1 \wedge A_2 \wedge A_3 \wedge \ldots) \supset (B_1 \vee B_2 \vee B_3 \vee \ldots).$$

Note that the antecedent formulas are numbered with negative integers and the consequent formulas with positive integers. These numberings are used in directing the PVS proof commands. If a formula number n appears as `[`n`]` in the sequent, it is an indication that the formula was unaffected by the proof step that created the sequent. It is a good heuristic is to examine the new formulas (*i.e.*, the formulas whose number appears as $\{n\}$) in the sequent to formulate the next proof step.

Starting and Quitting Proofs. As indicated earlier, the PVS Emacs command `M-x pr` initiates a proof with the cursor on the formula to be proved. This brings up the `*pvs*` buffer with the goal sequent and a `Rule?` prompt. Typing the PVS Emacs command `M-x help-pvs-prover` brings up help on the prover commands. To quit out of an existing proof attempt, type `q` or `quit` at the `Rule?` prompt. You will be asked whether you wish to save the partial proof. Remember that if you answer `yes`, the old proof will be overwritten, and if you answer `no`, you will lose the partial proof that you have developed up to this point.

Since PVS proof construction is carried out in a Lisp buffer, there is a small chance that you could find yourself at a Lisp breakpoint with a '`->`' prompt. Typing `(restore)` at this point should almost always take you back to the nearest sensible proof goal and a `Rule?` prompt.

The Structure of PVS Proofs. In the course of a proof, PVS builds up a tree of sequents where each sequent is a subgoal generated from its parent sequent by a PVS proof command. At any point in a proof attempt, the control is at a leaf sequent of such a proof tree. At this point a PVS proof command can either

- cause control to be transferred to next proof sequent in the tree (`postpone`)
- undo a subtree by causing control to move up to some ancestor node in the proof tree (`undo`)
- prove the *current sequent* causing control to move to the next remaining leaf sequent in the tree

- generate subgoals so that control moves to the first of these subgoals, or
- leave the proof tree unchanged while providing some useful status information.

A proof is completed when there are no remaining unproved leaf sequents in the proof tree. The resulting proof is saved and can be edited and rerun on the same or a different conjecture.

4.3 Using the Proof Checker

Propositional Proof Commands

Now that we have gotten past the preliminaries, we can look at examples of some simple interactions with the PVS proof checker. We start with the following PVS theory named `propositions` that declares three Boolean constants `A`, `B`, and `C`, and states a theorem named `prop` asserting that the conjunction of (`A` $\supset$ (`B` $\supset$ `C`)) and (`A` $\supset$ `B`) and `A` implies `C`.

```
propositions : THEORY
  BEGIN

  A, B, C: bool

  prop: THEOREM (A IMPLIES (B IMPLIES C)) AND (A IMPLIES B) AND A
               IMPLIES C

  END propositions
```

The proof script displayed below is the result of typing the PVS Emacs command `M-x pr` on the formula `prop` and typing the inputs (shown in bold-face) in response to the `Rule?` prompt or to other queries from PVS. The `(flatten)` command eliminates the disjunctive connectives in the formula so as to flatten the formula out into the sequent. The next proof command `(split)` picks the first available conjunctive formula, in this case `(A IMPLIES (B IMPLIES C))`, and generates the three subgoals resulting from the conjunctive splitting of this formula. PVS then observes that the first of these subgoals is trivially true since it has `C` in both the antecedent and consequent. The `(split)` command applied to the second subgoal generates two further subgoals which are both recognized as being trivially true, as is the remaining subgoal from the earlier `(split)` command. The proof has now been successfully completed generating the `Q.E.D.` message, and the new proof is automatically saved. The system inquires whether the user would like to see an abbreviated version of the proof which is then printed out following the `yes` response. For space reasons, we only display a few lines of this printout in the script below. The two timings printed out at the end provide the machine time and the human time for the proof attempt, respectively. The Emacs command `M-x show-last-proof` can be used to bring up an abbreviated version of the most recently completed proof that can be used as a guide in developing an informal presentation of the proof. It displays the sequents at the branch points in the proof and the commentary in between.

```
prop :

  |-------
{1}   (A IMPLIES (B IMPLIES C)) AND (A IMPLIES B) AND A IMPLIES C

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

{-1}   (A IMPLIES (B IMPLIES C))
{-2}   (A IMPLIES B)
{-3}   A
  |-------
{1}   C

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

{-1}   C
[-2]   (A IMPLIES B)
[-3]   A
  |-------
[1]   C

which is trivially true.

This completes the proof of prop.1.

prop.2 :

[-1]   (A IMPLIES B)
[-2]   A
  |-------
{1}   B
[2]   C

Rule? (split)
Splitting conjunctions,
this yields  2 subgoals:
prop.2.1 :

{-1}   B
[-2]   A
  |-------
[1]   B
[2]   C

which is trivially true.

This completes the proof of prop.2.1.

prop.2.2 :

[-1]   A
  |-------
{1}   A
[2]   B
[3]   C

which is trivially true.
```

```
This completes the proof of prop.2.2.


This completes the proof of prop.2.

prop.3 :

[-1]   (A IMPLIES B)
[-2]   A
  |-------
{1}   A
[2]   C

which is trivially true.

This completes the proof of prop.3.

Q.E.D.


Run time  = 0.52 secs.
Real time = 14.32 secs.
```

