A Tutorial Introduction to PVS

Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995

Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, Mandayam Srivas*
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

WWW: `http://www.csl.sri.com/sri-csl-fm.html`

Updated June 1995

Abstract

This document provides an introductory example, a tutorial, and a compact reference to the PVS verification system. It is intended to provide enough information to get you started using PVS, and to help you appreciate the capabilities of the system and the purposes for which it is suitable.

*Dave Stringer-Calvert provided valuable comments on earlier versions of this tutorial, and also checked the specifications and proofs appearing here. Preparation of this tutorial was partially funded by NASA Langley Research Center under Contract NAS1-18969, and by the Advanced Research Projects Agency through NASA Ames Research Center NASA-NAG-2-891 (Arpa order A721) to Stanford Unversity.

Overview 1

I Introduction to Mechanized Analysis of Specifications Using PVS 3

1 Introduction 5

2 An Electronic Phone Book: Simple Version 5

3 A Better Version of the Specification Using Sets 16

4 Version of the Specification That Maintains An Invariant 20

5 Summary 25

II Tutorial on Using PVS 27

1 Introducing PVS 29
1.1 Design Goals for PVS . . . 30
1.2 Uses of PVS . . . 31
1.3 Getting and Using PVS . . . 31

2 A Brief Tour of PVS 32
2.1 Creating the Specification . . . 33
2.2 Parsing . . . 34
2.3 Typechecking . . . 34
2.4 Proving . . . 35
2.5 Status . . . 38
2.6 Generating LaTeX . . . 38

3 The PVS Language 40
3.1 A Simple Example: The Rational Numbers . . . 40
3.2 A More Sophisticated Example: Stacks . . . 43
3.3 Implementing Stacks . . . 44
3.4 Using Theories: Partial and Total Orders . . . 46
3.5 Using Theories: Sort . . . 47
3.6 Sets in Higher-order Logic . . . 51
3.7 Recursion . . . 52
3.8 Dependent Typing . . . 53
3.9 Abstract Datatypes: Stacks . . . 54
3.10 Abstract Datatypes: Terms . . . 56

4 The PVS Proof Checker **57**
4.1 Introduction 57
4.2 Preliminaries 59
4.3 Using the Proof Checker 60
Propositional Proof Commands 60
Quantifier Proof Commands 66
Decision Procedures 68
Using Definitions and Lemmas 73
Proof Checker Pragmatics 75

5 Two Hardware Examples **76**
5.1 A Pipelined Microprocessor 77
Informal Description 77
Formal Specification 78
Proof of Correctness 81
5.2 An N-bit Ripple-Carry Adder 84
Typechecking 85
Proof of Adder_correct_n 86

6 Exercises **88**

III PVS Reference **89**

1 PVS Files **91**

2 PVS Language Summary **92**

3 PVS Emacs Commands **98**

4 PVS Prover Commands **103**

References **108**

Overview

PVS is a *verification system*: an interactive environment for writing formal specifications and checking formal proofs. It builds on nearly 20 years experience at SRI in building verification systems, and on substantial experience with other systems. The distinguishing feature of PVS is its synergistic integration of an expressive specification language and powerful theorem-proving capabilities. PVS has been applied successfully to large and difficult applications in both academic and industrial settings.

PVS provides an expressive specification language that augments classical higher-order logic with a sophisticated type system containing predicate subtypes and dependent types, and with parameterized theories and a mechanism for defining abstract datatypes such as lists and trees. The standard PVS types include numbers (reals, rationals, integers, naturals, and the ordinals to ϵ_0), records, tuples, arrays, functions, sets, sequences, lists, and trees, etc. The combination of features in the PVS type-system is very convenient for specification, but it makes typechecking undecidable. The PVS typechecker copes with this undecidability by generating proof obligations for the PVS theorem prover. Most such proof obligations can be discharged automatically. This liberation from purely algorithmic typechecking allows PVS to provide relatively simple solutions to issues that are considered difficult in some other systems (for example, accommodating partial functions such as division within a logic of total functions), and it allows PVS to enforce very strong checks on consistency and other properties (such as preservation of invariants) in an entirely uniform manner.

PVS has a powerful interactive theorem prover/proof checker. The basic deductive steps in PVS are large compared with many other systems: there are atomic commands for induction, quantifier reasoning, automatic conditional rewriting, simplification using arithmetic and equality decision procedures and type information, and propositional simplification using binary decision diagrams. The PVS proof checker manages the proof construction process by prompting the user for a suitable command for a given subgoal. The execution of the given command can either generate further subgoals or complete a subgoal and move the control over to the next subgoal in a proof. User-defined proof strategies can be used to enhance the automation in the proof checker. Model-checking capabilities used for automatically verifying temporal properties of finite-state systems have recently been integrated into PVS. PVS's automation suffices to prove many straightforward results automatically; for hard proofs, the automation takes care of the details and frees the user to concentrate on directing the key steps.

PVS is implemented in Common Lisp—with ancillary functions provided in C, Tcl/TK, and LaTeX—and uses GNU Emacs for its interface. Configured for Sun Sparc Workstations running under SunOS 4.1.3, the system is freely available under license from SRI.

PVS has been used at SRI to undertake proofs of difficult fault-tolerant algorithms [LR93a,LR93b,LR94], to verify the microcode for selected instructions of a complex, pipelined, commercial microprocessor having 500,000 transistors where seeded and unseeded errors were found [MS95], to provide an embedding for the Duration Calculus (an interval temporal logic [SS94]), and for several other applications. PVS is installed at many sites worldwide, and is in serious use at about a dozen of them. There is a growing list of significant applications undertaken using PVS by people outside SRI. Many of these can be examined at the WWW site `http://www.csl.sri.com/sri-csl-fm.html`.

This tutorial is intended to give you an idea of the flavor of PVS, of the opportunities created by effective mechanization of formal methods, and an introduction to the use of the system itself. PVS is a big and complex system, so we can really only scratch the surface here. To make advanced use of the system, you should study the manuals (there are three volumes: language [OSR93a], prover [SOR93], and system [OSR93b]), and some of the more substantial applications.

There are three parts to this tutorial.

- *An Introduction to the Mechanized Analysis of Requirements Specifications Using PVS.* This tutorial introduction shows how PVS can be used to actively explore and analyze a simple requirements specification. It is intended to demonstrate the utility of mechanized support for formal methods, and the opportunities for validation and exploration that are created by effective mechanization.

- *Tutorial on Using PVS.* This introduces many of the capabilities of PVS by means of simple examples and takes you through the process of using the system. While it can be read as an overview, it is best to have PVS available and to actively follow along.

- *PVS Reference.* This presents all PVS system and prover commands, and illustrates the language constructs in a very compact form.

A useful supplement to the material presented here is [ORSvH95], which describes some of the larger verifications undertaken using PVS and also motivates and describes some of the design decisions underlying PVS.

Part I

Introduction to Mechanized Analysis of Specifications Using PVS

1 Introduction

Simply using a formal notation does not ensure that specifications will be correct: writing a correct formal specification is no easier than writing a correct program or a correct description in English. Specifications—especially *requirements* specifications, where there is no higher-level specification against which they can be verified—need to be *validated* against informal expectations. This is generally done by human review and inspection (which can be very formalized processes), but with formal specifications it is possible to do more.

The distinctive feature of *formal* specifications is that they support formal deduction: it is possible to reduce certain questions about a formal specification to a process that resembles calculation and that can be checked by others or by machine. Thus, reviews and inspections can be supplemented by *analyses* of formal specifications, and those analyses can be mechanically checked.

In order to conduct mechanized analysis, it is necessary to support a specification language with powerful tools including, primarily, a theorem prover. The needs of efficient theorem proving drive specification language design in slightly different directions than for unmechanized notations such as Z, but the presence of mechanization also creates new linguistic opportunities—such as allowing typechecking to use theorem proving—that can enhance the clarity and precision of specifications.

PVS is a *verification system*: a specification language tightly integrated with a powerful theorem prover and other tools. This document is intended to serve as a first introduction to PVS: it is not intended to teach the details of the PVS language and theorem prover, but rather to give an appreciation of the opportunities created by mechanized analysis in general, and of some of the capabilities of PVS in particular.

2 An Electronic Phone Book: Simple Version

Suppose we are to formally specify the requirements for an electronic phone book, given the following informal description.[1]

- A phone book shall store the phone numbers of a city
- It shall be possible to retrieve a phone number given a name
- It shall be possible to add and delete entries from a phone book

Examining this description, we see that there are three types of entities mentioned: *phone books*, *phone numbers* and *names*; a phone book provides an association between names and phone numbers. We need three operations, which we can call *FindPhone*, *AddPhone*, and *DelPhone*. *FindPhone* should take a phone book and a name and return the phone number associated with that name. The exact functionality of the other two operations is less clear, so we have to make some design decisions. We decide that *AddPhone* should take a phone book, a name, and a phone number and should add the association

[1] This example is based on one by Ricky Butler and Sally Johnson of NASA Langley [BJ93].

between the name and number to the phone book; and that *DelPhone* should take a phone book and a name and delete the phone number associated with that name (if any).

The next step is decide how to represent these entities and operations in PVS. If we were programming, we would have to choose some specific representations for phone numbers and names—e.g., ascii strings, or more structured representations such as records containing the area-code and number—and would have to make several design decisions at this point. But for requirements specification, all we require is that phone numbers and names are distinguishable *types* of entities. In PVS, we can specify this as follows (a % sign introduces a comment that extends to the end of the line).

```
N: TYPE                 % names
P: TYPE                 % phone numbers
```

These types are *uninterpreted*, meaning that we know nothing about their members—not even whether they are zero, many, or infinite in number—except that elements of type `N` are distinguishable from those of type `P`, and that there is an equality predicate on each type (i.e., given two `P`s, it is possible to tell whether they are the same or not).

Next, we need to describe how phone books—associations between names and numbers—are to be represented. There are several possibilities: one is to record each association as a `(name, phone number)` pair, so that a phone book is a set of such pairs; another is as a function from names to phone numbers (you can think of a function as an array if that notion is more familiar to you). PVS is able to reason very effectively with functions, so there is some advantage to the latter representation. We can specify this as follows.

```
B: TYPE = [N -> P]    % phone books
```

This says that phone books have the *type* `B`, and are functions from names to phone numbers.

We must recognize that not all names will be in every phone book—a phone book only records those names that have a phone number—so we need some way to distinguish those names that have a phone number from those that do not. In the specification language Z, for example, this would be accomplished by specifying that phone books are *partial* functions. Efficient theorem proving, however, strongly encourages use of *total* functions, so PVS is a logic of total functions.[2] One way to indicate that a name has no phone number is to identify some particular phone number, represented by `n0` say, to indicate this fact. Of course we need to mentally make note that this number must be different from any "real" phone number (we will see later how we can enforce this requirement, and later still we will see a better way to deal this whole issue of names that have no phone number). Give this decision, we can next specify the empty phone book as the (unique) phone book that maps all names to `n0`. I will specify this axiomatically, later we will see how to do it definitionally.

```
n0: P
emptybook: B
emptyax: AXIOM   FORALL (nm: N): emptybook(nm) = n0
```

[2]PVS can represent partial functions very nicely using *dependent* types, but that is an advanced topic.

If we were programming an implementation, a literal translation of this representation would be grossly inefficient: it requires "space" for every possible name and it explicitly records for every name that there is no number associated with the name. When programming, we would seek more compact representations that traded off space for efficient access—perhaps a hash table or balanced binary tree. In requirements specification, however, the idea is simply to record the functionality required, and it is not our concern to suggest an efficient implementation.

We can specify the `FindPhone` operation as a function that takes a phone book and a name and returns the phone number associated with that name.

```
FindPhone: [B, N -> P]
Findax: AXIOM   FORALL (bk: B), (nm: N):  FindPhone(bk, nm) = bk(nm)
```

Notice that this is a *functional* specification style: the "state" of the system we are interested in (i.e., the phone book) is passed to the `FindPhone` function as an argument; this is in contrast to a more procedural style of specification (as in Z, for example), where there is a built-in notion of state. Functional specifications use conventional logic and can be mechanized straightforwardly, whereas procedural specifications involve some kind of Hoare logic—for which it is rather more difficult to provide mechanized deduction.

The distinction between functional and procedural kinds of specification is revealed more clearly in the case of our next operation, `AddPhone`. In a procedural specification, this operation would update the state of the phone book "in place." In the functional style used here, we model the operation by a function that takes a phone book, a name, and a number, and gives us back a "new" phone book in which the association between the name and number has been added.

```
AddPhone: [B, N, P -> B]
Addax: AXIOM   FORALL (bk: B), (nm: N), (pn: P):
   AddPhone(bk, nm, pn) = bk WITH [(nm) := pn]
```

The `WITH` construct is similar to function overriding in Z.

Now that we have specified two operations, perhaps we should check our understanding of them. If we were programming, we might run a couple of test cases. Some people advocate something similar (often called "animation") for specifications. This is generally feasible only with specifications that have a constructive character (i.e., that are essentially very high-level programs). Not all specifications are best presented in this way, however, so the desire to make specifications executable can distort their other characteristics. Another way to probe a specification is by means of "formal challenges." These are putative theorems: general statements that we think should to be true if our specification says what it ought to. This can yield more information than an individual test case (it is generally equivalent to running a whole class of test cases), and uses theorem proving (i.e., search), rather than direct execution, so it is possible even when the specification is not constructive. (If the specification is constructive—as in this example—then theorem proving generally comes down to symbolic execution and is very efficient.) A suitable challenge for the specification we have so far is: "if I add a name `nm` with phone number `pn` to a phone book and look up the name `nm`, I should get back the phone number `pn`." We can write this as follows.

```
  FindAdd: CONJECTURE  FORALL (bk: B), (nm: N), (pn: P):
    FindPhone(AddPhone(bk, nm, pn), nm) = pn
```

In order to test this conjecture, we have to extend the specification into a complete PVS "theory" (as modules are called in PVS). This is shown in Figure 1. Then we load the specification into PVS, parse and typecheck it, and start the prover. The mechanics of doing this are described in other PVS tutorial documents. Briefly, PVS uses an extended GNU Emacs as its interface, and PVS system functions are invoked by Emacs keystrokes. To invoke the prover, for example, place the cursor on the `CONJECTURE` and type `M-x prove` (this will automatically parse and typecheck if necessary).

```
phone_1: THEORY

BEGIN

  N: TYPE              % names
  P: TYPE              % phone numbers
  B: TYPE = [N -> P]   % phone books

  n0: P
  emptybook: B
  emptyax: AXIOM   FORALL (nm: N): emptybook(nm) = n0

  FindPhone: [B, N -> P]
  Findax: AXIOM   FORALL (bk: B), (nm: N):  FindPhone(bk, nm) = bk(nm)

  AddPhone: [B, N, P -> B]
  Addax: AXIOM   FORALL (bk: B), (nm: N), (pn: P):
     AddPhone(bk, nm, pn) = bk WITH [(nm) := pn]

  FindAdd: CONJECTURE  FORALL (bk: B), (nm: N), (pn: P):
    FindPhone(AddPhone(bk, nm, pn), nm) = pn

END phone_1
```

Figure 1: Specification Ready for Checking the First Challenge

Starting the prover on the `FindAdd` conjecture produces the following display.

```
FindAdd :

  |-------
{1}   FORALL (bk: B), (nm: N), (pn: P): FindPhone(AddPhone(bk, nm, pn), nm) = pn

Rule?
```

This is a *sequent*: in general there will be several numbered formulas above the turnstile symbol `|-------`, and several below. The idea is that we have to establish that the conjunction (and) of the formulas above the turnstile implies the disjunction (or) of the formulas

below the line. The `Rule?` prompt indicates that PVS is waiting for us to type a prover command. These use lisp syntax, with pieces of PVS syntax embedded in quotes: for example: `(grind :theories ("phone_1"))`.

