A Less Elementary Tutorial for the PVS Specification and Verification System[1]

Technical Report CSL-95-10

J. M. Rushby and D. W. J. Stringer-Calvert[2]
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
{Rushby,Dave_SC}@csl.sri.com

August 22, 1996

[1]This work was partially sponsored by NASA Langley under Contract NAS1-20334.

[2]Main affiliation: Department of Computer Science, University of York, U.K.

Abstract

PVS is a verification system that provides a specification language integrated with support tools and a theorem-prover. It has been used at SRI and elsewhere to perform verifications of several significant algorithms (primarily for fault-tolerance) and large hardware designs.

This tutorial introduces some of the more powerful strategies provided by the PVS theorem prover. It consists of two parts: the first extends a previous tutorial by Ricky Butler[But93], demonstrating how his proofs may be performed in a more automated manner; the second uses the "unwinding theorem" from the noninterference formulation of security to introduce theorem-proving strategies for induction that cannot be demonstrated in the framework of Ricky Butler's example.

Using the more powerful strategies of PVS to automate easy proofs (and the easy parts of hard proofs) frees users to concentrate on truly difficult proofs. Automation also makes proofs more robust to changes in the specification, thereby facilitating active design exploration and adaptation to changed requirements.

This tutorial also shows how specifications and proofs may be better presented using the LaTeX and PostScript generating facilities of PVS. The PVS files for these examples are available at `http://www.csl.sri.com/pvs/examples/csl-95-10.html`.

Contents

List of Figures

Chapter 1

Introduction

This tutorial invites you to explore some of the more powerful theorem proving capabilities of the PVS verification system, moving from long, mundane proof scripts full of trivial details, to more automatic theorem proving where the user directs only the key steps. It is suggested that you follow through the tutorial with PVS running on your workstation. You can obtain the specifications and proofs developed here over the World Wide Web from `http://www.csl.sri.com/csl-95-10.html` or by ftp from `ftp://ftp.csl.sri.com/pub/pvs/examples/elementary-tutorial`.

1.1 Why Seek Highly Automated Proofs?

The motivations for performing proofs in an automated manner are: first, to liberate human users from the drudgery of low-level details, so that they can best direct their energies to the truly difficult and significant steps in a proof; second, to make it feasible to prove big theorems; and, third, to make the investment in a proof an incentive—rather than a disincentive—to the exploration of alternative specifications and designs.

The first of these motivations should need little justification: exposure to the tedium of low-level "proof assistants" has convinced many that mechanized proof is infeasible for real examples. This is unfortunate, because many formal specifications contain significant errors when first written, and automated proof checking can be one of the *fastest* ways to detect errors early in the lifecycle. Note, however, that the theorem prover needs to be designed to facilitate this: most highly automated approaches to proof are intended to prove true theorems—not to help detect errors in untrue ones—so that when an automated prover fails to complete a proof, it can be difficult to determine whether the cause is a false theorem or an inadequate proof method. For this reason, the basic PVS proof steps are powerful, but deterministic (being based on decision procedures), and are used under interactive control. When one of the more powerful heuristic proof strategies fails, the user can explore the cause of failure by interactively invoking the more basic steps.

The second motivation concerns the fact that formal verification is often applied to theorems that are large, but shallow—such as those that assert the correctness of the microcode or the pipeline control circuitry of a microprocessor. Formal verification can accomplish what massive simulation and testing cannot: examination of the behaviors of these designs under *all* circumstances. But to be practical, it must be possible to actually carry out the formal verifications for industrial-scale designs. This is simply infeasible without massively automated theorem proving. Advances over the last few years have brought theorem proving to the point where it is now feasible to tackle such industrial-scale problems [SM96, RSS96, PD96].

The third motivation is prompted by the observation that formal specifications are seldom static: they change as flaws are corrected, as new requirements emerge, and as improved approaches and designs are discovered. Mechanization of formal methods should support such changes and should encourage active design exploration in the same way that computational fluid dynamics supports the refinement of aerofoil designs. To achieve this, it is important that previously developed proofs should be robust in the face of small changes to the specification—since otherwise investment in an existing proof will discourage change and experimentation. For proofs to be robust, they must be recorded at a fairly high level—giving just the main steps of the argument, and leaving automation to fill in the details—since highly detailed, line-by-line arguments are unlikely to remain correct in the face of changes. In the case of PVS proofs, the goal should generally be to use the highest level, most automatic proof strategies possible, and to use explicit proof steps (e.g., those naming a specific sequent formula) as sparingly as possible. Such proof descriptions guide the prover along the main steps of the argument (which is likely to be robust), and allow automation to calculate the details afresh each time.