Summary. The PVS Emacs command `M-x pr` is used to invoke the PVS proof checker. Proof goals are represented as sequents with the formulas numbered. The command `(flatten)` flattens the top-level disjunctive structure of all of the sequent formulas so that there are no disjunctive formulas in the resulting subgoal sequent. (Variations: `(flatten *)` is the same as `(flatten)`. `(flatten +)` flattens only the consequent formulas, and `(flatten -)` the antecedent formulas. `(flatten -2 3 4)` flattens formulas numbered `-2`, `3`, and `4` in the goal sequent.) The command `(split)` picks the first top-level conjunctive sequent formula and generates the subgoals that result from splitting this conjunction. As with `flatten`, `(split *)` is the same as `(split)`, `(split -)` splits the first antecedent conjunction, `(split +)` the first consequent conjunction, and `(split -3)` splits the formula numbered `-3`.

With the same example, we can now attempt to repeat the proof in order to explore some other commands. When we now type `M-x pr` at the formula `prop` in the theory `proposition`, PVS informs us that the formula has already been proved and asks whether we wish to retry proving the formula. If we respond that we do, then PVS inquires whether the existing proof should be rerun. If we choose to rerun the existing proof, the following script is automatically generated.

```
prop :

  |-------
{1}   (A IMPLIES (B IMPLIES C)) AND (A IMPLIES B) AND A IMPLIES C

Rerunning step: (FLATTEN)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
prop :
```

```
{-1}   (A IMPLIES (B IMPLIES C))
{-2}   (A IMPLIES B)
{-3}   A
  |-------
{1}   C

Rerunning step: (SPLIT)
Splitting conjunctions,
this yields  3 subgoals:
prop.1 :

{-1}   C
[-2]   (A IMPLIES B)
[-3]   A
  |-------
[1]   C

which is trivially true.

This completes the proof of prop.1.

   ⋮
```

Summary. Proofs can be rerun by responding suitably to the mini-buffer query when `M-x pr` is invoked on a formula that has a proof or a partial proof. Another way to rerun the existing proof is to type `(rerun)` as the first step in a manual proof.

We can retry the same example to explore some further proof commands. In this version, we choose not to rerun the existing proof. Typing the inappropriate command `(split)` results in `No change` to the proof state since there is no top level conjunctive formula in the sequent. We then type `(flatten)` which flattens the formula followed by `(split)` which generates three subgoals, the first of which is trivially true. We then type `(postpone)` at the second subgoal. This causes the control to shift to the third subgoal which is also trivially true. The control now returns to the second subgoal. A further `(postpone)` brings us back to the same subgoal since there are no other pending subgoals. At this point, we simply choose to quit the proof by typing `q` at the `Rule?` prompt. At the query, we choose to save the partial proof from the current proof attempt.

```
prop :

  |-------
{1}   (A IMPLIES (B IMPLIES C)) AND (A IMPLIES B) AND A IMPLIES C

Rule? (split)
No change.
prop :

  |-------
{1}   (A IMPLIES (B IMPLIES C)) AND (A IMPLIES B) AND A IMPLIES C

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

```
prop :

{-1}   (A IMPLIES (B IMPLIES C))
{-2}   (A IMPLIES B)
{-3}   A
  |-------
{1}   C

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

{-1}   C
[-2]   (A IMPLIES B)
[-3]   A
  |-------
[1]   C

which is trivially true.