This introduction is intended to describe the purpose and value of mechanized theorem proving in analysis of requirements specification; it is not intended as a tutorial on the PVS prover, so I will not explain all the various choices and considerations at each step. The prover provides a number (about 20) basic commands, and a similar-sized collection of higher-level commands called "strategies" that are programmed using the basic commands. You type a command at the `Ready?` prompt, and the prover applies the command and presents you with the transformed sequent and another prompt. When the prover recognizes that a sequent is trivially true, it terminates that branch of the proof. Some commands may split the proof into branches, in which case you will be presented with one of the branches, and the others will be remembered and popped up when the current branch terminates. When all branches are terminated the theorem is proved.

On straightforward theorems (and the straightforward parts of difficult theorems), it is generally best to use the highest-level, most automated strategies, and only to resort to the basic commands for crucial steps. The highest-level strategy is called `grind`. It does skolemization, heuristic instantiation, propositional simplification (using BDDs), if-lifting, rewriting, and applies decision procedures for linear arithmetic and equality. It takes several optional arguments which mostly supply the names of the formulas that can be used for automatic rewriting (i.e., replacing of an instance of the left hand side of an equation by the corresponding instance of the right hand side). In this case, we need to tell it that all the definitions and axioms in the theory `phone_1` may be used as rewrites. The command above does this, and is sufficient to prove the challenge.

```
Rule? (grind :theories ("phone_1"))
Addax rewrites AddPhone(bk, nm, pn)
  to bk WITH [(nm) := pn]
Findax rewrites FindPhone(bk WITH [(nm) := pn], nm)
  to pn
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.
```

Encouraged by this small confirmation that we are on the right track we can return to specifying the `DelPhone` operation. This is specified in a similar way to `AddPhone`.

```
  DelPhone: [B, N -> B]
  Delax: AXIOM   FORALL (bk: B), (nm: N): DelPhone(bk, nm) = bk WITH [(nm) := n0]
```

We can similarly test our understanding of this specification by checking the intuition that adding a name and phone number to a book and then deleting them leaves the book unchanged.

```
  DelAdd: CONJECTURE  FORALL (bk: B), (nm: N), (pn: P):
    DelPhone(AddPhone(bk, nm, pn), nm) = bk
```

The same proof strategy as before fails to prove the conjecture and produces the following result.

```
DelAdd :

  |-------
{1}   FORALL (bk: B), (nm: N), (pn: P): DelPhone(AddPhone(bk, nm, pn), nm) = bk

Rule? (grind :theories ("phone_1"))
Addax rewrites AddPhone(bk, nm, pn)
  to bk WITH [(nm) := pn]
Delax rewrites DelPhone(bk WITH [(nm) := pn], nm)
  to bk WITH [(nm) := pn] WITH [(nm) := n0]
Trying repeated skolemization, instantiation, and if-lifting, this simplifies to:
DelAdd :

  |-------
{1}   bk!1 WITH [(nm!1) := pn!1] WITH [(nm!1) := n0] = bk!1

Rule?
```

The identifiers with `!` in them are Skolem constants—arbitrary representatives for quantified variables. This sequent is requiring us to prove that two functions (i.e., phone books) are the same: one that has been modified by adding a name and then removing it, another that is unchanged. To prove that two functions are the same, we appeal to the principle of *extensionality*, which says that this is so if the values of the two functions are identical for every point in their domains.

```
Rule? (apply-extensionality)
  Applying extensionality, this simplifies to:
DelAdd :

  |-------
{1}   bk!1 WITH [(nm!1) := pn!1] WITH [(nm!1) := n0](x!1) = bk!1(x!1)
[2]   bk!1 WITH [(nm!1) := pn!1] WITH [(nm!1) := n0] = bk!1

Rule? (delete 2)
  Deleting some formulas, this simplifies to:
DelAdd :

  |-------
[1]   bk!1 WITH [(nm!1) := pn!1] WITH [(nm!1) := n0](x!1) = bk!1(x!1)

Rule?
```

It is always possible to delete formulas from a sequent; here I have deleted the original formula to reduce clutter, since it is the extensional form that is interesting. This sequent is asking us to show that the phone number associated with an arbitrary name `x!1` is the same both before and after the phone book has been updated for name `nm!1`. A

case-analysis is appropriate here, according to whether or not `x!1 = nm!1`. This can be accomplished by the `(lift-if)` command, which converts `WITH` expressions to their corresponding `IF-THEN-ELSE` form. The `(ground)` command (a slightly less muscular command than `(grind)`) then takes care of the various cases, except for one.

```
DelAdd :

  |-------
[1]   bk!1 WITH [(nm!1) := pn!1] WITH [(nm!1) := n0](x!1) = bk!1(x!1)

Rule? (lift-if)
Lifting IF-conditions to the top level,
this simplifies to:
DelAdd :

  |-------
{1}   IF nm!1 = x!1 THEN n0 = bk!1(x!1)
      ELSE IF nm!1 = x!1 THEN n0 = bk!1(x!1)
        ELSE bk!1(x!1) = bk!1(x!1)
        ENDIF
      ENDIF

Rule? (ground)
Applying propositional simplification and decision procedures,
this simplifies to:
DelAdd :

{-1}   nm!1 = x!1
  |-------
{1}   n0 = bk!1(x!1)

Rule?
```

(A `(grind)` command would have performed both these steps.) For this sequent to be true, we need to to demonstrate that if `x!1 = nm!1`, then the phone number originally associated with `x!1` is the special number `n0`. But, by virtue of the equality, this is the same as asking us to prove that the phone number originally associated with `nm!1` is `n0`—and there is no reason why this should be true! Suddenly, we understand the problem: if the number associated with `nm!1` beforehand was a real phone number, `nm!2`, say, then the `AddPhone` operation *changes* the association to the new number, and the `DelPhone` operation changes it again to `n0`—which is not equal to `nm!2`. Thus our conjecture is only true under the assumption that the name we add to the phone book currently has no number associated with it. We can test this by modifying the conjecture as follows.

```
DelAdd2: CONJECTURE  FORALL (bk: B), (nm: N), (pn: P):
  FindPhone(bk, nm) = n0 => DelPhone(AddPhone(bk, nm, pn), nm) = bk
```

And the `(grind :theories ("phone_1"))` strategy proves this.

Another conjecture is that the result of adding a name and then deleting it is the same as just deleting it.

```
DelAdd3: CONJECTURE  FORALL (bk: B), (nm: N), (pn: P):
  DelPhone(AddPhone(bk, nm, pn), nm) = DelPhone(bk, nm)
```

The `(grind :theories ("phone_1"))` strategy proves this conjecture also.

Notice how our inability to prove the original `DelAdd` conjecture exposed a deficiency in our specification and led us to discover the source of the deficiency. Individual test cases might have missed the particular circumstance that exposes the problem, but the strict requirements of mechanically checked proof systematically led us to examine all the cases until we discovered the one that manifested the problem.

Another conjecture we might try to prove is that after adding a name and phone number to the phone book, the number stored for that name is a "real" number (i.e., not `n0`).

```
KnownAdd: CONJECTURE  FORALL (bk: B), (nm: N), (pn: P):
  FindPhone(AddPhone(bk, nm, pn), nm) /= n0
```

The same kind of exploration with the prover will rapidly show that this is unprovable because there is nothing that requires the `pn` argument to `AddPhone` to be a "real" phone number.

Our exploration of this specification has revealed a couple of deficiencies.

1. `AddPhone` has the side effect of changing the phone number when applied to someone who already has a number.

2. Our specification does not rule out the possibility of giving someone `n0` as a phone number

We can deal with the second deficiency by introducing a type `GP` of "good phone numbers" as a *subtype* of P, with the constraint that `n0` is not a member of `GP`. In PVS, this is done by means of a *predicate subtype*, which can be written as follows.

```
GP: TYPE = { pn: P | pn /= n0 }
```

We will see later that predicate subtypes are a very powerful element of the PVS specification language. Here we can make simple use of them by changing the *signature* of the `AddPhone` function from `[B, N, P -> B]` to `[B, N, GP -> B]`, and this will automatically prevent the addition of `n0` to a phone book as a real number.

We can deal with the first deficiency noted above by dividing the functionality of `AddPhone` in two: the revised `AddPhone` will make no change to the phone book if the name concerned already has a phone number, and the new `ChangePhone` operator will change an existing number, but will not add a number to a name that currently lacks one.

In order to specify these functions, it is convenient to add a predicate `Known?` that takes a phone book and a name and returns `true` if that name has a "real" phone number in the book concerned. (A *predicate* is just a function whose range type is boolean.) This can be specified as follows.

```
Known?: [B, N -> bool]
Known_ax: AXIOM   FORALL (bk: B), (nm: N): Known?(bk, nm) = (bk(nm) /= n0)
```

This axiomatic style of specification has the disadvantage that axioms can introduce inconsistencies. An individual axiom is seldom dangerous: rather, danger lies in the interactions among several axioms. For example, with the original signature and definition of `AddPhone`, adding the following axiom to that above yields an inconsistent specification.

```
Whoops: AXIOM   FORALL (bk: B), (nm: N), (pn, P): Known?(AddPhone(bk, nm, pn), nm)
```

Inconsistent specifications are dangerous because they can be used to prove anything at all,[3] and because they cannot be implemented. It is disturbingly easy to introduce inconsistent axioms, so it is generally best to use them sparingly. Axioms are really needed only when it is necessary to constrain (rather than fully define) the values of a function, or when it is necessary to constrain the interactions of several functions. When the intent is to fully define the values of a function, it is generally better to state it as a *definition*, since PVS will then check that it is indeed a "conservative extension" (and therefore does not introduce an inconsistency).

The predicate `Known?` can be introduced by means of a definition by replacing the two lines used earlier (the specification of its signature and axiom) by the following single line.

```
Known?: [B, N -> bool] = LAMBDA (bk: B), (nm: N): bk(nm) /= n0
```

The use of `LAMBDA` notation can be a little daunting, so PVS allows an alternative, "applicative," form of definition as follows.

```
Known?(bk: B, nm: N): bool = bk(nm) /= n0
```

The need to specify the types of the variables in this declaration can be eliminated by declaring them separately.

```
bk: VAR B
nm: VAR N
Known?(bk, nm): bool = bk(nm) /= n0
```

In this way, the previous axiomatic specification for `AddPhone` can be changed to the following definition, which incorporates the refinement that the function does not change the phone book if the name already has a number known for it.

```
gp: VAR GP
AddPhone(bk, nm, gp): B =
  IF Known?(bk, nm) THEN bk ELSE bk WITH [(nm) := gp] ENDIF
```

[3] For example, when used in conjunction with the `AXIOM`s `emptyax`, `Known_ax`, and `Addax`, `Whoops` allows us to prove `true = false`.

We can check that these changes provide some of the properties we expect by considering the following formal challenge.

```
KnownAdd: CONJECTURE  FORALL bk, nm, gp: Known?(AddPhone(bk, nm, gp), nm)
```

This says that a name is definitely known (i.e., has a "real" phone number) after applying `AddPhone` to it. Notice that since the variables `bk`, `nm`, and `gp` have already been declared, there is no need to specify their types in the `FORALL` construction. In fact, there is no need to provide the `FORALL` construction at all: the following specification is equivalent to the one above, since PVS automatically interprets "free" variables as universally quantified at the outermost level.

```
KnownAdd: CONJECTURE  Known?(AddPhone(bk, nm, gp), nm)
```

This conjecture is easily proved by the `grind` strategy.

Proceeding in this way, we can construct the theory `phone_2` shown in Figure 2. All the conjectures in that theory are proved by the simple command `(grind)`. There is no need to specify auto-rewriting of the `phone_2` theory, since definitions are automatically available for rewriting (another advantage that they have over axioms).

If we try to add the dangerous `AXIOM Whoops` to this new specification, PVS will note that the third argument supplied to `Addphone (pn)` is a `P`, whereas the signature of `AddPhone` says it requires a `GP` in this position. PVS allows a value of a supertype to be used where one of a subtype is required, provided the value can be proven, in its context, to satisfy the predicate of the subtype concerned. The corresponding proof obligation is generated automatically by PVS as a Type-Correctness Condition (TCC). PVS does not consider a specification fully typechecked until all its TCCs have been proved (though you can postpone doing the proof until convenient). TCCs are displayed by the command `M-x show-tccs`; in the present case, the TCC generated by Whoops is the following.

```
% Subtype TCC generated (line 37) for pn
  % untried
whoops_TCC1: OBLIGATION (FORALL (pn: P): pn /= n0);
```

This is obviously unproveable (and untrue!), and the folly of adding the axiom `Whoops` is thereby brought to our attention.