This tutorial introduces some of the higher-level PVS proof strategies and explains, by example, how to use them effectively. The remainder of this chapter provides a brief introduction to the PVS verification system. Chapter two presents an extension to a tutorial by Ricky Butler [But93], describing how more automated proofs can be developed for his examples. The third and final chapter uses a verification of Goguen and Meseguer's unwinding theorem for noninterference security policies [GM84] to illustrate some PVS induction strategies, and also demonstrates how PVS can be used to formalize a pencil-and-paper development.

1.2 PVS

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) [RLS79, SLR78], STP [SSMS82], and EHDM [MSR85, RvHO91]. PVS stands for "Prototype Verification System," because it was built partly as a lightweight prototype to explore "next generation" technology for EHDM, though it has now outgrown that role.

PVS consists of a specification language, a number of predefined theories, a theorem prover, various utilities, documentation, and several examples that illustrate different methods of using the system in several application areas. PVS exploits the synergy between a highly expressive specification language and powerful automated deduction; for example, some elements of the specification language are made possible because the typechecker can use theorem proving. This distinguishing feature of PVS has allowed perspicuous and efficient treatment of many examples that are considered difficult for other verification systems.

The specification language of PVS is based on classical, typed higher-order logic. The base types include uninterpreted types that may be introduced by the user, and built-in types such as the booleans, integers, reals, and the ordinals up to ϵ_0; the type-constructors include functions, sets, tuples, records, enumerations, and recursively-defined abstract data types such as lists and trees. Predicate subtypes and dependent types can be used to introduce constraints, such as the type of prime numbers. These constrained types may incur proof obligations during typechecking, but greatly increase the expressiveness and naturalness of specifications. In practice, most of the obligations are discharged automatically by the theorem prover. PVS specifications are organized into parameterized theories that may contain assumptions, definitions, axioms, and theorems. Definitions are conservative (i.e., cannot introduce inconsistencies); to ensure this, recursive function definitions generate proof obligations to guarantee termination. PVS expressions provide the usual arithmetic and logical operators, function application, lambda abstraction, and quantifiers, with a traditional syntax. Names may be freely overloaded, including those of the built-in operators such as `AND` and `+`. A case expression provides pattern-matching over the constructors of abstract data types, and tables allow piecewise-continuous functions to be specified in a visually appealing manner.

The PVS theorem prover provides a collection of powerful primitive inference procedures that are applied interactively under user guidance within a sequent calculus framework. The primitive inferences include propositional and quantifier rules, induction, rewriting, and decision procedures for linear arithmetic over both integers and reals and for Park's μ-calculus. The implementations of these primitive inferences are optimized for large proofs: for example, propositional simplification and μ-calculus use BDDs, and auto-rewrites are cached for efficiency. User-defined procedures can combine these primitive inferences to yield higher-level proof strategies, such as those for induction and CTL model checking. Proofs yield scripts that can be edited, attached to additional formulas, and rerun. This allows many similar theorems to be proved efficiently, permits proofs to be adjusted economically to follow changes in requirements or design, and encourages the development of readable proofs.

PVS is fully documented in separate manuals for the language [OSR93a], prover [SOR93], system [OSR93b], and semantics [SO96]. Tutorials provide a general introduction [But93, COR$^+$95], plus more specialized treatments for hardware [ORSS94], abstract data types [Sha93a], and tabular and requirements specifications [ORS95].

PVS has been installed at hundreds of sites in North America, Europe, and Asia; recent work has developed PVS methodologies for highly automated hardware verification [CRSS94, RSS96, SM96] (including integration with model checking [RSS95]), and for concurrent and real-time systems [Sha93b, Hoo94, AH96] (including a transparent embedding of the duration calculus [SS94]). Applications have included microcode verification for a commercial microprocessor [SM95], verification of fault-tolerant algorithms [LR93, LR94] and a cache-coherence protocol [PD96], and formalization of Space Shuttle requirements [Di 96, CD96], IEEE standards for floating point [CM95] and multimedia collaborations [RRV95]. A comprehensive bibliography of applications performed in PVS is available [Rus].