This completes the proof of prop.1.

prop.2 :

[-1]   (A IMPLIES B)
[-2]   A
  |-------
{1}   B
[2]   C

Rule? (postpone)
Which takes us to,
prop.3 :

[-1]   (A IMPLIES B)
[-2]   A
  |-------
{1}   A
[2]   C

which is trivially true.

This completes the proof of prop.3.

prop.2 :

[-1]   (A IMPLIES B)
[-2]   A
  |-------
{1}   B
[2]   C

Rule? (postpone)
Which takes us to,
prop.2 :

[-1]   (A IMPLIES B)
[-2]   A
  |-------
{1}   B
[2]   C

Rule? q
```

```
Do you really want to quit?   (Y or N):  y
Would you like the partial proof to be saved?
(***Old proof will be overwritten.***)
 (Yes or No)  yes
Use M-x revert-proof to revert to previous proof.

Run time  = 0.77 secs.
Real time = 22.63 secs.
```

We can again type `M-x pr` and this time we can rerun the partial proof that we saved. Notice that we are back at the subgoal where we quit the proof since this is the only unfinished subgoal in the proof.

Summary. The command `(postpone)` is used to navigate cyclically around the unproved subgoals in a proof. The PVS Emacs command `M-x siblings` displays all those subgoals that share the same parent goal as the current subgoal in the proof. The PVS Emacs command `M-x ancestry` displays the chain of goals leading back from the current goal back to the root node of the proof tree. A `q` or `quit` can be used to quit out of a proof-in-progress with the option of saving the partial proof. If a previous proof is overwritten as a result, then the PVS Emacs command `M-x revert-proof` can be used to recover the earlier proof. The PVS Emacs command `M-x show-proof` can be used to display a proof in progress in such a way that parts of it can be edited and used as input to the `rerun` proof command. The PVS Emacs command `M-x edit-proof` with the cursor positioned on a formula in a theory brings up a buffer containing the proof of the formula displayed as a tree of commands. This displayed proof can also be edited and rerun.

Quantifier Proof Commands

We now consider a simple example involving quantifiers displayed in the theory `predicate` below.

```
predicate: THEORY
  BEGIN
   T : TYPE
   x, y, z: VAR T
   P, Q : [T -> bool]

   pred_calc: THEOREM
      (FORALL x: P(x) AND Q(x))
      IMPLIES (FORALL x: P(x)) AND (FORALL x: Q(x))

  END predicate
```

The proof script for this example starts with the application of `(flatten)` to the given conjecture followed by the `(split)` command to break the consequent conjunction. In the first branch of the proof, we use the `(skolem)` command to replace the universally quantified variable `x` in the consequent formula numbered `1` with the (Skolem) constant `X`, where X is new (*i.e.*, undeclared) in the present context. The next step is to instantiate the universally quantified variable x in the antecedent formula numbered -1 with the constant X using the (quant) command. The first branch of the proof is then easily completed by propositional reasoning. Note that the two quantifier steps, skolem and quant, only affect the outermost quantifier of a formula in the sequent. Also, universally quantified variables in consequent formulas are replaced by new constants, whereas antecedent universally quantified variables are instantiated with terms. Existentially quantified variables behave dually. The second branch of the proof employs minor variants of the skolem and quant. Here the (skolem!) command picks the first "skolemizable" sequent formula and replaces the quantified variables with internally generated constants (containing exclamations). The (quant?) command picks the first instantiable sequent formula and tries to find an instantiation for the quantified variables by matching against the rest of the sequent.

```
pred_calc :

  |-------
{1}   (FORALL x: P(x) AND Q(x)) IMPLIES (FORALL x: P(x)) AND (FORALL x: Q(x))

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

{-1}   (FORALL x: P(x) AND Q(x))
  |-------
{1}   (FORALL x: P(x)) AND (FORALL x: Q(x))

Rule? (split)
Splitting conjunctions,
this yields  2 subgoals:
pred_calc.1 :

[-1]   (FORALL x: P(x) AND Q(x))
  |-------
{1}   (FORALL x: P(x))

Rule? (skolem 1  "X")
For the top quantifier in 1, we introduce Skolem constants: X
this simplifies to:
pred_calc.1 :

[-1]   (FORALL x: P(x) AND Q(x))
  |-------
{1}   P(X)

Rule? (quant -1  "X")
Instantiating the top quantifier in -1 with the terms:
 X
this simplifies to:
pred_calc.1 :

{-1}   P(X) AND Q(X)
  |-------
[1]   P(X)