Notice that if the `pn` in `Whoops` is changed to `gp`, then the formula not only becomes harmless (and no TCC is generated), but a proveable consequence of the definitions.

```
phone_2: THEORY

BEGIN

  N: TYPE               % names
  P: TYPE               % phone numbers
  B: TYPE = [N -> P]    % phone books

  n0: P

  GP: TYPE = {pn: P | pn /= n0}

  nm: VAR N
  pn: VAR P
  bk: VAR B
  gp, gp1, gp2: VAR GP

  emptybook(nm): P = n0

  FindPhone(bk, nm): P = bk(nm)

  Known?(bk, nm): bool = bk(nm) /= n0

  AddPhone(bk, nm, gp): B =
    IF Known?(bk, nm) THEN bk ELSE bk WITH [(nm) := gp] ENDIF

  ChangePhone(bk, nm, gp): B =
    IF Known?(bk, nm) THEN bk WITH [(nm) := gp] ELSE bk ENDIF

  DelPhone(bk, nm): B = bk WITH [(nm) := n0]

  FindAdd: CONJECTURE
    NOT Known?(bk, nm) => FindPhone(AddPhone(bk, nm, gp), nm) = gp

  FindChange: CONJECTURE
    Known?(bk, nm) => FindPhone(ChangePhone(bk, nm, gp), nm) = gp

  DelAdd: CONJECTURE
    DelPhone(AddPhone(bk, nm, gp), nm) = DelPhone (bk, nm)

  KnownAdd: CONJECTURE Known?(AddPhone(bk, nm, gp), nm)

  AddChange: CONJECTURE
    ChangePhone(AddPhone(bk, nm, gp1), nm, gp2) =
      AddPhone(ChangePhone(bk, nm, gp2), nm, gp2)

END phone_2
```

Figure 2: Revised Specification

3 A Better Version of the Specification Using Sets

The realization that `AddPhone` had the effect of changing the phone number associated with a name if that name already had a phone number led us to revise the specification so that `AddPhone` has no effect when the name already has a phone number. This treatment assumes that names can have at most one phone number associated with them. On reflection, or after consultation with the customer, we may decide that it is better to allow names to have multiple numbers associated with them. We can accommodate this by changing the range of the phone book function from a single phone number to a *set* of phone numbers as follows.

```
B: TYPE = [N -> setof[P]]   % phone books
```

This approach has the benefit that we now have a "natural" representation for names that do not have phone numbers: they can be associated with the emptyset of phone numbers.

A specification based on this approach is shown in Figure 3. The set-constructing functions such as `add`, `remove`, `emptyset`, etc., and the predicates on sets such as `disjoint?` are defined in a PVS *prelude* (i.e., built-in) theory called `set`. You can inspect this theory with the command `M-x view-prelude-theory`. A rather more attractive rendition of this specification is shown in Figure 4; this is produced by the command `M-x latex-theory`, which typesets the specification using LaTeX.

The first conjecture in this specification is easily proved using `(grind)`. The second one requires the more complex proof shown below.

```
("" (GRIND)
    (APPLY-EXTENSIONALITY)
    (DELETE 2)
    (LIFT-IF)
    (GROUND)
    (APPLY-EXTENSIONALITY)
    (DELETE 2)
    (GRIND))
```

This is the form in which PVS proofs are stored for later replay.

We have specified single additions to the phone book, but it seems likely that bulk additions will also be necessary. This will give us an opportunity to explore some more advanced features of the PVS language and prover. We would like to specify a function `AddList`, say, that takes a phone book and some collection of names and phone numbers and adds all of those names and phone numbers to the phone book. Each name-and-number is a pair, which can be represented in PVS by the tuple-type `[N, P]`. We could represent a collection of such pairs by either a sequence, or a list—a list is most convenient here, and is represented in PVS by the type `list[[N, P]]`. In this expression, the outermost brackets enclose the type parameter (here `[N, P]`) to the generic `list` theory (e.g., a list of phone numbers would be `list[P]`). In order to process such a list, we specify `AddList` as a recursive function that returns the phone book it is given if the list is empty, and otherwise recurses by applying the tail of the list to the phone book that results from applying `AddPhone` to the first name and number pair in the list.

```
phone_3 : THEORY

  BEGIN

  N: TYPE                       % names
  P: TYPE                       % phone numbers
  B: TYPE = [N -> setof[P]]     % phone books
  nm, x: VAR N
  pn: VAR P
  bk: VAR B

  emptybook(nm): setof[P] = emptyset[P]

  FindPhone(bk, nm): setof[P] = bk(nm)

  AddPhone(bk, nm, pn): B = bk WITH [(nm) := add(pn, bk(nm))]

  DelPhone(bk,nm): B = bk WITH [(nm) := emptyset[P]]

  DelPhoneNum(bk,nm,pn): B = bk WITH [(nm) := remove(pn, bk(nm))]

  FindAdd: CONJECTURE member(pn, FindPhone(AddPhone(bk, nm, pn), nm))

  DelAdd: CONJECTURE DelPhoneNum(AddPhone(bk, nm, pn), nm, pn) =
                DelPhoneNum(bk, nm, pn)

  END phone_3
```

Figure 3: Specification Using Set Constructions

```
phone_3: THEORY
  BEGIN

  N: TYPE

  P: TYPE

  B: TYPE = [N → setof[P]]

  nm, x: VAR N

  pn: VAR P

  bk: VAR B

  emptybook(nm): setof[P] = ∅_P

  FindPhone(bk, nm): setof[P] = bk(nm)

  AddPhone(bk, nm, pn): B = bk WITH [(nm) := {pn} ∪ bk(nm)]

  DelPhone(bk, nm): B = bk WITH [(nm) := ∅_P]

  DelPhoneNum(bk, nm, pn): B = bk WITH [(nm) := bk(nm) \ {pn}]

  FindAdd: CONJECTURE pn ∈ FindPhone(AddPhone(bk, nm, pn), nm)

  DelAdd: CONJECTURE
    DelPhoneNum(AddPhone(bk, nm, pn), nm, pn) = DelPhoneNum(bk, nm, pn)

  END phone_3
```

Figure 4: LaTeX-Printed Version of the Specification in Figure 3

```
  updates: VAR list[[N, P]]

  AddList(bk, updates): RECURSIVE B =
    CASES updates OF
      null: bk,
      cons(upd, rest): AddList(AddPhone(bk, proj_1(upd), proj_2(upd)), rest)
    ENDCASES
    MEASURE length(updates)
```

In this specification, the `CASES` expression introduces a pattern-matching enumeration over the constructors of an abstract data type (here, `list`), and the `proj_i` functions project out the `i`'th member of a tuple. The `MEASURE` clause indicates the argument that decreases across recursive calls (more generally, it specifies a function of the arguments, and an ordering relation according to which it decreases). PVS uses the `MEASURE` to generate a TCC to ensure that the function is total (i.e., that the recursion always "terminates"). In this case, the TCC is

```
% Termination TCC generated (line 48) for AddList

  AddList_TCC1: OBLIGATION
      (FORALL (rest: list[[N, P]], upd: [N, P], updates: list[[N, P]]):
         updates = cons[[N, P]](upd, rest)
           IMPLIES length[[N, P]](rest) < length[[N, P]](updates))
```

and it is proved automatically by PVS's standard strategy for proving TCCs (this strategy, called `(tcc)`, is a variety of `(grind)`).

The `list` datatype is specified in the PVS prelude using the datatype construction (similar to a "free type" in Z).

```
list [T: TYPE]: DATATYPE
 BEGIN
  null: null?
  cons (car: T, cdr:list):cons?
 END list
```

This specifies that `list` is a datatype that takes a single type parameter and has constructors `null` and `cons`, with corresponding recognizers and predicate subtypes `null?` and `cons?`, and accessors `car` and `cdr`. This specification expands internally into many axioms and definitions that are guaranteed to be conservative (i.e., not to introduce inconsistencies), and that are used very efficiently by the prover.

To validate our understanding of this function, we can try a couple of challenges. A reasonable expectation is that if a number is a member of the set of phone numbers for a given name, then it is still a member of that set after an arbitrary list of names and phone numbers have been added to the phone book.

```
  AddList_member: CONJECTURE
    member(pn, FindPhone(bk, nm)) =>
       member(pn, FindPhone(AddList(bk, updates), nm))
```

Like most conjectures involving recursively-defined functions, this one requires a proof by induction. PVS provides some powerful strategies for inductive proofs. Here, the single strategy `(induct-and-simplify "updates" :defs t)` is sufficient to prove the challenge. The argument `"updates"` is the name of the variable on which to induct, and `:defs t` instructs PVS that it may treat all definitions as rewrites. PVS automatically selects the correct induction rule (here, induction on lists), based on the type of the induction variable. The induction rule itself is defined automatically as part of the expansion of the `list` datatype definition.

A rather more complicated conjecture is that the set of phone numbers associated with a given name is unchanged when a list of names and phone numbers are added to the phone book if the given name is not mentioned in the list. This can be specified as follows.

```
FindList: CONJECTURE
  (every! (upd:[N, P]): proj_1(upd)/=nm) (updates) =>
    FindPhone(AddList2(bk, updates), nm) = FindPhone(bk, nm)
```

In this specification, `every!` introduces the body of a predicate that is true of all members of the list supplied as its argument (here, `updates`). It is another of the constructions defined automatically as a result of expanding the `list` datatype definition. As with the previous example, the `induct-and-simplify` strategy is able to prove this conjecture automatically.

4 Version of the Specification That Maintains An Invariant

A reasonable expectation is that the same phone number is never assigned simultaneously to two different names. We can extend the specification to ensure this by adding a predicate `UnusedPhoneNum` that returns `true` if a given number is not assigned to any name in a given phone book, and then modifying `AddPhone` to check the number being added is indeed unused.

```
UnusedPhoneNum(bk, pn): bool =
   (FORALL nm: NOT member(pn,FindPhone(bk, nm)))

AddPhone(bk, nm, pn): B =
  IF UnusedPhoneNum(bk, pn) THEN bk WITH [(nm) := add(pn, bk(nm))]
    ELSE bk
  ENDIF
```

If we've got this right, then it ought to be the case that the sets of phone numbers assigned to different names are always disjoint. We could generate a few challenges to check this, but we really want to be sure that the disjointness condition is an *invariant* of the specification. Recognizing this, we could try to generate the proof obligations that ensure this property. It is tedious and error-prone to generate proof obligations of this kind by hand, so some systems have special provision for generating the proof obligations necessary to guarantee invariants. PVS, however, can generate the necessary proof obligations as part of a much more general mechanism.

We have already seen that PVS allows predicate subtypes. The first step is to define those phone books that are "valid" as the subtype `VB` of phone books in which the sets of numbers associated with different names are disjoint.

```
VB: TYPE = { b:B | (FORALL (x,y: N): x /= y => disjoint?(b(x), b(y))) }
```

Then we change the specification of `FindPhone` to specify that it takes a `VB` and returns a `VB`:

```
bk: VAR VB

AddPhone(bk, nm, pn): VB =
  IF UnusedPhoneNum(bk, pn) THEN bk WITH [(nm) := add(pn, bk(nm))]
    ELSE bk
  ENDIF
```

Now the expression `bk WITH [(nm) := add(pn, bk(nm))]` appearing here is a `B`, but not necessarily a `VB`. But in order to satisfy the return type specified for `AddPhone`, this expression must be a `VB`. As already explained, PVS allows a value of a supertype to be used where one of a subtype is required, provided the value can be proven, in its context, to satisfy the predicate of the subtype concerned. The context here is `UnusedPhoneNum(bk, pn)`, so the proof obligation that needs to be discharged in order to ensure this specification is well-typed is the following.

```
% Subtype TCC generated (line 37) for bk WITH [(nm) := add(pn, bk(nm))]

AddPhone_TCC1: OBLIGATION
      (FORALL (bk: VB, nm: N, pn: P):
         UnusedPhoneNum(bk, pn) IMPLIES
           (FORALL (x, y: N):
              x /= y =>
                disjoint?[P](bk WITH [(nm) := add[P](pn, bk(nm))](x),
                             bk WITH [(nm) := add[P](pn, bk(nm))](y))));
```

This proof obligation is called a Type-Correctness Condition (TCC) and it is generated automatically by PVS. Proving it requires the following steps.

```
("" (GRIND :IF-MATCH NIL)
    (("1" (GRIND)) ("2" (INST -1 "x!1" "y!1")
                        (GRIND))
                   ("3" (GRIND))
                   ("4" (INST -1 "x!1" "y!1")
                        (GRIND))
                   ("5" (INST -1 "x!1" "y!1")
                        (GRIND))))
```

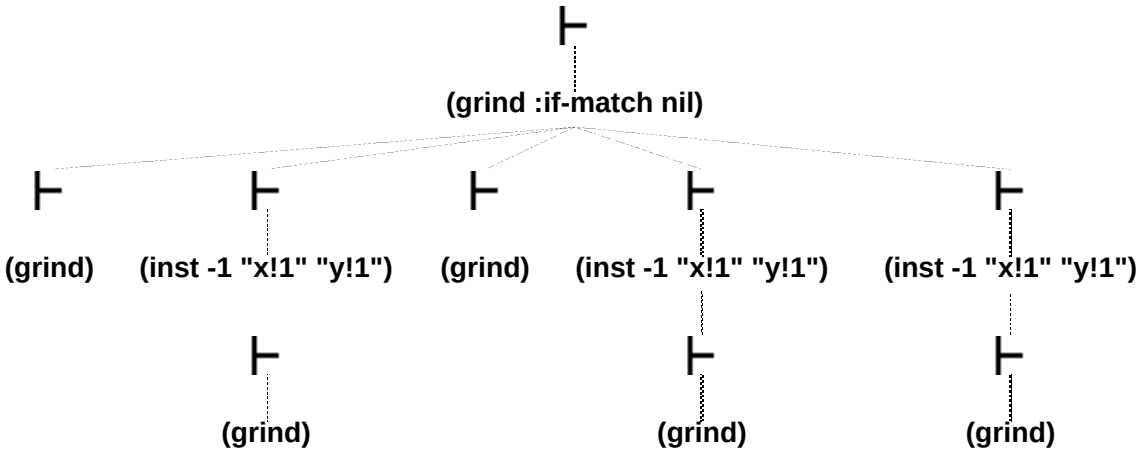

Figure 5: Graphical Display of the Proof Tree for TCC `AddPhone_TCC1`

Notice that the proof splits into several branches after the first step. PVS can generate a graphical display of the proof tree—which can then be saved as a postscript file—using the command `M-x x-show-proof`. The output for this proof is shown in Figure 5,

The important point to note, however, is that the close integration between language and prover in PVS allows the mechanization of very strong checks on specifications.