1.3 Obtaining PVS

PVS is implemented in Common Lisp and runs on several modern Unix workstations. Versions in Allegro Lisp for Sun and IBM workstations are available by anonymous ftp. All PVS installations must be licensed by SRI International, but there is no charge. (We do charge for tapes and for nonstandard versions.)

The specifications and proofs presented here require PVS 2, released June 1, 1995. To obtain a copy of PVS by anonymous ftp, retrieve the file `README` from directory `/pub/pvs/pvs2` on `ftp.csl.sri.com` [192.12.33.94] and follow the instructions.[1] Or via the World Wide Web, open the URL `http://www.csl.sri.com/pvs.html` (this also gives access to the mirror sites). For further information on PVS, please send a message to `pvs-request@csl.sri.com`.

Acknowledgments

We are grateful to Ricky Butler, Drew Dean, Piotr Rudnicki, and N. Shankar for reading earlier versions of this report and for providing comments that substantially changed and improved its content and presentation.

[1]There are mirror sites at the Universities of York, England (`ftp://ftp.cs.york.ac.uk/pub/pvs`), Paris VI, France (`ftp://ftp.ibp.fr/pub/pvs`), Ulm, Germany (`ftp://ftp.informatik.uni-ulm.de/pub/KI/pvs`), and Tokyo, Japan (`ftp://nicosia.is.s.u-tokyo.ac.jp/pub/misc/pvs`).

Chapter 2

Seat Reservation Problem

This chapter develops proofs for an example due to Ricky Butler [But93]. Butler's tutorial also develops proofs for the same theorems, but in a low level, step-by-step manner. Here, we show how the more powerful rules and strategies of PVS may be used to produce higher level, more automated proofs. As explained in the introduction, the main benefit of automated proofs is that they tend to be robust in the face of reasonably small changes to a specification. They are also closer to the level at which you might wish to describe a proof to a human colleague, and thereby facilitate the extraction of a "journal style" proof description. Also, as you gain experience, you will find that it is generally faster and less distracting to let the automation deal with easy theorems (and the easy parts of hard theorems), leaving you free to concentrate on the hard theorems and crucial steps.

2.1 Requirements

Ricky Butler's report considers the formal specification and verification of an automated airline seat assignment system. This section briefly outlines the problem covered—for more details refer to Ricky Butler's original report.[1] The requirements for the system are given as:

1. The system shall make seat assignments for passengers on scheduled airline flights.
2. The system shall maintain a database of seat assignments.
3. The system shall support a fleet having different aircraft types.
4. Passengers shall be allowed to specify preferences for seat type (e.g., window or aisle).
5. The system shall provide the following operations or transactions:
    - Make a new seat assignment
    - Cancel an existing seat assignment

[1] This is available electronically from `http://atb-www.larc.nasa.gov/ftp/larc/PVS-tutorial`; get the files named "revised-pvs-tutorial.*" and "revised-specs.dmp."

2.2 The PVS specification

Ricky Butler specifies the basic properties of aircraft and reservations in the PVS theory `basic_defs`. These include the seating grid of the aircraft (`row/position`), identifiers for the flight, aircraft type, position preference and passenger ID. Uninterpreted functions are used to specify existence of a particular seat on a particular type of aircraft (not all will hold `nrows` × `nposits` passengers), whether a particular seat meets a passenger preference, and a mapping from flight identifier to aircraft type.

The PVS theory `ops` then uses these definitions to specify the required operations on the database, i.e., to make a reservation/seat assignment and to cancel an assignment. Putative theorems are specified to validate the specification of these operations, which are also shown to preserve certain invariants.

The PVS specification used here differs in only minor ways from that given in Ricky Butler's report. These differences are explained below. To format the specification as presented here, we used the PVS command `M-x latex-pvs-file` to generate prettyprinted LaTeX output (you will need to use the style option `pvs.sty` to process the output of this command). Note that comments are dropped in LaTeX-printed specifications; this is a bug. The raw ascii representations of the specification are given in Appendix A.1.