Rule? (prop)
By propositional simplification,
```

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

pred_calc.2 :

[-1]   (FORALL x: P(x) AND Q(x))
  |-------
{1}   (FORALL x: Q(x))

Rule? (skolem!)
For the top quantifier in 1, we introduce Skolem constants: (x!1)
this simplifies to:
pred_calc.2 :

[-1]   (FORALL x: P(x) AND Q(x))
  |-------
{1}   Q(x!1)

Rule? (quant?)
Found substitution:
x gets x!1,
Instantiating quantified variables,
this simplifies to:
pred_calc.2 :

{-1}   P(x!1) AND Q(x!1)
  |-------
[1]   Q(x!1)

Rule? (prop)
By propositional simplification,

This completes the proof of pred_calc.2.

Q.E.D.
```

Summary. The command `(skolem 1 "X")` is used to introduce a new constant `X` in place of the universally quantified variable in the formula numbered `1`. `(skolem 1 ("X" "_" "Z"))` is to be used if there are three variables bound by the universal quantifier and only the first and third are to be replaced by constants. `(skolem + "X")` carries out the skolemization step for the first consequent universally quantified formula, and `(skolem - "X")` for the first antecedent existentially quantified formula. The variations of the instantiation command `quant` are similar to those of `skolem`. The command forms `(skolem!)`, `(skolem! 1)`, `(skolem! -)`, etc., are variants of `skolem` where the new constant names are internally generated. The command `(quant?)` is a version of `quant` that tries to find a matching substitution for a chosen quantified formula. It can also be supplied a partial substitution to disambiguate the matching process as in `(quant? - :subst ("x" "X"))`. Both `quant` and `quant?` take an optional `:copy?` argument that can be given as `T` in order to retain a copy of the original quantified formula in the sequent in case further instances of the formula are needed, as in `(quant + ("x" "X") :copy? T)`. The PVS rule `inst-cp` is a version of the `quant` that automatically copies the quantified formula, and `inst` is the non-copying variant of `quant`. Correspondingly, `inst?` is synonymous with `quant?`. Note that optional arguments to PVS proof commands can be given by order or by keyword. To find out the order, the keywords, and defaults for each of the proof commands, use `M-x help-pvs-prover`.

Decision Procedures

The equality and linear inequality decision procedures are the workhorses of almost any nontrivial PVS proof. The theory `decisions` displayed below illustrates some of the power of these decision procedures. The formulas marked `THEOREM` are those that can be proved using the decision procedures, and the ones marked `CONJECTURE` are either true but cannot be proved solely by the decision procedures (like `badarith1`) or false (like `badarith` and `badarith2`) and hence unprovable. The reader should invoke `M-x pr` on each of the formulas in `decisions` and type either `(then (skolem!)(ground))` or `(then* (skolem!)(flatten)(assert))` to the `Rule?` prompt to observe the effects of the decision procedures. The command `assert` is used to either record equality or inequality information into the data-structures used by the decision procedures, or to simplify propositional or `IF-THEN-ELSE` structures in a formula, or carry out the automatic rewrites (to be described below). The command `(ground)` is a combination of `(prop)` and `(assert)`.

```
decisions: THEORY
  BEGIN
   x,y,v: VAR number
   f: [number -> number]

   eq1: THEOREM x = f(x) IMPLIES f(f(f(x))) = x

   g : [number, number -> number]

   eq2: THEOREM x = f(y) IMPLIES g(f(y + 2 - 2), x + 2) = g(x, f(y) + 2)

   arith: THEOREM  %Proved by decision procedures
     x < 2*y AND y < 3*v IMPLIES 3*x < 18*v

   badarith: CONJECTURE %Not proved; statement is false.
     x < 2*y AND y < 3*v IMPLIES 3*x < 17*v

   badarith1: CONJECTURE %Not proved; statement true but non-linear
     x<0 AND y<0 IMPLIES x*y>0

   i, j, k: VAR int

   intarith: THEOREM %Proved by decision procedures
     2*i < 5 AND i > 1 IMPLIES i = 2

   badarith2: CONJECTURE  %Not proved; stmt. true of integers but not reals.
     2*x < 5 AND x > 1 IMPLIES x = 2

   range : THEOREM  %Proved by decision procedures
     i > 0 AND i < 3 IMPLIES i = 1 OR i = 2

  END decisions
```