The full version of the specification of the previous section, adjusted to ensure that only valid phone books are generated is shown in figure 6 and the TCCs generated are shown in Figure 7.

```
phone_4: THEORY
  BEGIN

  N: TYPE

  P: TYPE

  B: TYPE = [N → setof[P]]

  VB: TYPE = {b: B | (∀ (x, y: N): x ≠ y ⇒ disjoint?(b(x), b(y)))}

  nm, x: VAR N

  pn: VAR P

  bk: VAR VB

  emptybook: VB = (λ (x: N): ∅_P)

  FindPhone(bk, nm): setof[P] = bk(nm)

  UnusedPhoneNum(bk, pn): bool = (∀ nm: ¬ pn ∈ FindPhone(bk, nm))

  AddPhone(bk, nm, pn): VB =
   IF UnusedPhoneNum(bk, pn) THEN bk WITH [(nm) := {pn} ∪ bk(nm)]
    ELSE bk
    ENDIF

  DelPhone(bk, nm): VB = bk WITH [(nm) := ∅_P]

  DelPhoneNum(bk, nm, pn): VB = bk WITH [(nm) := bk(nm) \ {pn}]

  FindAdd: CONJECTURE
     UnusedPhoneNum(bk, pn)
        ⊃ pn ∈ FindPhone(AddPhone(bk, nm, pn), nm)

  DelAdd: CONJECTURE
     DelPhoneNum(AddPhone(bk, nm, pn), nm, pn) = DelPhoneNum(bk, nm, pn)

  END phone_4
```

Figure 6: Specification Enforcing the Invariant that Different Names Have Disjoint Sets of Phone Numbers

```
% Subtype TCC generated (line 15) for (LAMBDA (x: N): emptyset[P])

emptybook_TCC1: OBLIGATION
      (FORALL (x, y: N): x /= y => disjoint?[P](emptyset[P], emptyset[P]));

% Subtype TCC generated (line 23) for bk WITH [(nm) := add(pn, bk(nm))]

AddPhone_TCC1: OBLIGATION
      (FORALL (bk: VB, nm: N, pn: P):
         UnusedPhoneNum(bk, pn) IMPLIES
           (FORALL (x, y: N):
              x /= y =>
                disjoint?[P](bk WITH [(nm) := add[P](pn, bk(nm))](x),
                             bk WITH [(nm) := add[P](pn, bk(nm))](y))));

% Subtype TCC generated (line 28) for bk WITH [(nm) := emptyset[P]]

DelPhone_TCC1: OBLIGATION
      (FORALL (bk: VB), (nm: N), (x, y: N):
         x /= y =>
           disjoint?[P](bk WITH [(nm) := emptyset[P]](x),
                        bk WITH [(nm) := emptyset[P]](y)));

% Subtype TCC generated (line 30) for bk WITH [(nm) := remove(pn, bk(nm))]

DelPhoneNum_TCC1: OBLIGATION
      (FORALL (bk: VB), (nm: N), (pn: P), (x, y: N):
         x /= y =>
           disjoint?[P](bk WITH [(nm) := remove[P](pn, bk(nm))](x),
                        bk WITH [(nm) := remove[P](pn, bk(nm))](y)));
```

Figure 7: TCCs for the Specification of Figure 6

5 Summary

It is no easier to write correct specifications than to write correct programs; just like programs, specifications need to be validated against their informal requirements and expectations. The mechanization provided by PVS allows the human inspections and reviews that are an essential element of validation to be supplemented by mechanically checked analyses.

I hope the example considered here has conveyed some appreciation for the opportunities created by mechanically supported formal specification. Other tutorials describe more of the mechanics of using PVS, and give examples of its use to verify algorithm correctness and to prove difficult theorems.

Part II

Tutorial on Using PVS

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. More realistic examples are provided in Section 5. 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, Sreeranga Rajan, Jens Skakkebæk, 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 [ORSvH95] 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 [Sha93a], a compiler for simple arithmetic expressions [Rus95], and several small hardware examples including pipeline and microcode correctness [CRSS94]. Examples of this scale can typically be completed within a day. More substantial examples include the correctness of a real-time railroad crossing controller [Sha93b], an embedding of the Duration Calculus [SS94], the correctness of some transformations used in digital syntheses [Raj94], and the correctness of distributed agreement protocols for a hybrid fault model consisting of Byzantine, symmetric, and crash faults [LR93a, LR93b, LR94]. These harder examples can take from several days to several weeks. Industrial applications of PVS include verification of selected elements of a commercial avionics microprocessor whose implementation has 500,000 transistors [MS95]. Some of these applications of PVS are summarized in [ORSvH95], which also motivates and describes some of the design decisions underlying PVS. Applications of PVS undertaken independently of SRI include [Hoo94, But93, JMC94, MPJ94].

1.3 Getting and Using PVS

At the moment, PVS is readily available only for Sun SPARC workstations running SunOS 4.1.3, although versions of the system have been run on IBM Risc 6000 (under AIX) and DECSystem 5000 (under Ultrix). PVS is implemented in Common Lisp (with CLOS), and has been ported to Lucid, Allegro, AKCL, CMULISP, and Harlequin Lisps. Only the Lucid and Allegro versions deliver acceptable performance. 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, although some facilities for presenting graphical representaitons of theory dependencies and proof trees (implemented in Tcl/TK) do require

[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.

X-Windows. In addition, LaTeX and an appropriate viewer are needed to support certain optional features of PVS.

PVS is quite large, requiring about 50 megabytes of disk space. In addition, any system on which it is to be run should have a minimum of 100 megabytes of swap space and 48 megabytes of real memory (more is better). 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. Alternatively, you may inspect the installation instructions over WWW at URL `http://www.csl.sri.com/pvs.html`. All installations of PVS must be licensed by SRI. The Lucid Lisp version requires that you have a runtime license for Lucid Lisp. 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

[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.

`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. Although the basic keyword commands described here are preferred by most serious users, PVS commands are also available as menu selections if you are running under EMACS 19.

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`).

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.[12] 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,[13] which can be done by

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

[12] In this case, the measure is the identity function, which could have been written simply as `MEASURE n`.

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

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.

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 two 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 8 below.

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

```
% Subtype TCC generated (line 7) for n - 1
  % unchecked
sum_TCC1: OBLIGATION (FORALL (n: nat): NOT n = 0 IMPLIES n - 1 >= 0);

% Termination TCC generated (line 7) for sum
  % unchecked
sum_TCC2: OBLIGATION (FORALL (n: nat): NOT n = 0 IMPLIES n - 1 < n);
```

Figure 8: TCCs for Theory `sum`

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.

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.[14] 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")`.[15] 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

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

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

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`).[16]

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`, we send the proof to the PVS decision procedures, which automatically decide certain fragments of arithmetic, by typing `(assert)`.[17] This 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[18], 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.[19]

The final step is to send the proof to the PVS decision procedures by typing `(assert)`. 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,
```

[16] 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.

[17] 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.

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

[19] 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].

```
this yields  2 subgoals:
closed_form.1 :

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

Rule?  (postpone)
Postponing 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?  (postpone)
Postponing closed_form.2.

closed_form.1 :

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

Rule?  (expand "sum")
(IF 0 = 0 THEN 0 ELSE 0 + sum(0 - 1) ENDIF)
simplifies to 0
Expanding the definition of sum,
this simplifies to:
closed_form.1 :

  |-------
{1}   0 = 0 / 2

Rule?  (assert)
Simplifying, rewriting, and recording with decision procedures,

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!)
Skolemizing,
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" +)
(IF j!1 + 1 = 0 THEN 0 ELSE j!1 + 1 + sum(j!1 + 1 - 1) ENDIF)
```

```
simplifies to 1 + sum(j!1) + j!1
Expanding the definition of sum,
this simplifies to:
closed_form.2 :

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

Rule?  (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of closed_form.2.

Q.E.D.


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

Note: The proof presented here is a low-level interactive one chosen for illustrative purposes. In practice, trivial theorems such as this are handled automatically by the higher-level strategies of PVS. This particular theorem, for example, is proved automatically by the single command `(induct-and-simplify "n" :defs T)`.

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.

After a few moments the previewer will pop up displaying the `sum` theory, as shown in Figure 9. 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

sum: THEORY
BEGIN
n: VAR nat
sum(n): RECURSIVE nat =
(IF n = 0 THEN 0 ELSE n + sum(n − 1) ENDIF)
MEASURE (λ n: n)
closed_form: THEOREM sum(n) = (n × (n + 1)) / 2
END sum

Figure 9: Theory `sum`

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

the output will look like Figure 10.

sum: **Theory**
BEGIN

n: VAR nat

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

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

END sum

Figure 10: 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} $\sum_{i=0}^{j'} i$ = (j' × (j' + 1)) / 2

{1} (IF j' + 1 = 0 THEN 0 ELSE j' + 1 + $\sum_{i=0}^{j'+1-1} i$ ENDIF)
= ((j' + 1) × (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`.[20]

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.[21] 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:

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

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

```
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.

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)`.[22] 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)
```

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

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 and unary minus operations, along with a statement asserting a property of subtraction, division, and the less-than ordering.[23]

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

Notice here that the relation `<` is declared as a function with range type `bool` representing the booleans[24] Typechecking the formula `div_test` generates the TCC

```
div_test_TCC1: OBLIGATION
    (FORALL (x:rat, 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.[25]

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

[23] Note the overloading of the name `-`. All names of the same kind within a theory must be unique, with the exception of expression kinds, which need only be unique up to the signature. The signature is enough to distinguish these declarations.

[24] 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.

[25] 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.

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.

```
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 `empty` is an uninterpreted constant 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.[26] 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]`.

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

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
  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 (s: stack, x: t, y: t):
     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.[27]

[27] This is built in to PVS in a different form; again, the development here is for pedagogical purposes. Note the use of ";" to terminate the definition of `antisym`. Semicolons are optional, except in circumstances such as this when the parser needs more information. In this case, the semicolon informs the parser that the operator `<` is a declaration rather than part of the preceding expression.

```
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`.

```
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[28] are visible, including all instances of imported theories that were referenced.[29] 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]
```

[28] With the exception of variable declarations.

[29] 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.

```
    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 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 an `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[t])
       d_order: (total_order?[domain]),
       r_order: (partial_order?[range])] : 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 `well_founded_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.

```
well_founded_induction [t: type,
               (IMPORTING orderings[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 well_founded_induction
```

The `well_founded_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.[30]

```
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 [(IMPORTING functions_) domain, range : TYPE] : THEORY
 BEGIN
  x: VAR int
  square: (injective?[int, nat]) = (LAMBDA (x): x * x)
 END square
```

`square` generates an unprovable TCC 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`, `well_founded_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)}
```

[30] Both well-founded induction and functions are built in to the PVS prelude (theories `wf_induction` and `functions`, respectively) and may not be redefined, hence the name variations introduced here.

```
    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 and are built in to the PVS prelude.[31]

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.[32] 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`.[33] The `MEASURE` function is used to show that the definition terminates, by generating an obligation that the `MEASURE` decreases with each call:

[31] PVS prelude theories may be viewed via the command `M-x view-prelude-theory` (`M-x vpt`). The command `M-x view-prelude-file` (`M-x vpf`) displays the entire prelude.

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

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

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

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

```
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 =
    (# size := size(ns) -1,
       elements := (LAMBDA (j: i|i<size(ns)-1):
                    elements(ns)(j)) #)
  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 value returned by the `pop` operation is a stack in which the size field is one less than that of the input stack and the `elements` array is defined on one fewer indices.[34] Given these definitions, 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 a new theory (and file) called `stack_adt` containing[35]:

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

```
stack_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);
```

[34] A slightly abbreviated form of the `LAMBDA` expression:
`(LAMBDA (i|i<size(ns)-1): elements(ns)(i))`
is also possible. To appreciate the subtlety of this example, note the considerable care necessary when constructing a new record of type `stack` to insure that the domains of `elements` match properly.

[35] Disjointness and inclusion axioms are not explicitly generated, but are built in to the prover (principally through the semantics of induction and the case construct).

- An eta axiom:

```
stack_push_eta: AXIOM
      (FORALL (nonemptystack?_var: (nonemptystack?)):
         push(top(nonemptystack?_var), pop(nonemptystack?_var))
           = nonemptystack?_var);
```

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

```
 stack_pop_push: AXIOM
   (FORALL (push1_var: t, push2_var: stack):
    pop(push(push1_var, push2_var)) = push2_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)));
```

- Functions distributing predicates over the stack base type[36]:

```
every(p: PRED[t])(a: stack): boolean =
  CASES a OF
    empty: TRUE,
    push(push1_var, push2_var):
        p(push1_var) AND every(p)(push2_var)
    ENDCASES

some(p: PRED[t])(a: stack): boolean =
  CASES a OF
    empty: FALSE,
    push(push1_var, push2_var):
        p(push1_var) OR some(p)(push2_var)
    ENDCASES
```

- A subterm function:

```
<<(x: stack, y: stack): boolean =
  CASES y OF
    empty: FALSE,
    push(push1_var, push2_var): x = push2_var OR x << push2_var
  ENDCASES
```

- A well-foundedness axiom:

```
stack_well_founded: AXIOM well_founded?[stack](<<);
```

- A recursive combinator[37]:

[36] These functions are available both in curried form (shown above) and uncurried form.

[37] Another function, `reduce_ordinal`, that reduces a stack to an ordinal rather than a natural number is also generated.

```
reduce_nat(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,
                         reduce_nat(emptystack?_fun,
                                    nonemptystack?_fun)
                           (push2_var))
   ENDCASES
```

The recursive combinator allows the specification of functions such as `length`[38]:

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

- 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_adt_reduce`.
- Another separate parameterized theory, providing a mapping function on stacks, is also generated[39]:

```
stack_adt_map[t: TYPE, t1: TYPE]: THEORY
  BEGIN

  IMPORTING stack_adt

  map(f: [t -> t1])(a: stack[t]): stack[t1] =
    CASES a OF
      empty: empty[t1],
      push(push1_var, push2_var):
          push[t1](f(push1_var), map(f)(push2_var))
    ENDCASES

  END stack_adt_map
```

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?
```

[38] 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.)

[39] The map function is also available in curried form (shown above) and uncurried form.

```
    app(op: term, args: list[term]): app?
  END term
```

Note that the `args` accessor is of type `list[term]`. There are restrictions on the types allowed for 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 generated for abstract datatypes are modified when accessors are of complex types; here, in particular, the induction axioms and recursive combinators generated are modified to handle the list argument.

4 The PVS Proof Checker

The PVS Proof Checker is also referred to as an interactive theorem prover. It is much more automated than the low-level "proof editors" that support some specification notations, but operates under the user's direction and is therefore more controllable than purely automatic theorem provers.

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.
- 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 [Const86], 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.[40] 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

[40] 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.