basic_defs : THEORY

BEGIN

nrows : posnat

nposits : posnat

row : TYPE $= \{n : \text{posnat} \mid 1 \leq n \wedge n \leq \text{nrows}\}$ CONTAINING 1

position : TYPE $= \{n : \text{posnat} \mid 1 \leq n \wedge n \leq \text{nposits}\}$ CONTAINING 1

flight : TYPE

plane : NONEMPTY_TYPE

preference : TYPE

passenger : NONEMPTY_TYPE

seat_assignment : TYPE = [# seat : [row, position], pass : passenger #]

flight_assignments : TYPE = set[seat_assignment]

flt_db : TYPE = [flight $\rightarrow$ flight_assignments]

initial_state((flt : flight)) : flight_assignments = $\emptyset$[seat_assignment]

seat_exists : pred[[plane, [row, position]]]

meets_pref : pred[[plane, [row, position], preference]]

aircraft : [flight $\rightarrow$ plane]

END basic_defs

ops : THEORY
BEGIN

IMPORTING basic_defs

flt : VAR flight

pas : VAR passenger

db : VAR flt_db

a, b : VAR seat_assignment

pref : VAR preference

seat : VAR [row, position]

Cancel_assn(flt, pas, db) : flt_db =
db WITH [(flt) := $\{a \mid a \in$ db(flt) $\wedge$ pass(a) $\neq$ pas$\}$]

pref_filled(db, flt, pref) : bool =
$\forall$ seat : meets_pref(aircraft(flt), seat, pref) $\supset$ ($\exists\, a$: $a \in$ db(flt) $\wedge$ seat(a) = seat)

Next_seat : [flt_db, flight, preference $\rightarrow$ [row, position]]

Next_seat_ax : AXIOM
$\neg$ pref_filled(db, flt, pref) $\supset$ seat_exists(aircraft(flt), Next_seat(db, flt, pref))

Next_seat_ax_2 : AXIOM
$\neg$pref_filled(db, flt, pref) $\supset$ ($\forall\, a$: $a \in$ db(flt) $\supset$ seat(a) $\neq$ Next_seat(db, flt, pref))

Next_seat_ax_3 : AXIOM
$\neg$ pref_filled(db, flt, pref) $\supset$ meets_pref(aircraft(flt), Next_seat(db, flt, pref), pref)

pass_on_flight(pas, flt, db) : bool = $\exists\, a$: pass(a) = pas $\wedge$ $a \in$ db(flt)

Make_assn(flt, pas, pref, db) : flt_db =
IF pref_filled(db, flt, pref) $\vee$ pass_on_flight(pas, flt, db) THEN db
ELSE LET a = (# seat := Next_seat(db, flt, pref), pass := pas #)
IN db WITH [(flt) := add(a, db(flt))]
ENDIF

Lookup(flt, pas, db) : [row, position] = seat(ε($\{a \mid a \in$ db(flt) $\wedge$ pass(a) = pas$\}$))

existence(db) : bool = $\forall\, a$, flt : $a \in$ db(flt) $\supset$ seat_exists(aircraft(flt), seat(a))

uniqueness(db) : bool =
$\forall\, a, b$, flt : $a \in$ db(flt) $\wedge$ $b \in$ db(flt) $\wedge$ pass(a) = pass(b) $\supset$ $a = b$

one_per_seat(db) : bool =
$\forall\, a, b$, flt : $a \in$ db(flt) $\wedge$ $b \in$ db(flt) $\wedge$ seat(a) = seat(b) $\supset$ $a = b$

db_invariant(db) : bool = existence(db) $\wedge$ uniqueness(db)

Cancel_assn_inv : THEOREM
  db_invariant(db) $\supset$ db_invariant(Cancel_assn(flt, pas, db))

MAe : THEOREM existence(db) $\supset$ existence(Make_assn(flt, pas, pref, db))

MAu : THEOREM uniqueness(db) $\supset$ uniqueness(Make_assn(flt, pas, pref, db))

Make_assn_inv : THEOREM
  db_invariant(db) $\supset$ db_invariant(Make_assn(flt, pas, pref, db))

initial_state_inv : THEOREM db_invariant(initial_state)

Cancel_inv_one_per_seat : THEOREM
  one_per_seat(db) $\supset$ one_per_seat(Cancel_assn(flt, pas, db))

Make_inv_one_per_seat : THEOREM
  one_per_seat(db) $\supset$ one_per_seat(Make_assn(flt, pas, pref, db))

initial_one_per_seat : THEOREM one_per_seat(initial_state)

Make_Cancel : THEOREM
  $\neg$ pass_on_flight(pas, flt, db)
    $\supset$ Cancel_assn(flt, pas, Make_assn(flt, pas, pref, db)) = db