We now consider an example proof that further illustrates the use of decision procedures. The theory `stamps` below contains the formula asserting that any postage requirement of

8 cents or more can be met solely with 3 and 5 cent stamps, *i.e.*, is the sum of some multiple of 3 and some multiple of 5.

```
stamps: THEORY
  BEGIN

  i, three, five: VAR nat

  stamps: LEMMA (FORALL i: (EXISTS three, five: i + 8 = 3 * three + 5 * five))

  END stamps
```

In abstract terms, the proof proceeds by induction on `i`. In the base case, when `i` is `0`, the left-hand side is 8. Letting `m` and `n` both be `1` fulfills the equality. In the induction case, we know that `that i + 8` can be expressed as `3*M + 5*N` for some `M` and `N` and we need to find `m` and `n` such that `i + 8 + 1` is `3*m + 5*n`. If `N = 0`, then `M` is at least `3`. We then let `m` be `M - 3` and `n` be `2`, *i.e.*, we remove three 3 cent stamps and add two 5 cent stamps to get postage worth `i + 8 + 1`. If `N > 0`, then we simply remove a 5 cent stamp and add two 3 cent stamps to prove the induction conclusion.

In the proof script below, the first command `(induct "i")` directs PVS to use induction on `i`. PVS deduces from the type `nat` of `i` that natural number induction is to be used and formulates an induction predicate based on the formula number 1 in the sequent. The command `induct`, like `prop` and `ground`, is a compound step or a *proof strategy*. Two subgoals are generated corresponding to the base and induction cases. In the base case, the `quant` command is used to instantiate `three` with `1` and `five` with `1`. The decision procedures are invoked to prove the resulting trivial arithmetic equality. In the induction case, the `skolem` command followed by `flatten` results in a sequent containing the induction hypothesis in its antecedent and the conclusion in its consequent part. The witnesses corresponding to the induction hypothesis are produced by the `skolem!` command. The case-split according to `five!1 = 0` is created by the `case` command. In the first `five!1 = 0` case, we instantiate the existential quantifiers in the conclusion as required by the abstract proof. Since the bound variable `three` has type `nat` (which is a subtype of the `integer` type consisting of the non-negative integers), the `quant` command generates a second (type correctness) subgoal demanding proof that `three!1 - 3` is at least `0`. Both subgoals are discharged through the use of `assert`. In the case when `five!1 = 0` is false, note that the assumption of falsity is indicated by the formula `five!1 = 0` appearing in the consequent part of the goal sequent. We now follow an approach that is slightly different from that of the previous branch; we use `assert` at this point. This has no visible effect on the sequent to be proved, but the falsity of `five!1 = 0` is noted by the decision procedures for use deeper in the proof. Now note that the `quant` command instantiating `five` with `five!1 - 1` does not generate the type correctness subgoal that was generated in the previous branch since the decision procedures were able to automatically demonstrate that `five!1 - 1` was non-negative from the known information.

```
stamps :

  |-------
{1}   (FORALL i: (EXISTS three, five: i + 8 = 3 * three + 5 * five))

Rule? (induct "i")
Inducting on i,
this yields  2 subgoals:
stamps.1 :

  |-------
{1}   (EXISTS (three: nat), (five: nat): 0 + 8 = 3 * three + 5 * five)

Rule? (quant 1 (1 1))
Instantiating the top quantifier in 1 with the terms:
 (1 1)
this simplifies to:
stamps.1 :

  |-------
{1}   0 + 8 = 3 * 1 + 5 * 1

Rule? (assert)
Invoking decision procedures,

This completes the proof of stamps.1.

stamps.2 :

  |-------
{1}   (FORALL (j: nat):
         (EXISTS (three: nat), (five: nat): j + 8 = 3 * three + 5 * five)
           IMPLIES
           (EXISTS (three: nat), (five: nat):
              j + 1 + 8 = 3 * three + 5 * five))

Rule? (skolem + "JJ")
For the top quantifier in +, we introduce Skolem constants: JJ
this simplifies to:
stamps.2 :

  |-------
{1}   (EXISTS (three: nat), (five: nat): JJ + 8 = 3 * three + 5 * five)
        IMPLIES
        (EXISTS (three: nat), (five: nat): JJ + 1 + 8 = 3 * three + 5 * five)

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
stamps.2 :

{-1}   (EXISTS (three: nat), (five: nat): JJ + 8 = 3 * three + 5 * five)
  |-------
{1}   (EXISTS (three: nat), (five: nat): JJ + 1 + 8 = 3 * three + 5 * five)

Rule? (skolem!)
For the top quantifier in -1, we introduce Skolem constants: (three!1 five!1)
this simplifies to:
stamps.2 :

{-1}   JJ + 8 = 3 * three!1 + 5 * five!1
  |-------
[1]   (EXISTS (three: nat), (five: nat): JJ + 1 + 8 = 3 * three + 5 * five)
```

```
Rule? (case "five!1 = 0")
Case splitting on
   five!1 = 0,
this yields  2 subgoals:
stamps.2.1 :

{-1}   five!1 = 0
[-2]   JJ + 8 = 3 * three!1 + 5 * five!1
  |-------
[1]   (EXISTS (three: nat), (five: nat): JJ + 1 + 8 = 3 * three + 5 * five)

Rule? (quant + ("three!1 - 3" 2))
Instantiating the top quantifier in + with the terms:
 (three!1 - 3 2)
this yields  2 subgoals:
stamps.2.1.1 :

[-1]   five!1 = 0
[-2]   JJ + 8 = 3 * three!1 + 5 * five!1
  |-------
{1}   JJ + 1 + 8 = 3 * (three!1 - 3) + 5 * 2

Rule? (assert)
Invoking decision procedures,

This completes the proof of stamps.2.1.1.

stamps.2.1.2 :

[-1]   five!1 = 0
[-2]   JJ + 8 = 3 * three!1 + 5 * five!1
  |-------
{1}   three!1 - 3 >= 0
[2]   (EXISTS (three: nat), (five: nat): JJ + 1 + 8 = 3 * three + 5 * five)

Rule? (assert)
Invoking decision procedures,

This completes the proof of stamps.2.1.2.


This completes the proof of stamps.2.1.

stamps.2.2 :

[-1]   JJ + 8 = 3 * three!1 + 5 * five!1
  |-------
{1}   five!1 = 0
[2]   (EXISTS (three: nat), (five: nat): JJ + 1 + 8 = 3 * three + 5 * five)

Rule? (assert)
Invoking decision procedures,
this simplifies to:
stamps.2.2 :

[-1]   JJ + 8 = 3 * three!1 + 5 * five!1
  |-------
[1]   five!1 = 0
[2]   (EXISTS (three: nat), (five: nat): JJ + 1 + 8 = 3 * three + 5 * five)

Rule? (quant + ("three!1 + 2" "five!1 - 1"))
Instantiating the top quantifier in + with the terms:
 (three!1 + 2 five!1 - 1)
this simplifies to:
```

```
stamps.2.2 :

[-1]   JJ + 8 = 3 * three!1 + 5 * five!1
  |-------
[1]   five!1 = 0
{2}   JJ + 1 + 8 = 3 * (three!1 + 2) + 5 * (five!1 - 1)

Rule? (assert)
Invoking decision procedures,

This completes the proof of stamps.2.2.


This completes the proof of stamps.2.

Q.E.D.
```