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

```
{-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 on: (SPLIT)
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)
Postponing 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.

prop.2 :

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

Rule?  (postpone)
Postponing prop.2.

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 `(inst)` command. The first branch of the proof is then easily completed by propositional reasoning. Note that the two quantifier steps, `skolem` and `inst`, 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 `inst`. Here the `(skolem!)` command picks the first "skolemizable" sequent formula and replaces the quantified variables with internally generated constants (containing exclamations). The `(inst?)` 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? (inst -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!)
Skolemizing,
this simplifies to:
pred_calc.2 :

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

Rule? (inst?)
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 `inst` 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 `(inst?)` is a version of `inst` 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 `(inst? - :subst ("x" "X"))`. Both `inst` and `inst?` 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 `(inst + ("x" "X") :copy? T)`. The PVS rule `inst-cp` is a version of the `inst` that automatically copies the quantified formula, and `inst` is the non-copying variant. 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 `inst` 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 `inst` 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 `inst` 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? (inst 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)
Simplifying, rewriting, and recording with 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!)
Skolemizing,
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?  (inst + "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)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of stamps.2.1.1.
```

```
stamps.2.1.2 (TCC):

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

Rule?  (assert)
Simplifying, rewriting, and recording with 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)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
stamps.2.2 :

{-1}   8 + JJ = 5 * five!1 + 3 * three!1
  |-------
[1]   five!1 = 0
{2}   (EXISTS (three: nat), (five: nat): 9 + JJ = 5 * five + 3 * three)

Rule?  (inst + "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]   8 + JJ = 5 * five!1 + 3 * three!1
  |-------
[1]   five!1 = 0
{2}   9 + JJ = 5 * (five!1 - 1) + 3 * (three!1 + 2)

Rule?  (assert)
Simplifying, rewriting, and recording with 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 not mentioned many others. A partial account of some of those we've omitted is given below; a complete, annotated list of prover commands can be found in The PVS Prover Checker Reference Manual [SOR93]. 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.

Proof Checker Pragmatics

The PVS proofs in the tutorial examples reflect a very low level of automation and should be viewed merely as pedagogical exercises. The proof checker actually provides several powerful commands for the advanced user that make it possible to verify large classes of theorems using only a small number of steps. For example, the `grind` command is usually a good way to complete a proof that only requires definition expansion, and arithmetic, equality, and quantifier reasoning. The decision procedure command `assert` is used very frequently

since it does simplification, automatic rewriting, and records the sequent formulas in the decision procedure database. The `inst?` command is the most effective way to automatically instantiate quantifiers of existential strength. The `induct-and-simplify` command is a powerful way to construct proofs by induction. The commands `induct-and-rewrite` and `induct-and-rewrite!` are variants of `induct-and-simplify`. These induction commands are able to automatically complete a fairly large class of induction proofs.

It is not necessary to master all the proof commands in order to use the PVS proof checker effectively. In general, it is advisable to learn the most powerful commands first and only rely on the simpler commands when the powerful ones fail. For example, the initial step in a proof is usually skolemization, and the preferred and most powerful form here is `skosimp*`. Similarly, `induct-and-simplify` or one of its variants should be used to initiate induction proofs.

Typically, the creative choices in a proof are:

1. The induction scheme: One of the above induction commands should be employed here.

2. The case analysis: If the case analysis is not explicit in the propositional structure, then it might be implicit in an embedded `IF-THEN-ELSE` or `CASES` structure in which case the `lift-if` command should be used to bring the case analyses to the surface of the sequent where they can be propositionally simplified. Otherwise, the case analysis has to be supplied explicitly using the `case` command.

3. The quantifier instantiations: The instantiation of antecedent universal and succedent existential quantifiers is done automatically by the `inst?` command. When this fails, the more manual `inst` and `inst-cp` commands should be used.

The `bddsimp` command is the most efficient way to do propositional simplification, but `prop` will do when efficiency is not important. Propositional simplification has to be used with care because it can generate many subgoals that share the same proof. The `flatten` and `split` commands are used to do the propositional simplification more delicately.

User-defined *proof strategies*, similar to the tactics and tacticals of LCF, can be employed by more advanced PVS users. A file containing definitions of basic strategies is distributed with PVS and provides a good introduction to this topic. The PVS Prover Checker Reference Manual [SOR93] can be consulted for additional information on user-defined proof strategies.

Finally, it is helpful to be familiar with the PVS prelude theories, which provide very useful background mathematics, as well as a rich source of examples.

5 Two Hardware Examples

In this final section, we develop two hardware examples that illustrate a more sophisticated use of PVS and suggest the intellectual discipline involved in specifying and proving industrial-strength applications. The pipelined microprocessor and n-bit ripple-carry adder

examples provide an opportunity to touch on modeling issues, specification styles, and hardware proof strategies, as well as a chance to review many of the PVS language and prover features described in earlier sections of this tutorial.[41]

5.1 A Pipelined Microprocessor

We first develop a complete proof of a correctness property of the controller logic of a simple pipelined processor design described at a register-transfer level. The design and the property verified are both based on the processor example given in [BCM+90]. The example has been used as a benchmark for evaluating how well finite state-enumeration based tools, such as model checkers, can handle datapath-oriented circuits with a large number of states by varying the size of the datapath. From the perspective of a theorem prover, the size of the datapath is irrelevant because the specification and proof are independent of the datapath size. As a theorem proving exercise, the challenge is to see if the proof can be done as automatically as a model checker proof.

Informal Description

Figure 11 shows a block diagram of the pipeline design. The processor executes instructions of the form `(opcode src1 src2 dstn)`, i.e., "destination register `dstn` in the register file `REGFILE` becomes some `ALU` function determined by `opcode` of the contents of source registers `src1` and `src2`. Every instruction is executed in three stages (cycles) by the processor:

[41] One point worth noting that may not be apparent in reading these examples is that the process of specification and verification is an iterative one in which proof is used not to certify a completed specification, but as an aid to developing the specification.

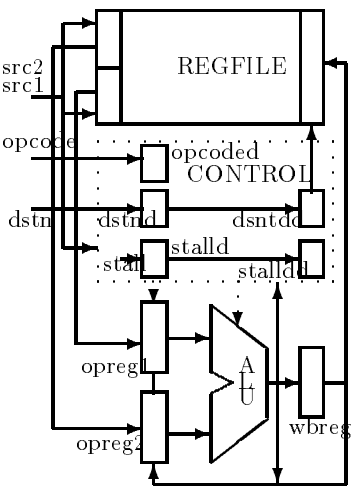

Figure 11: A Pipelined Microprocessor

1. *Read*: Obtain the proper contents of the register file at `src1` and `src2` and clock them into `opreg1` and `opreg2`, respectively.
2. *Compute*: Perform the ALU operation corresponding to the opcode (remembered in `opcoded`) of the instruction and clock the result into `wbreg`.
3. *Write*: Update the register file at the destination register (remembered in `dstndd`) of the instruction with the value in `wbreg`.

The processor uses a three-stage pipeline to simultaneously execute distinct stages of three successive instructions. That is, the read stage of the current instruction is executed along with the compute stage of the previous instruction and the write stage of the previous-to-previous instruction. Since the `REGFILE` is not updated with the results of the previous and previous-to-previous instructions while a read is being performed for the current instruction, the controller "bypasses" `REGFILE`, if necessary, to get the correct values for the read. The processor can abort, i.e., treat as `NOP`, the instruction in the read stage by asserting the `stall` signal true. An instruction is aborted by inhibiting its write stage by remembering the `stall` signal until the write stage via the registers `stalld` and `stalldd`. We verify that an instruction entering the pipeline at any time gets completed correctly, i.e., will write the correct result into the register file, three cycles later, provided the instruction is not aborted.

Formal Specification

PVS specifications consist of a number of files, each of which contains one or more theories. The microprocessor specification is organized into three theories, selected parts of which are shown in Figures 12 and 13. (The complete specification can be found in [SSR95].) The theory `pipe` (Figure 12) contains a specification of the design and a statement of the correctness property to be proved. The theories `signal` and `time`, (Figure 13) imported by `pipe`, declares the types `signal` and `time` used in `pipe`.

The theory `pipe` is parameterized with respect to the types of the register address, data, and the opcode field of the instructions. A theory parameter in PVS can be either a type parameter or a parameter belonging to a particular type, such as `nat`. Since `pipe` does not impose any restriction on its parameters, other than the requirement that they be nonempty, which is stated in the `ASSUMING` part of the theory, one can instantiate them with any type. Every entity declared in a parameterized theory is implicitly parameterized with respect to the parameters of the theory. For example, the type `signal` declared in the parameterized theory `signal` is a parametric type denoting a function that maps `time` (a synonym for `nat`) to the type parameter `T`. (The type `signal` is used to model the wires in our design.) By importing the theory `signal` uninstantiated in `pipe`, we have the freedom to create any desired instances of the type `signal`.

In this example, we use a *functional* style of specification to model register-transfer-level digital hardware in logic. In this style, the inputs to the design and the outputs of every component in the design are modeled as signals. Every signal that is an output of a component is specified as a function of the signals appearing at the inputs to the component.

```
pipe[addr: TYPE, data: TYPE, opcodes: TYPE]: THEORY
  BEGIN
   IMPORTING signal, time

   ASSUMING
    addr_nonempty: ASSUMPTION (EXISTS (a: addr): TRUE)
    data_nonempty: ASSUMPTION (EXISTS (d: data): TRUE)
    opcodes_nonempty: ASSUMPTION (EXISTS (o: opcodes): TRUE)
   ENDASSUMING

   t: VAR time

   %% Signal declarations
      opcode: signal[opcodes]
      src1, src2, dstn: signal[addr]
      stall: signal[bool]
      aluout: signal[data]
      regfile: signal[[addr -> data]]
       ...

    %% Specification of constraints on the signals
      dstnd_ax: AXIOM dstnd(t+1) = dstn(t)
      dstndd_ax: AXIOM dstndd(t+1)= dstnd(t)
      .....

      regfile_ax: AXIOM regfile(t+1) =
                          IF stalldd(t) THEN regfile(t)
                          ELSE regfile(t)
                               WITH [(dstndd(t)) := wbreg(t)]
                          ENDIF
      opreg1_ax: AXIOM opreg1(t+1) =
                     IF src1(t) = dstnd(t) & NOT stalld(t)
                        THEN aluout(t)
                     ELSIF src1(t) = dstndd(t) & NOT stalldd(t)
                           THEN wbreg(t)
                     ELSE regfile(t)(src1(t)) ENDIF
       opreg2_ax: AXIOM ...

       aluop: [opcodes, data, data -> data]
       ALU_ax: AXIOM aluout(t) = aluop(opcoded(t), opreg1(t),
                                       opreg2(t))
      correctness: THEOREM (FORALL t:
        NOT(stall(t)) IMPLIES regfile(t+3)(dstn(t)) =
                     aluop(opcode(t), regfile(t+2)(src1(t)),
                                      regfile(t+2)(src2(t))) )
  END pipe
```

Figure 12: Microprocessor Specification

This style should be contrasted with a *predicative* style, which is commonly used in most HOL applications. In the predicative style every hardware component is specified as a predicate relating the input and output signals of the component and a design is specified as a conjunction of the component predicates, with all the internal signals used to connect the components hidden by existential quantification. A proof of correctness for a predicative style specification usually involves executing a few additional steps at the start of the proof to essentially transform the predictative specification into an equivalent functional style. After that, the proof proceeds similar to that of a proof in a functional specification. The additional proof steps required for a predicative specification essentially

```
signal[val: TYPE]: THEORY
  BEGIN
   signal: TYPE = [time -> val]
  END signal

time: THEORY
  BEGIN
   time: TYPE nat
  END signal
```

Figure 13: Signal Specification

unwind the component predicates using their definitions and then appropriately instantiate the existentially quantified variables. An automatic way of performing this translation is discussed in [SSR95], which illustrates more examples of hardware design verification using PVS.

Returning to our example, the microprocessor specification in `pipe` consists of two parts. The first part declares all the signals used in the design—the inputs to the design and the internal wires that denote the outputs of components. The composite state of `REGFILE`, which is represented as a function from `addr` to `data`, is modeled by the signal `regfile`. The signals are declared as uninterpreted constants of appropriate types. The second part consists of a set of AXIOMs that specify the the values of the signals over time. (To conserve space, we have only shown the specification of a subset of the signals in the design.) For example, the signal value at the output of the register `dstnd` at time `t+1` is defined to be that of its input a cycle earlier. The output of the ALU, which is a combinational component, is defined in terms of the inputs at the same time instant.

In PVS, we can use a descriptive style of definition, as illustrated in this example, by selectively introducing properties of the constants declared in a theory as AXIOMs. Alternatively, we can use the definitional forms provided by the language to define the constants. An advantage of using the definitions is that a specification is guaranteed to be consistent. A disadvantage is that the resulting specification may be overly specific (i.e., overspecified). An advantage of the descriptive style is that it gives better control over the degree to which an entity is defined For example, we could have specified `dstnd` prescriptively, using the conventional function definition mechanism of PVS, which would have forced us to specify the value of the signal at time `t = 0` to ensure that the function is total. In the descriptive style used, we have left the value of the signal at `0` unspecified.

In the present example, the specifications of the signals `opreg1` and `opreg2` are the most interesting of all. They have to check for any register collisions that might exist between the instruction in the read stage and the instructions in the later stages and bypass reading from the register file in case of collisions. The `regfile` signal specification is recursive since the register file state remains the same as its previous state except, possibly, at a single register location. The `WITH` expression is an abbreviation for the result of updating a function at a given point in the domain value with a new value. Note that the function `aluop` that denotes the operation ALU performs for a given `opcode` is left completely unspecified since it is irrelevant to the controller logic.

The theorem (`correctness`) to be proved states a correctness property about the execution of the instruction that enters the pipeline at `t`, provided the instruction is not aborted, i.e., `stall(t)` is not true. The equation in the conclusion of the implication compares the actual value (left hand side) in the destination register three cycles later, when the result of the instruction would be in place, with the expected value. The expected value is the result of applying the `aluop` corresponding to the opcode of the instruction to the values at the source field registers in the register file at `t+2`. We use the state of the register file at `t+2` rather than `t` to allow for the results of the two previous instructions in the pipeline to be completed.

Proof of Correctness

Once the specification is complete, the next step is to typecheck the file, which parses and checks for semantic errors, such as undeclared names and ambiguous types. As we have already seen, typechecking may build new files or internal structures such as *type correctness conditions* (*TCCs*) that represent *proof obligations* that must be discharged before the `pipe` theory can be considered typechecked. The typechecker does not generate any TCCs in the present example. If, for example, one of the assumptions, say for `addr`, in the `ASSUMING` part of the theory was missing, the typechecker would generate the following TCC to show that the `addr` type is nonempty. The declaration of the signal `src1` forces generation of this TCC because a function is nonexistent if its range is empty.

```
% Existence TCC generated (line 17) for src1: signal[addr]
% May need to add an assuming clause to prove this.
  % unproved
src1_TCC1: OBLIGATION (EXISTS (x1: signal[addr]): TRUE);
```

By way of review, the basic objective of developing a proof in PVS as in other subgoal-directed proof checkers (e.g., HOL), is to generate a *proof tree* in which all of the leaves are trivially true. The nodes of the proof tree are sequents, and in the prover you are always 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.

The primitive inference steps in PVS are a lot more powerful than in HOL; it is not necessary to build complex tactics to handle tedious lower level proofs in PVS. A knowledgeable PVS user can typically get proofs to go through mostly automatically by making a few critical decisions at the start of the proof. However, as noted previously, PVS does provide the user with the equivalent of HOL's tacticals, called *strategies*, and other features to control the desired level of automation in a proof.

The proof of the microprocessor property shown below follows a certain general pattern that works successfully for most hardware proofs. This general proof pattern, variants of which have been used in other verification exercises [KSK93, ALW93], consists of the following sequence of general proof tasks.

Quantifier elimination: Since the decision procedures work on ground formulas, the user must eliminate the relevant universal quantifiers by skolemization or selecting variables on which to induct and existential quantifiers by suitable instantiation.

Unfolding definitions: The user may have to simplify selected expressions and defined function symbols in the goal by rewriting using definitions, axioms or lemmas. The user may also have to decide the level to which the function symbols have to rewritten.

Case analysis: The user may have to split the proof based on selected boolean expressions in the current goal and simplify the resulting goals further.

Each of the above tasks can be accomplished automatically using a short sequence of primitive PVS proof commands. The complete proof of the theorem is shown below. Selected parts of the proof session are reproduced below as we describe the proof.