Cancel_putative : THEOREM
  $\neg$($\exists$ (a : seat_assignment) :
    $a \in$ Cancel_assn(flt, pas, db)(flt) $\wedge$ pass(a) = pas)

Make_putative : THEOREM
  $\neg$ pref_filled(db, flt, pref)
    $\supset$ ($\exists$ (x : seat_assignment) : $x \in$ Make_assn(flt, pas, pref, db)(flt) $\wedge$ pass(x) = pas)

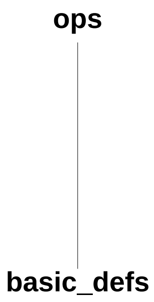
Lookup_putative : THEOREM
  $\neg$ (pref_filled(db, flt, pref) $\vee$ pass_on_flight(pas, flt, db))
    $\supset$ meets_pref(aircraft(flt), Lookup(flt, pas, Make_assn(flt, pas, pref, db)), pref)

END ops

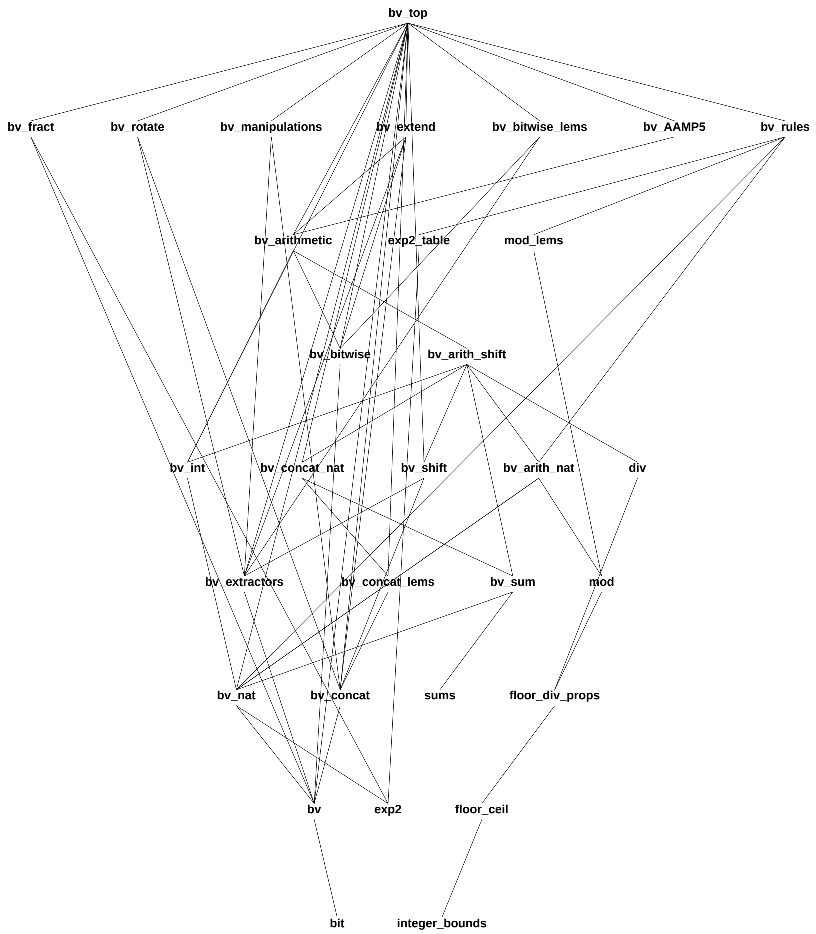
Incidentally, the exact form of a LaTeX-printed PVS specification is partly determined by the value of PVS's LaTeX-linelength variable, which influences where the prettyprinter chooses to break lines. This variable can be set by `M-x latex-set-linelength`. Often, some declarations look best set with one value of this parameter, and others with another. In this case, it is often simplest to generate two copies of the LaTeX-printed specification using different values of the parameter and then select individual declarations from one file or the other for the final version. This was done here, using linelengths of 100 and 120. Editing the LaTeX text generated by PVS is also possible, but requires a good understanding of the macro package `pvs.sty`.

2.3 Theory Dependencies

Ricky Butler structured his specification into two theories: `basic_defs` and `ops`. In more complicated specifications it is easy to lose track of the dependencies between theories, and PVS can help (if it is running under X-windows on a machine with Tcl/Tk available) by representing these graphically. On Ricky Butler's example, the command `M-x x-theory-hierarchy` produces the following display.