Summary. PVS proofs make heavy use of decision procedures to simplify tedious equality and arithmetic reasoning so that the number of trivial subgoals can be minimized and to keep the sequent formulas simple. The equality decision procedure employs congruence closure to propagate equality information along the term structure to quickly decide whether a sequent containing equalities and other propositions is true. An antecedent formula P that is not an equality can be treated as $P=$ `TRUE`, and a consequent formula P as the equality $P=$ `FALSE`. The `assert` rule is the most powerful form in which decision procedures are applied. It is a combination of the `record` rule which records sequent formulas in the data-structures used by the decision procedures, `simplify` which simplifies branching and propositional structure using the decision procedures, `beta` which beta-reduces record, tuple, function-update, `LAMBDA`, and abstract datatype redexes, and `do-rewrite` which applies the rewrites specified by `auto-rewrite` and `auto-rewrite-theory`.

The `(case` $\langle formula \rangle^*$`)` command used in the above proof is extremely useful for case-splitting on a formula. For example, if there is no straightforward way to simplify a formula A to another formula A', then one can case-split on A' so that we can use A' on one branch and prove it from A on the other branch. The `case` command can also be used to replace a term s by s' by case-splitting on $s = s'$ and using the `replace` proof command (which is not explained here) to carry out the replacement.