```
1: ( then* (skosimp)
2:         (auto-rewrite-theory ``pipe'' :always? t)
3:         (repeat (do-rewrite))
4:         (apply (then* (repeat (lift-if))
5:                       (bddsimp)
6:                       (assert))))
```

In the proof, the names of strategies are shown in *italics* and the primitive inference steps in `type-writer font`. (We have numbered the lines in the proof for reference.) `Then*` applies the first command in the list that follows to the current goal; the rest of the commands in the list are then applied to each of the subgoals generated by the first command application. The `apply` command used in line 5 makes the application of a compound proof step implemented by a strategy behave as an atomic step.

The first goal in the proof session is shown below. It 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.

```
correctness :

  |-------
{1}   (FORALL t: NOT (stall(t))
                 IMPLIES regfile(t + 3)(dstn(t)) =
                    aluop(opcode(t), regfile(t + 2)(src1(t)),
                          regfile(t + 2)(src2(t))))
```

The quantifier elimination task of the proof is accomplished by the command `skosimp`, which skolemizes all the universally quantified variables in a formula and flattens the sequent resulting in the following goal. Note that `stall(t!1)` has been moved to the succedent in the sequent because PVS displays every atomic formula in its positive form.

```
Rule? (skosimp)
Skolemizing and flattening, this simplifies to:
correctness :

  |-------
{1}   (stall(t!1))
{2}   regfile(t!1 + 3)(dstn(t!1))
        =
        aluop(opcode(t!1), regfile(t!1 + 2)(src1(t!1)),
              regfile(t!1 + 2)(src2(t!1)))
```

The next task—unfolding definitions—is performed by the commands in lines 2 through 3. PVS provides a number of ways of unfolding definitions ranging from unfolding one step at a time to automatic rewriting that performs unfolding in a brute-force fashion. Brute-force rewriting usually results in larger expressions than controlled unfolding and, hence, potentially larger number of cases to consider. If a system provides automatic and efficient rewriting and case analysis facilities, then the automatic approach is viable, as illustrated here. In PVS automatic rewriting is performed by first entering the definitions and AXIOMs to be used for unfolding as rewrite rules. Once entered, the commands that perform rewriting as part of their repertoire, such as `do-rewrite` and `assert`, repeatedly apply the rewrite rules until none of the rules is applicable. To control the size of the expression resulting from rewriting and the potential for looping, the rewriter uses the following restriction for stopping a rewrite: If the right-hand-side of a rewrite is a conditional expression, then the rule is applied only if the condition simplifies to true or false.

Here our aim is to unfold every signal in the sequent so that every signal expression contains only the start time `t!1`. So, we make a rewrite rule out of every AXIOM in the theory `pipe` by means of the command `auto-rewrite-theory` on line 2. We also force an over-ride of the default restriction for stopping rewriting by setting the tag[42] `always?` to true in the `auto-rewrite-theory` command and embed `do-rewrite` inside a `repeat` loop to force maximum rewriting. In the present example, the rewriting is guaranteed to terminate because every feedback loop is cut by a sequential component.

At the end of automatic rewriting, the succedent we are trying to prove is in the form of an equation on two deeply nested conditional expressions as shown below in an abbreviated fashion. The various cases in conditional expression shown above arise as a result of the different possible conflicts between instructions in the pipeline. The equation we are trying to prove contains two distinct, but equivalent conditional expressions, as in `IF a THEN b ELSE c ENDIF = IF NOT a THEN c ELSE b ENDIF`, that can only be proved equal by performing a case-split on one or more of the conditions. While `assert` simplifies the leaves of a conditional expression assuming every condition along the path to the leaves holds, it does not split propositions. One way to perform the case-splitting task automatically is to "lift" all the `IF-THEN-ELSE`s to the top so that the equation is transformed into a propositional formula with unconditional equalities as atomic predicates. After performing such a lifting, we can try to reduce the resulting proposition to true using the propositional

[42] Tags are one of the ways in which PVS permits the user to modify the functionality of proof commands.

simplification command `bddsimp`. If `bddsimp` does not simplify the proposition to true, then it is most likely the case that equations at one or more of the leaves of the proposition need to be further simplified, e.g., by `assert`, using the conditions along the path. If the propositional formula does not reduce to true or false, `bddsimp` produces a set of subgoals to be proved. In the present case, each of these goals can be discharged by `assert`. The compound proof step appearing on lines 4 through 6 of the proof accomplishes the case-splitting task.

```
correctness :

  |-------
[1]   (stall(t!1))
{2}   aluop(opcode(t!1),
            IF src1(t!1) = dstnd(t!1) & NOT stalld(t!1)
              THEN aluop(opcoded(t!1), opreg1(t!1), opreg2(t!1))
            ELSIF src1(t!1) = dstndd(t!1) & NOT stalldd(t!1)
            THEN wbreg(t!1)
            ELSE regfile(t!1)(src1(t!1)) ENDIF,
            ....
            ENDIF)
       = aluop(opcode(t!1),
              IF stalld(t!1) THEN IF stalldd(t!1) THEN regfile(t!1)
                ELSE regfile(t!1) WITH [(dstndd(t!1)) := wbreg(t!1)]
                ENDIF
              ELSE ...
              ENDIF(src1(t!1)),
              IF stalld(t!1) THEN IF stalldd(t!1) THEN regfile(t!1)
                ELSE ... ENDIF
              ELSE ...
              ENDIF(src2(t!1)))
```

We have found that the sequence of steps shown above works successfully for proving safety properties of finite state machines that relate states of the machine that are finite distance apart. If the strategy does not succeed, the most likely cause is that either the property is not true or that a certain property about some of the functions in the specification unknown to the prover needs to be proved as a lemma. In either case, the unproven goals remaining at the end of the proof provide information about the probable cause.

5.2 An N-bit Ripple-Carry Adder

The second example we consider is the verification of a parametrized N-bit ripple-carry adder circuit. The theory `adder`, shown in Figure 14, specifies a ripple-carry adder circuit and a statement of correctness for the circuit.

The theory is parameterized with respect to the length of the bit-vectors. It imports the theories (not shown here) `full_adder`, which contains a specification of a full adder circuit (`fa_cout` and `fa_sum`), and `bv`, which specifies the bit-vector type (`bvec[N]`) and functions. An N-bit bit-vector is represented as an array, i.e., a function, from the the type

```
adder[N: posnat] : THEORY
BEGIN
  IMPORTING bv[N], full_adder

  n: VAR below[N]
  bv, bv1, bv2: VAR bvec
  cin: VAR bool

  nth_cin(n, cin, bv1, bv2): RECURSIVE bool =
      IF n = 0 THEN cin
      ELSE fa_cout(nth_cin(n - 1, cin, bv1, bv2), bv1(n - 1), bv2(n - 1))
      ENDIF
    MEASURE n

  bv_sum(cin, bv1, bv2): bvec =
    (LAMBDA n: fa_sum(bv1(n), bv2(n), nth_cin(n, cin, bv1, bv2)))

  bv_cout(n, cin, bv1, bv2): bool =
    fa_cout(nth_cin(n, cin, bv1, bv2), bv1(n), bv2(n))

  adder_correct_n: LEMMA
      bvec2nat_rec(n, bv1) + bvec2nat_rec(n, bv2) + bool2bit(cin)
        = exp2(n + 1) * bool2bit(bv_cout(n, cin, bv1, bv2))
          + bvec2nat_rec(n, bv_sum(cin, bv1, bv2))

  adder_correct: THEOREM
      bvec2nat(bv1) + bvec2nat(bv2) + bool2bit(cin)
        = exp2(N) * bool2bit(bv_cout(N - 1, cin, bv1, bv2))
          + bvec2nat(bv_sum(cin, bv1, bv2))
END adder
```

Figure 14: Adder Specification

`below[N]`, a subtype of `nat` ranging from `0` through `N-1`, to `bool`; the index `0` denotes the least significant bit. Note that the parameter `N` is constrained to be a `posnat` since we do not permit bit vectors of length `0`. The `adder` theory contains several declarations including a set of initial variable declarations.

The carry bit that ripples through the full adder is specified recursively by means of the function `nth_cin`. The function `bv_cout` and `bv_sum` define the carry output and the bit-vector sum of the adder, respectively. The theorem `adder_correct` expresses the conventional correctness statement of an adder circuit using `bvec2nat`, which returns the natural number equivalent of an N-bit bit-vector. Note that variables that are left free in a formula are assumed to be universally quantified. We state and prove a more general lemma `adder_correct_rec` of which `adder_correct` is an instance. For a given `n < N`, `bvec2nat_rec` returns the natural number equivalent of the least significant n-bits of a given bit-vector and `bool2bit` converts the boolean constants `TRUE` and `FALSE` into the natural numbers `1` and `0`, respectively.

Typechecking

The typechecker generates several TCCs (shown in Figure 15 below) for `adder`.

```
% Subtype TCC generated (line 13) for n - 1
  % unproved
nth_cin_TCC1: OBLIGATION (FORALL n: NOT n = 0 IMPLIES n - 1 >= 0 AND n - 1 < N)

% Subtype TCC generated (line 31) for N - 1
  % unproved
adder_correct_TCC1: OBLIGATION N - 1 >= 0
```

Figure 15: TCCs for Theory `adder`

The first TCC is due to the fact that the first argument to `nth_cin` is of type `below[N]`, but the type of the argument (`n-1`) in the recursive call to `nth_cin` is integer, since `below[N]` 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. A TCC identical to this one is generated for each of the two other occurrences of the expression `n-1` because `bv1` and `bv2` also expect arguments of type `below[N]`. These TCCs are not retained because they are subsumed by the first one.

The second TCC is generated by the expression `N-1` in the definition of the theorem `adder_correct` because the first argument to `bv_cout` is expected to be the subtype `below[N]`.

There is yet another TCC that is internally generated by PVS but is not even included in the TCCs file because it can be discharged trivially by the typechecker, which calls the prover to perform simple normalizations of expressions. This TCC is generated to ensure that the recursive definition of `nth_cin` 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. As discussed earlier, the measure function is used to show that recursive definitions are total by requiring the measure to decrease with each recursive call. For the definition of `nth_cin`, this entails showing `n-1 < n`, which the typechecker trivially deduces.

In the present case, all the remaining TCCs are simple, and in fact can be discharged automatically by using the `typecheck-prove` command, which attempts to prove all TCCs that have been generated using a predefined proof strategy called `tcc`.

Proof of Adder_correct_n

The proof of the lemma uses the same core strategy as in the microprocessor proof except for the quantifier elimination step. Since the specification is recursive in the length of the bit-vector, we need to perform induction on the variable `n`. As we've seen in earlier proofs, the user invokes an inductive proof in PVS by means of the command `induct` with the variable to induct on (`n`) and the induction scheme to be used (`below_induction[N]`) as arguments. The induction used in this case is defined in the PVS prelude and is parameterized, as is the type `below[N]`, with respect to the upper limit of the subrange.

This command generates two subgoals: the subgoal corresponding to the base case, which is the first goal presented to prove, is shown in Figure 16.

The goal corresponding to the inductive case is shown below.

```
adder_correct.1 :

  |-------
1   (N > 0
         IMPLIES
         (FORALL
          (bv1: bvec[N], bv2: bvec[N], cin: bool):
            bvec2nat_rec(0, bv1) + bvec2nat_rec(0, bv2)
              + bool2bit(cin)
              = exp2(0 + 1) * bool2bit(bv_cout(0, cin, bv1, bv2))
                  + bvec2nat_rec(0, bv_sum(cin, bv1, bv2))))
```

Figure 16: Base Step

```
The remaining siblings are:
adder_correct_n.2 :

  |-------
{1}   (FORALL (r: below[N]):
         r < N - 1
           AND (FORALL (bv1, bv2: bvec[N]), (cin: bool):
                  bvec2nat_rec(r, bv1) + bvec2nat_rec(r, bv2)
                    + bool2bit(cin)
                    = exp2(r + 1) * bool2bit(bv_cout(r, cin, bv1, bv2))
                        + bvec2nat_rec(r, bv_sum(cin, bv1, bv2)))
           IMPLIES (FORALL (bv1, bv2: bvec[N]), (cin: bool):
                      bvec2nat_rec(r + 1, bv1)
                        + bvec2nat_rec(r + 1, bv2)
                        + bool2bit(cin)
                        = exp2(r + 1 + 1)
                            * bool2bit(bv_cout(r + 1, cin, bv1, bv2))
                            +
                            bvec2nat_rec(r + 1,
                                         bv_sum(cin, bv1, bv2))))
```

Figure 17: Inductive Step

The base and the inductive steps can be proved automatically using essentially the same strategy used in the microprocessor proof. A complete proof of `adder_correct_n` is shown in Figure 17.

```
 1: ( spread (induct ``n'' 1 ``below_induction[N]'')
 2:   ( ( then*  (skosimp*)
 3:              (auto-rewrite-defs :always? t)
 4:              (do-rewrite)
 5:              ( repeat (lift-if))
 6:              ( apply ( then* (bddsimp)(assert))))
 7:     ( then* (skosimp*)
 8:             (inst?)
 9:             (auto-rewrite-defs :always? t)
10:             (do-rewrite)
11:             ( repeat (lift-if))
12:             ( apply ( then* (bddsimp)(assert))))))
```

The strategy *spread* used on line 1 applies the first proof step (`induct`) and then applies the i^{th} element of the list of commands that follow to the i^{th} subgoal resulting from the

application of the first prof step. Thus, the proof steps listed on lines 2 through 6 prove the base case of induction, the steps on lines 7 through 12 prove the inductive case, and the proof step on line 13 takes care of the third TCC subgoal.

We consider the base case first. The `induct` command has already instantiated the variable `n` to `0`. The remaining variables are skolemized away by `skosimp*`. To unfold the definitions in the resulting goal, we use the command `auto-rewrite-defs`, which makes rewrite rules out of the definition of every function either directly or indirectly used in the given formula. The rest of the proof proceeds exactly as for the microprocessor.

The proof of the inductive step follows exactly the same pattern except that we need to instantiate the induction hypothesis and use it in the process of unfolding and case-analysis. PVS provides a command `inst?` that tries to find instantiations for existential-strength variables in a formula by searching for possible matches between terms involving these variables with ground terms inside formulas in the rest of the sequent. This command finds the desired instantiations in the present case. The rest of the proof proceeds as in the basis case.

Since the inductive proof pattern shown above is applicable to any iteratively generated hardware designs, we have packaged it into a general proof strategy called `name-induct-and-bddrewrite`. The strategy is parameterized with respect to an induction scheme and the set of rewrite rules to be used for unfolding. We have used the strategy to prove an N-bit ALU [Can94] that executes 12 microoperations by cascading N 1-bit ALU slices.

6 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 73).*

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.*

Part III

PVS Reference

Reference to PVS Version 2.0β

PVS Files

PVS Files

foo.pvs	Specification file (contains theories)
foo.bin	Binary form of the typed specification file
foo.prf	Saved proofs for *foo*.pvs
.pvscontext	Context information
foo-alltt.tex	Alltt-printed version of *foo*
foo.tex	LaTeX-printed version of *foo*
pvs-files.tex	LaTeX file generated for testing Alltt and LaTeX-printed files

LaTeX Substitution Files

LaTeX Substitutions for file `foo.pvs` may come from any of the following files.

File name	Location
`foo.sub`	the directory of the current context
`pvs-tex.sub`	the directory of the current context
`pvs-tex.sub`	user's home directory
`pvs-tex.sub`	the main PVS directory

Examples of substitution entries—numbers refer to the number of arguments; thus the third entry translates `f2[3,G]` (to G_3^f) but not `f2[int]`, and the last entry translates, e.g., `f4(G)(1,n)` (to $\sum_{i=1}^{n} G(i,1)$). Length is an estimation of the size of the translation, ignoring the size of the arguments.

Identifier	Type	Length	Substitution
`THEORY`	`key`	9	`{\large\bf Theory}`
`f1`	`id`	3	`{\rm bar}`
`f2`	`id[2]`	2	`{#2_{#1}^{f}}`
`f3`	`2`	2	`{#1^#2}`
`f4`	`(1 2)`	3	`{\sum_{i=#2}^{#3}#1(i,#2)}`

PVS Language Summary

Theories

```
function_properties [D, R: TYPE]: THEORY
 BEGIN
  f, g: VAR [D -> R]
  x, x1, x2: VAR D
  y: VAR R

  injective?(f): bool = (FORALL x1, x2: (f(x1) = f(x2) => (x1 = x2)))

  surjective?(f): bool = (FORALL y: (EXISTS x: f(x) = y))
 END function_properties

finite[t: TYPE]: THEORY
 BEGIN
  IMPORTING function_properties
  is_finite_type: bool = (EXISTS (n:nat), (f:[upto[n] -> t]): surjective?(f))

  is_finite_type_alt: LEMMA
    is_finite_type IFF (EXISTS (n:nat), (g:[t -> upto[n]]): injective?(g))
 END finite

best_choice[t: TYPE, meas: TYPE FROM real]: THEORY
 BEGIN
  ASSUMING
    IMPORTING finite[t]
    finite: ASSUMPTION is_finite_type[t]
  ENDASSUMING

  best: [[t -> meas], setof[t] -> t]

  f: VAR [t -> meas]
  s: VAR setof[t]

  best_ax: AXIOM
     nonempty?(s) => member(best(f, s), s)
       AND (FORALL (x: t): member(x, s) => f(x) <= f(best(f, s)))
 END best_choice
```

Lexical Rules

Comments start with % and go to the end of the line

Identifiers are composed of letters, digits, question mark,and underscores; they must begin with a letter and are case-sensitive.

Numbers are composed of digits—no floating point numbers.

Strings are enclosed in double quotes "

Reserved Words

Reserved words are not case sensitive.

AND	CONTAINING	FALSE	LEMMA	SUBTYPE_OF
ANDTHEN	CONVERSION	FORALL	LET	TABLE
ARRAY	COROLLARY	FORMULA	LIBRARY	THEN
ASSUMING	DATATYPE	FROM	MEASURE	THEOREM
ASSUMPTION	ELSE	FUNCTION	NONEMPTY_TYPE	THEORY
AXIOM	ELSIF	HAS_TYPE	NOT	TRUE
BEGIN	END	IF	O	TYPE
BUT	ENDASSUMING	IFF	OBLIGATION	TYPE+
BY	ENDCASES	IMPLIES	OF	VAR
CASES	ENDCOND	IMPORTING	OR	WHEN
CHALLENGE	ENDIF	IN	ORELSE	WHERE
CLAIM	ENDTABLE	INDUCTIVE	POSTULATE	WITH
CLOSURE	EXISTS	JUDGEMENT	PROPOSITION	XOR
COND	EXPORTING	LAMBDA	RECURSIVE	
CONJECTURE	FACT	LAW	SUBLEMMA	

Special Symbols

!!	$	(	*	-	/=	:=	<=	==	>>	[]	]|	|)	|[
#	$$	(#	**	->	/\	;	<=>	==>	@	[|	^	|-	|]
##	%	(:	+	.	:	<	<>	=>	@@	[||]	^^	|->	||
#)	&	(|	++	/	:)	<-	<|	>	[	\/	{	|=	}
#]	&&	)	,	//	::	<<	=	>=	[#	]	|	|>	

Type Declarations

- Uninterpreted types
  - `foo: TYPE`
  - `bar: NONEMPTY_TYPE  % same as TYPE+`
  - `some_nums: NONEMPTY_TYPE FROM number`
- Subtypes
  - `nat_to_10: TYPE = {x:nat | x <= 10}`
  - `posint: TYPE = {x:integer | x > 0} CONTAINING 1`
  - `ptype: TYPE = (pred?)  % same as {x | pred?(x)}`
  - `rtype: TYPE = {x, y: nat | x < y} % subtype of [nat, nat]`
- Function types
  - `intf: TYPE = FUNCTION[int, int -> int]`
  - `altf: TYPE = [int, int -> int]     % same as above`
  - `inta: TYPE = ARRAY[int,int -> int] % same as above`

- Tuple Types
  - `tuptype: TYPE = [int, bool, [int -> int]]`
- Record types
  - ```
    stack: TYPE = [# pointer: nat,
                     astack: [nat -> t] #]
    ```
- Dependent Types
  - `pfun: TYPE = [# dom: predicate[t1], pfn:[(dom)->t2] #]`
  - `date: TYPE = [y,m:nat, {d:nat | d <= days(m,y)}]`
  - `tmod: TYPE = [n,m:int -> {x:nat | x < m}]`
- Enumeration types
  - `color: TYPE = {red, green, blue}`
- Datatypes
  - ```
    list[t:TYPE] : DATATYPE
          BEGIN
            null: null?
            cons (car: t, cdr :list) :cons?
          END list
    ```

Libraries, Importings, Exportings, and Theory Abbreviations

- `fsets: LIBRARY = "/homes/pvs/lib/finite_sets"`
- `IMPORTING orderings[int], set[foo[nat]], fsets@finite_sets[nat]`
- `EXPORTING foo, bar WITH set[foo]`
- `pset: THEORY = sets[list[nat]]`

Constants and Recursive Definitions

- `some_int: int`
- `max: int = 10`
- ```
  abs: [int -> nat] =
       (LAMBDA x: IF x < 0 THEN -x ELSE x ENDIF)
  ```
- `abs(x:int): nat = IF x < 0 THEN -x ELSE x ENDIF`
- `sum(f,x,y): int` `% f,x,y prev declared VAR`
- `sum(f,(x,y:int)): int` `% f prev declared VAR`
  ```

- fac(n): RECURSIVE nat =
    (IF n = 0 THEN 1 ELSE n*fac(n-1) ENDIF)
  MEASURE (LAMBDA n: n)
- length(l:list): RECURSIVE nat =
    CASES l OF
      null: 0,
      cons(x, y): length(y) + 1
    ENDCASES
  MEASURE l BY << % Subterm measure

Variable Declarations

- x, y, z: VAR int
- f: VAR [int -> [int -> int]]

Formula Declarations

- transitive: AXIOM x < y AND y < z IMPLIES x < z
- nonzero_fac: THEOREM fac(n) /= 0
- poset: ASSUMPTION poset?(T,<=) % Only in ASSUMINGs

Judgements

- JUDGEMENT {x :int | x > 10} SUBTYPE_OF posint
- JUDGEMENT c HAS_TYPE (even?)
- JUDGEMENT +, -, * HAS_TYPE [(even?), (even?) -> (even?)]

Conversions

- C: [int -> bool] = (LAMBDA (i:int): i=0)
  CONVERSION C
  foo: FORMULA d + 1 % ≡ foo: FORMULA C(d + 1)
- state: TYPE
  K: [int -> [state -> int]] = (LAMBDA i: (LAMBDA s: i))
  f: [[state -> int] -> [state -> int]]
  x: [state -> int]
  B: [[state -> int] -> bool] = (LAMBDA si: FORALL i: si(i))
  CONVERSION K, B
  bar: LEMMA f(x+1) % ≡ bar: LEMMA B(f(LAMBDA s: x(s)+1))

Expressions

- Equality — (`=`, `/=`)
  Defined for any type; both sides must be the same type. With boolean, `=` is treated as `IFF`.

  ```
  ○ x * y = 4
  ○ true /= 1            % Illegal
  ```

- Arithmetic — (`+`, `-`, `*`, `/`, `<`, `<=`, `>`, `>=`, `0`, `1`, ...)

  ```
  ○ ((x + 1) * x) / 2 < x * x
  ```

- Lists and Strings

  ```
  ○ (: 1, 2 :)  % ≡ cons(1, cons(2, null))
  ○ "A string"  % A finite sequence of characters
  ```

- Logical — (`true`, `false`, `AND`, `&`, `OR`, `IMPLIES`, `=>`, `WHEN`, `NOT`, `IFF`, `<=>`, `FORALL`, `ALL`, `EXISTS`, `SOME`)

  ```
  ○ (FORALL e: (EXISTS d:
        abs(f(x) - f(y)) < d) IMPLIES abs(x - y) < e)
  ```

- `IF-THEN-ELSE` — The `THEN` and `ELSE` parts must have compatible types.

  ```
  ○ IF x=0 THEN 1 ELSIF y=0 THEN 2 ELSE y/x ENDIF
  ```

- `CASES` — Pattern matching on datatypes.

  ```
  ○ CASES x OF
      cons(x,y): append(reverse(y), cons(x, null))
      ELSE null
    ENDCASES
  ```

- `COND` — generates coverage and disjointness TCCs

  ```
  ○ COND m = n -> m,
         m > n -> gcd(m-n, n),
         m < n -> gcd(m, n-m)
    ENDCOND
  ○ Same as above, but no coverage TCC
       COND m = n -> m,
         m > n -> gcd(m-n, n),
         ELSE -> gcd(m, n-m)
    ENDCOND
  ```

- Function application, lambda-abstraction & function update
  - `f(1,2)(0)`
  - `(lambda x: x + 1)`
  - `f WITH [(0) := 1, (1) := 0]`
  - `foo ! (x:int): e  % ≡ foo(LAMBDA (x: int): e)`
- Set expressions
  - `{x: int | x < 10}  % same as (LAMBDA (x: int): x < 10)`
- Record construction, field selection & record update
  - `(# pointer := 1, astack := (LAMBDA x: 0) #)`
  - `astack(r)  % r.astack NOT allowed`
  - `r WITH [pointer := 2, (astack)(1) := 1]`
- Tuple construction, projection, and update
  - `(1, true, (LAMBDA (x:int) x + 37))`
  - `proj_3(tup)`
  - `tup WITH [2 := false]`
- LET & WHERE
  - `LET x = 2, y:nat = x*x IN f(x,y)  % ≡ f(2,4)`
  - `f(x,y) WHERE x = 2, y:nat = x*x  % same`
  - `LET (x, y, z) = t IN x + y * z  % same as next line`
  - `LET x=PROJ_1(t), y=PROJ_2(t), z=PROJ_3(t) IN x + y * z`
- Coercion — Coercion indicates the expected type to the typechecker to resolve ambiguity.
  - `a + b:natural`
  - `(LAMBDA n -> nat: n - m)  % LAMBDA coercion`
- Names — If `foo` is declared in theory `bar`, then the following are allowable references (the first two may be ambiguous).
  - `foo`
  - `foo[int]`
  - `bar[int].foo`

PVS Emacs Commands

The commands which appear below are given with their abbreviations and keybindings, if any. For example, the command Prove with aliases `pr` and `C-c p` indicates that the parse command can be invoked by the Emacs extended commands `M-x prove` or `M-x pr`, or the key binding `C-c p`.

Entering and Exiting PVS

To enter PVS just `cd` to a working directory (*PVS context*) and type `pvs`.

suspend-pvs	`C-x C-z`
exit-pvs	`C-x C-c`

Getting Help

help-pvs	`C-c h`
help-pvs-language	`C-c C-h l`
help-pvs-prover	`C-c C-h p`
help-pvs-prover-command	`C-c C-h c`
help-pvs-prover-strategy	`C-c C-h s`
help-pvs-prover-emacs	`C-c C-h e`

Editing PVS Files

forward-theory	`M-}`
backward-theory	`M-{`
find-unbalanced-pvs	`C-c ]`
comment-region	`C-c ;`

Parsing and Typechecking

parse	`M-x pa`
typecheck	`M-x tc, C-c t`
typecheck-importchain	`M-x tci`
typecheck-prove	`M-x tcp`
typecheck-prove-importchain	`M-x tcpi`

Prover Invocation Commands

prove	`M-x pr, C-c p`
x-prove	`M-x xpr, C-c C-p x`
step-proof	`M-x prs, C-c C-p s`
x-step-proof	`M-x xsp, C-c C-p X`
redo-proof	`M-x prr, C-c C-p r`
prove-theory	`M-x prt, C-c C-p t`
prove-pvs-file	`M-x prf, C-c C-p f`
prove-importchain	`M-x pri, C-c C-p i`
prove-proofchain	`M-x prp, C-c C-p p`

Proof Editing Commands

edit-proof, show-proof	
install-proof	`C-c C-i`
revert-proof	
remove-proof	
show-proof-file	
show-orphaned-proofs	
show-proofs-theory	
show-proofs-pvs-file	
show-proofs-importchain	
install-pvs-proof-file	
load-pvs-strategies	
set-print-depth	
set-print-length	
set-rewrite-depth	
set-rewrite-length	

Proof Information Commands

show-current-proof
explain-tcc
show-last-proof
ancestry
siblings
show-hidden-formulas
show-auto-rewrites
show-expanded-sequent
show-skolem-constants

Adding and Modifying Declarations

add-declaration
modify-declaration

Prettyprinting Commands

prettyprint-theory	`M-x ppt, C-c C-q t`
prettyprint-pvs-file	`M-x ppf, C-c C-q f`
prettyprint-declaration	`M-x ppd, C-c C-q d`
prettyprint-region	`M-x ppr, C-c C-q r`