Using Definitions and Lemmas

For the purpose of this discussion, we use the following very simple example of a recursive function that halves a given natural number.

```
half: THEORY
  BEGIN

  i, j, k: VAR nat

  half(i): RECURSIVE nat =
      (IF i = 0 THEN 0 ELSIF i = 1 THEN 0 ELSE half(i - 2) + 1 ENDIF)
```

```
    MEASURE (LAMBDA i: i)

half_halves: THEOREM half(2 * i) = i

half_half: THEOREM half(2 * half(2 * i)) = i

END half
```

We show a segment of the proof of `half_halves` where the definition of `half` is expanded. Notice that the first use of `expand` brings in an unsimplified expansion of the definition of `half`. When we `undo` this proof step and retry the same `expand` step following an `assert`, not only is the expansion simplified, but the equality is itself reduced to `TRUE`.

```
  ⋮
half_halves.2 :

{-1}   half(2 * J) = J
  |-------
{1}   half(2 * (J + 1)) = J + 1

Rule? (expand "half" +)
Expanding the definition of half
this simplifies to:
half_halves.2 :

[-1]   half(2 * J) = J
  |-------
{1}   (IF 2 * (J + 1) = 0 THEN 0 ELSE half(2 * (J + 1) - 2) + 1 ENDIF) = J + 1

Rule? (undo)
This will undo the proof to:
half_halves.2 :

{-1}   half(2 * J) = J
  |-------
{1}   half(2 * (J + 1)) = J + 1
Sure? (Y or N): y
half_halves.2 :

{-1}   half(2 * J) = J
  |-------
{1}   half(2 * (J + 1)) = J + 1

Rule? (assert)
Invoking decision procedures,
this simplifies to:
half_halves.2 :

[-1]   half(2 * J) = J
  |-------
[1]   half(2 * (J + 1)) = J + 1

Rule? (expand "half" +)
Expanding the definition of half
this simplifies to:
half_halves.2 :

[-1]   half(2 * J) = J
```

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

which is trivially true.
   ⋮
```

The `rewrite` command is an alternative to `expand`, though `rewrite` can be used to rewrite with both formulas and definitions. In the script below, the `rewrite` step replaces the second of the above applications of `expand`. Notice that `rewrite` behaves slightly differently from `expand`, but it too is sensitive to the facts recorded by the decision procedures from a previous `assert`.

```
   ⋮
half_halves.2 :

[-1]   half(2 * J) = J
  |-------
[1]   half(2 * (J + 1)) = J + 1

Rule? (rewrite "half" +)
Rewriting using half,
this simplifies to:
half_halves.2 :

[-1]   half(2 * J) = J
  |-------
{1}   half(2 * (J + 1) - 2) + 1 = J + 1

Rule? (assert)
Invoking decision procedures,

This completes the proof of half_halves.2.
   ⋮
```

In summary, `expand` is used to expand definitions, and `rewrite` is used to rewrite using definitions and formulas. Both employ decision procedures for simplification during rewriting. Decision procedures are also used to discharge any conditions (arising from a conditional rewrite rule) and the type-correctness conditions arising from the lemma instantiation applied by `rewrite`. The `expand` step is the preferred way to expand definitions.

Other Commands. We have described some typical commands but have omitted to mention many other commands. The `lemma` command is used to bring in an instance of a lemma as an antecedent sequent formula. The `extensionality` proof command is similarly used to bring in the extensionality scheme given a suitable type expression, *i.e.*, a function, record, or tuple type or an abstract datatype. The `beta` rule is used to carry out beta-reduction of redexes including those involving `LAMBDA`-abstraction, record access, tuple access, function updates, and datatype expressions. The command `delete` can be used to drop irrelevant sequent formulas; `hide` is a more conservative form of `delete` where the formula can be restored using the `reveal` command. The PVS Emacs command `M-x show-hidden` shows the hidden formulas. The command `typepred` can be used to make the subtype predicates on a given expression explicit as sequent formulas. The `lift-if` command lifts `IF`-branching to the top-level of a sequent formula through F`(IF` A `THEN` s `ELSE` t `ENDIF)` being transformed to `(IF` A `THEN` $F(s)$ `ELSE` $F(t)$ `ENDIF)`. The commands `auto-rewrite` and `auto-rewrite-theory` are used to install rewrite rules to be used automatically by the `assert` command.

User-defined *proof strategies* can be employed by more advanced PVS users. These are similar to the tactics and tacticals of LCF. The PVS Prover Checker Reference Manual [SOR93] can be consulted for more on this topic.

5 Exercises

Problem 1 *Based on the discussion of the specification of stacks, try to specify a PVS theory formalizing queues. Can the PVS abstract datatype facility be used for specifying queues?*

Problem 2 *Specify binary trees with value type* `T` *as a parametric abstract datatype in PVS.*

Problem 3 *Specify a PVS theory formalizing* ordered *binary trees with respect to a type parameter* `T` *and a given total-ordering relation,* i.e., *define a predicate* `ordered?` *that checks if a given binary tree is ordered with respect to the given total ordering.*

Problem 4 *Prove the stack axioms for the definitions stated in* `newstacks`*.*

Problem 5 *Prove the theorems in the theory* `half` *(Page 43).*

Problem 6 *Define the operation for carrying out the ordered insertion of a value into an ordered binary tree. Prove that the insertion operator applied to an ordered binary tree returns an ordered binary tree.*

References

[AJ90] Heather Alexander and Val Jones. *Software Design and Prototyping using me too.* Prentice Hall International, Hemel Hempstead, UK, 1990.

[BCM+90] J. R. Burch, E. M. Clarke, K. L McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking 2^{20} states and beyond. In *5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, PA, June 1990. IEEE Computer Society.

[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.

[BM91] Robert S. Boyer and J Strother Moore. MJRTY—a fast majority vote algorithm. In Robert S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, volume 1 of *Automated Reasoning Series*, pages 105–117. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991.

[CH85] T. Coquand and G.P. Huet. Constructions: A higher order proof system for mechanizing mathematics. In *Proceedings of EUROCAL 85, Linz (Austria)*. Springer-Verlag, 1985.

[Con86] R. L. Constable, *et al.*. *Implementing Mathematics with the Nuprl.* Prentice-Hall, New Jersey, 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.

[EGMS79] B. Elspas, M. Green, M. Moriconi, and R. Shostak. A JOVIAL verifier. Technical report, Computer Science Laboratory, SRI International, January 1979.

[FGT91] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An interactive mathematical proof system. Technical Report M90-19, Mitre Corporation, 1991.

[GHW85] John V. Guttag, James J. Horning, and Jeannette M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[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, 1988.

[HI88] Sharam Hekmatpour and Darrel Ince. *Software Prototyping, Formal Methods, and VDM.* International Computer Science Series. Addison-Wesley, Wokingham, England, 1988.

[McA89] D. A. McAllester. *ONTIC: A Knowledge Representation System for Mathematics.* MIT Press, 1989.

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

[MSR85] P. Michael Melliar-Smith and John Rushby. The Enhanced HDM system for specification and verification. In *Proc. VerkShop III*, pages 41–43, Watsonville, CA, February 1985. Published as ACM Software Engineering Notes, Vol. 10, No. 4, Aug. 85.

[ORSvH93] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Some lessons learned. In *Formal Methods Europe '93*, Lecture Notes in Computer Science, Odense, Denmark, April 1993. Springer Verlag. Accepted for publication.

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

[Pra92] Sanjiva Prasad. Verification of numerical programs using Penelope/Ariel. In *COMPASS '92 (Proceedings of the Seventh Annual Conference on Computer Assurance)*, pages 11–24, Gaithersburg, MD, June 1992. IEEE Washington Section.

[PS89] W. Pase and M. Saaltink. Formal verification in m-EVES. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Theorem Proving*, pages 268–302, New York, NY, 1989. Springer Verlag.

[RL76] Lawrence Robinson and Karl N. Levitt. Proof techniques for hierarchically structured programs. *Communications of the ACM*, 20(4):271–283, April 1976.

[RLS79] L. Robinson, K. N. Levitt, and B. A. Silverberg. *The HDM Handbook*. Computer Science Laboratory, SRI International, Menlo Park, CA, June 1979. Three Volumes.

[RvHO91] John Rushby, Friedrich von Henke, and Sam Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1991.

[SGGH91] James B. Saxe, Stephen J. Garland, John V. Guttag, and James J. Horning. Using transformations and verification in circuit design. Technical Report 78, DEC Systems Research Center, Palo Alto, CA, September 1991.

[SOR93] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[SSMS82] R. E. Shostak, R. Schwartz, and P. M. Melliar-Smith. STP: A mechanized logic for specification and verification. In D. Loveland, editor, *6th International Conference on Automated Deduction (CADE)*, volume 138 of *Lecture Notes in Computer Science*, New York, NY, 1982. Springer Verlag.