Viewing TCCs

prettyprint-expanded	`M-x ppe, C-c C-q e`
show-tccs	`M-x tccs, C-c C-q s`

PVS File and Theory Commands

find-pvs-file	`M-x ff, C-c C-f`
find-theory	`M-x ft`
view-prelude-file	`M-x vpf`
view-prelude-theory	`M-x vpt`
new-pvs-file	`M-x nf`
new-theory	`M-x nt`
import-pvs-file	`M-x imf`
import-theory	`M-x imt`
delete-pvs-file	`M-x df`
delete-theory	`M-x dt`
save-pvs-file	`C-x C-s`
save-some-pvs-files	`M-x ssf`
smail-pvs-files	
rmail-pvs-files	
dump-pvs-files	
undump-pvs-files	

Printing Commands

pvs-print-buffer	
pvs-print-region	
print-theory	`M-x ptt`
print-pvs-file	`M-x ptf`
print-importchain	`M-x pti`
alltt-theory	`M-x alt, C-c C-a t`
alltt-pvs-file	`M-x alf, C-c C-a f`
alltt-importchain	`M-x ali, C-c C-a i`
alltt-proof	`M-x alp, C-c C-a p`
latex-theory	`M-x ltt, C-c C-l t`
latex-pvs-file	`M-x ltf, C-c C-l f`
latex-importchain	`M-x lti, C-c C-l i`
latex-proof	`M-x ltp, C-c C-l p`
latex-theory-view	`M-x ltv, C-c C-l v`
latex-set-linelength	

Display Commands

x-theory-hierarchy	
x-show-proof	
x-show-current-proof	
x-prover-commands	

Context Commands

list-pvs-files	`M-x lf`
list-theories	`M-x lt`
change-context	`M-x cc`
save-context	`M-x sc`
pvs-remove-bin-files	
pvs-dont-write-bin-files	
pvs-do-write-bin-files	
context-path	`M-x cp`

Library Commands

load-prelude-library	
remove-prelude-library	

Browsing Commands

show-declaration	`M-.`
find-declaration	`M-,`
whereis-declaration-used	`M-;`
list-declarations	`M-:`

Status Commands

status-theory	`M-x stt, C-c C-s t`
status-pvs-file	`M-x stf, C-c C-s f`
status-importchain	`M-x sti, C-c C-s i`
status-importbychain	`M-x stb, C-c C-s b`
status-proof	`M-x sp`
status-proof-theory	`M-x spt`
status-proof-pvs-file	`M-x spf`
status-proof-importchain	`M-x spi`
status-proofchain	`M-x spc`
status-proofchain-theory	`M-x spct`
status-proofchain-pvs-file	`M-x spcf`
status-proofchain-importchain	`M-x spci`

Environment Commands

whereis-pvs	
pvs-version	
pvs-mode	
pvs-log	
status-display	
pvs-status	
remove-popup-buffer	`C-z 1`
pvs	
pvs-load-patches	
pvs-interrupt-subjob	`C-c C-c`
reset-pvs	`C-z z`

PVS Prover Commands

Prover commands are entered in the `*pvs*` buffer at the `Rule?` prompt. Commands are interpreted by Lisp, so must be surrounded by parentheses, and arguments are separated by whitespace (`Space`, `Tab`, or `Return`). Some commands require PVS names or expressions; these must be surrounded by double quotes ("). `Return` enters the command, unless parentheses or strings are unbalanced.

The arguments are shown in *emphasized* font. Optional arguments follow the keyword &OPT and may be omitted. They may be provided in the order listed, or followed by a keyword whose name is derived form the argument name preceded by a colon, *e.g.,* `(expand "foo" :beta-reduce t)`. An *&rest* keyword indicates that one or more of the following argument may be provided, and may also be given in keyword form.

Help

(help &OPT *name[*]*)

Control

(fail)
(postpone &OPT *print?*)
(quit)
(rewrite-msg-off)
(rewrite-msg-on)
(skip)
(skip-msg *msg*)
(undo &OPT *to[1]*)
(time *strat*)

Structural Rules

(copy *fnum*)
(delete *&rest fnums*)
(hide *&rest fnums*)
(reveal *&rest fnums*)

Propositional Rules

(bddsimp &OPT *fnums[*] dynamic-ordering?*)
(case *&rest formulas*)
(flatten *&rest fnums[*]*)
(iff *&rest fnums*)
(lift-if *&rest fnums[*] updates?[t]*)
(prop)
(propax)
(split &OPT *fnum[*]*)

Quantifier Rules

(inst &OPT *fnum[*] &rest terms*)
(instantiate *fnum[*] terms* &OPT *copy?*)
(inst-cp &OPT *fnum[*] &rest terms*)
(inst? &OPT *fnums[*] subst where[*] copy? if-match*)
(skolem *fnum constants*)
(skolem! &OPT *fnums[*]*)
(skolem-typepred &OPT *fnum[*]*)
(skosimp &OPT *fnum[*]*)
(skosimp*)

Equality Rules

(beta &OPT *fnums rewrite-flag*)
(case-replace *formula*)
(name *name expr*)
(name-replace *name expr* &OPT *hide?[T]*)
(name-replace* *name-and-exprs* &OPT *hide?[T]*)
(replace *fnum* &OPT *fnums[*] dir hide?*)
(replace* *&rest fnums*)
(same-name *name1 name2* &OPT *type*)

Definition and Lemma Rules

(expand *name* &OPT *fnum[*] occurrence if-simplifies assert?*)
(forward-chain *name*)
(lemma *name* &OPT *subst*)
(rewrite *lname* &OPT *fnums[*] subst target-fnums[*] dir[LR] order[IN]*)
(rewrite-lemma *lemma subst* &OPT *fnums[*] dir[LR]*)
(simplify-with-rewrites &OPT *defs theories rewrites*)
(use *lname* &OPT *subst if-match[best]*)

Extensionality Rules

(apply-eta *term* &OPT *type*)
(apply-extensionality &OPT *fnum[+] keep? hide?*)
(eta *type*)
(extensionality *type*)
(replace-eta *term* &OPT *type keep?*)
(replace-extensionality *expr expr* &OPT *expected keep?*)

Induction Rules

(induct *var* &OPT *fnum[1] name*)
(induct-and-rewrite *var* &OPT *fnum &rest rewrites*)
(induct-and-rewrite! *var* &OPT *fnum &rest rewrites*)
(induct-and-simplify *var* &OPT *fnum[1] name defs[T] if-match[best] theories rewrites exclude*)
(measure-induct *measure vars* &OPT *fnum[1]*)
(name-induct-and-rewrite *var* &OPT *fnum[1] name &rest rewrites*)

Decision Procedure and Rewriting Rules

(assert &OPT *fnums[*] rewrite-flag flush? linear? cases-rewrite?*)
(bash &OPT *if-match[T] updates?[T]*)
(do-rewrite &OPT *fnums[*] rewrite-flag flush? linear? cases-rewrite?*)
(grind &OPT *defs[!] theories rewrites exclude if-match[T] updates?[T]*)
(ground)
(record &OPT *fnums[*] rewrite-flag flush? linear?*)
(reduce &OPT *if-match[T] updates?[T]*)
(simplify &OPT *fnums[*] record? rewrite? rewrite-flag flush? linear? cases-rewrite?*)
(smash &OPT *updates?[T]*)

Installation of Rewrite Rules

(auto-rewrite *&rest names*)
(auto-rewrite! *&rest names*)
(auto-rewrite-defs &OPT *explicit? always? exclude-theories*)
(auto-rewrite-explicit &OPT *always?*)
(auto-rewrite-theories *&rest names*)
(auto-rewrite-theory *name* &OPT *exclude defs always? tccs?*)
(install-rewrites &OPT *defs theories rewrites exclude-theories exclude*)

Removing Installed Rewrite Rules

(stop-rewrite *&rest names*)
(stop-rewrite-theory *&rest names*)

Tracking Rewrite Rules

(trace *&rest names*)
(track-rewrite *&rest names*)
(untrace *&rest names*)
(untrack-rewrite *&rest names*)

Type Constraint Rules

(skolem-typepred &OPT *fnum[*]*)
(typepred *&rest exprs*)

Mu Calculus Rules

(musimp &OPT *fnums[*] dynamic-ordering?*)
(model-check &OPT *dynamic-ordering?[T] cases-rewrite?[T]*)

Convert Strategy to a Rule

(apply *strategy* &OPT comment)

Proof Strategies

(branch *strat strats*)
(else *strat strat*)
(if *condition strat strat*)
(let ((*var*$_1$ *expr*$_1$)$\cdots$(*var*$_n$ *expr*$_n$)) *strat*)
(query*)
(quote *strat*)
(repeat *strat*)
(repeat* *strat*)
(rerun &OPT *proof recheck? break?*)
(spread *strat strats*)
(spread! *strat strats*)
(spread@ *strat strats*)
(tcc &OPT *defs[!]*)
(then *&rest steps*)
(then@ *&rest strats*)
(try *strat strat strat*)
(try-branch *strat strats strat*)

Prover Emacs Commands

These commands are only available when a proof is in progress, and the `*pvs*` buffer is current.

Prover Emacs Help	`TAB h`
Prover Command Help	`TAB H`
Any Command	`TAB TAB`
apply-extensionality	`TAB E`
assert	`TAB a`
auto-rewrite	`TAB A`
auto-rewrite-theory	`TAB C-a`
bddsimp	`TAB B`
beta	`TAB b`
case	`TAB c`
case-replace	`TAB C`
copy	`TAB 2`
delete	`TAB d`
do-rewrite	`TAB D`
expand	`TAB e`
extensionality	`TAB x`
flatten	`TAB f`
grind	`TAB G`
ground	`TAB g`
hide	`TAB C-h`
iff	`TAB F`
induct	`TAB I`
induct-and-simplify	`TAB C-s`
inst	`TAB i`
inst?	`TAB ?`
lemma	`TAB L`
lift-if	`TAB l`
model-check	`TAB M`
musimp	`TAB m`
name	`TAB n`
postpone	`TAB P`
prop	`TAB p`
quit	`TAB C-q`
replace	`TAB r`
replace-eta	`TAB 8`
rewrite	`TAB R`
skolem!	`TAB !`
skosimp	`TAB S`
skosimp*	`TAB *`
split	`TAB s`
tcc	`TAB T`

then	`TAB C-t`
typepred	`TAB t`
undo	`TAB u`
Run Proof Step	`TAB 1`
Undo Proof Step	`TAB U`
Skip Proof Step	`TAB #`
Insert Quotes	`TAB ´`
Wrap with Parens	`TAB C-j`

References

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

[ALW93] Mark D. Aagard, Miriam E. Leeser, and Phillip J. Windley. Toward a super duper hardware tactic. In *Proceedings of the HOL User's Group Workshop*, pages 401–414, 1993.

[BCM+90] J. R. Burch, E. M. Clarke, K. L McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, PA, June 1990. IEEE Computer Society.

[BJ93] Ricky W. Butler and Sally C. Johnson. Formal methods for life-critical software. In *Computing in Aerospace Conference*, pages 319–329, San Diego, CA, October 1993.

[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.

[But93] Ricky W. Butler. An elementary tutorial on formal specification and verification using PVS. NASA Technical Memorandum 108991, NASA Langley Research Center, Hampton, VA, June 1993. Available through WWW or `ftp` from `ftp://air16.larc.nasa.gov/pub/fm/larc/PVS-tutorial/pvs-tutorial.ps`.

[Can94] F. J. Cantu. Verifying an *n-bit* arithmetic logic unit. Blue book note 935, University of Edinburgh, June 1994.

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

[Const86] R. L. Constable, *et al*. *Implementing Mathematics with the Nuprl*. Prentice-Hall, New Jersey, 1986.

[Cou93] Costas Courcoubetis, editor. *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, Elounda, Greece, June/July 1993. Springer-Verlag.

[CRSS94] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Kumar and Kropf [KK94], pages 203–222.

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

[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, Dordrecht, The Netherlands, 1988.

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

[Hoo94] Jozef Hooman. Correctness of real time systems by construction. In Langmaack et al. [LdV94], pages 19–40.

[JMC94] Steven D. Johnson, Paul S. Miner, and Albert Camlleri. Studies of the single-pulser in various reasoning systems. In Kumar and Kropf [KK94], pages 126–145.

[KK94] Ramayya Kumar and Thomas Kropf, editors. *Theorem Provers in Circuit Design (TPCD '94)*, volume 910 of *Lecture Notes in Computer Science*, Bad Herrenalb, Germany, September 1994. Springer-Verlag.

[KSK93] R. Kumar, K. Schneider, and T. Kropf. Structuring and automating hardware proofs in a higher-order therem proving environment. *Formal Methods in System Design*, 2(2):165–223, 1993.

[LdV94] H. Langmaack, W.-P. de Roever, and J. Vytopil, editors. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, Lübeck, Germany, September 1994. Springer-Verlag.

[LR93a] Patrick Lincoln and John Rushby. Formal verification of an algorithm for interactive consistency under a hybrid fault model. In Courcoubetis [Cou93], pages 292–304.

[LR93b] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23*, pages 402–411, Toulouse, France, June 1993. IEEE Computer Society.

[LR94] Patrick Lincoln and John Rushby. Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model. In *COMPASS '94 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, pages 107–120, Gaithersburg, MD, June 1994. IEEE Washington Section.

[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.

[MPJ94] Paul S. Miner, Shyamsundar Pullela, and Steven D. Johnson. Interaction of formal design systems in the development of a fault-tolerant clock synchronization circuit. In *13th Symposium on Reliable Distributed Systems*, pages 128–137, Dana Point, CA, October 1994. IEEE Computer Society.

[MS95] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL, 1995. IEEE Computer Society.

[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.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[OSR93a] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. A new edition for PVS Version 2 is expected in early 1995.

[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. A new edition for PVS Version 2 is expected in early 1995.

[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.

[Raj94] P. Sreeranga Rajan. Transformations in high-level synthesis: Formal specification and efficient mechanical verification. Technical Report SRI-CSL-94-10, Computer Science Laboratory, SRI International, Menlo Park, CA, October 1994. Revised version of Technical Report NL-TN 118/94, Philips Research Laboratories, Eindhoven, The Netherlands, April 1994.

[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.

[Rus95] John Rushby. Proof Movie II: A proof with PVS. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1995. Forthcoming.

[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.

[Sha93a] N. Shankar. Abstract datatypes in PVS. Technical Report SRI-CSL-93-9, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.

[Sha93b] Natarajan Shankar. Verification of real-time systems using PVS. In Courcoubetis [Cou93], pages 280–291.

[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. A new edition for PVS Version 2 is expected in early 1995.

[SS94] Jens U. Skakkebæk and N. Shankar. Towards a Duration Calculus proof assistant in PVS. In Langmaack et al. [LdV94], pages 660–679.

[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)*, New York, NY, 1982. Volume 138 of *Lecture Notes in Computer Science*, Springer-Verlag.

[SSR95] Mandayam Srivas, Natarajan Shankar, and Sreeranga Rajan. Hardware verification using PVS: A tutorial. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1995. Forthcoming.