A much more complicated example might produce something like the following.



The theory names in these displays are mouse sensitive: clicking left-mouse on a theory name causes PVS to jump to the corresponding theory in its Emacs buffer. Holding down the `control` key and left-mouse simultaneously allows you to rearrange the layout of the graphical display, while clicking left-mouse on the `Gen PS` button generates a postscript file that can be included in a document such as this.

2.4 Adjustments to the Specifications

The differences between our specification and Ricky Butler's are fivefold:

- We give the declarations in a different order in `ops.pvs`, and tend to use global variable declarations, rather than declarations local to a given declaration. These differences are simply stylistic and due to the fact that Ricky Butler rearranged his specification in updating it to PVS 2, whereas we independently updated his original PVS 1 specification.

- We define `seat_exists` and `meets_pref` in `basic_defs.pvs` explicitly as predicates (using the PVS identifier `pred`), rather than as functions with range type `bool`. These are semantically equivalent, but we consider it closer to their intended interpretation to declare `seat_exists` and `meets_pref` explicitly as predicates.

- We use `NONEMPTY_TYPE` and `CONTAINING` clauses in a few places in order to eliminate TCCs or to automate their proofs. This is the most significant difference and is explained in the section that follows.

- At the end of his Section 3.5, Ricky Butler challenges the "ambitious reader" to add the following definition to the specification.

```
Lookup(flt: flight, pas: passenger, db: flt_db): [row,position] =
        seat(choose( {a | member(a,db(flt)) AND pass(a) = pas} ))
```

  Our specification uses `epsilon` in place of `choose`. The differences between these two approaches are discussed later in Section 2.4.1.

- We add `NOT pref_filled(db, flt, pref)` as an antecedent to the axiom `Next_seat_ax_2` that is introduced by Ricky Butler at the end of his Section 3.4, thereby producing the following modified axiom.

```
Next_seat_ax_2: AXIOM
       NOT pref_filled(db, flt, pref) IMPLIES
          (FORALL a: member(a,db(flt)) IMPLIES
               seat(a) /= Next_seat(db,flt,pref))
```

Without the antecedent, this axiom is false in the case of a full flight, and renders the whole specification inconsistent. This flaw was pointed out by Piotr Rudnicki of the University of Alberta[2] and illustrates the perennial danger of axiomatic specifications: it is all too easy to write inconsistent axioms. PVS specifications that do not use axioms (except those from the prelude) are guaranteed to be consistent (provided all the TCCs have been proved), but it is not always appropriate to restrict specifications to this definitional fragment of PVS. Here, for example, the goal is to develop and to validate a "requirements" specification in which we postulate only the properties required of the `Next_seat` function; we do not wish to prescribe an implementation. A definitional specification of `Next_seat` would necessarily suggest an implementation. When an axiomatic style of specification seems appropriate, the best approach is indeed to present the specification in this style, but also to supply a definitional (and therefore consistent) specification that is proven to satisfy the axioms. This is described in Section 2.4.3.

2.4.1 Nonempty Types and Type Correctness Conditions

PVS allows empty types, provided you do not attempt to declare (or assert the existence of) any constants of such types, since this would be unsound. By default, PVS makes no assumptions about uninterpreted types (such as `flight` or `plane`), other than that different types are disjoint; in particular, the set that interprets an uninterpreted PVS type may have zero, finite, countable, or uncountable cardinality. When you declare a constant of a type, however, PVS needs to be sure that the type is nonempty. If both the type and constant are interpreted (e.g., if we added `x: row = 1` to `basic_defs`) the typechecker may need to generate a TCC to check that the constant satisfies the definition for the type (i.e., `1 <= 1 AND 1 <= nrows`). If the type is interpreted but the constant is uninterpreted (e.g., if we simply had added `x: row` to the specification), PVS may generate a TCC requiring you to show that the type is nonempty (i.e., `EXISTS (y:row): TRUE`). PVS is usually unable to prove such TCCs on its own (because they require exhibition of a member of the type concerned) unless you give it a hint by adding a `CONTAINING` clause to the type declaration concerned. Finally, if both the type and constant are uninterpreted, PVS requires you to explicitly declare the type to be nonempty using the `NONEMPTY_TYPE` or `TYPE+` keywords instead of simply `TYPE` (otherwise you will get an unprovable TCC).[3]

With the present specification, in the absence of `NONEMPTY_TYPE` and `CONTAINING` keywords, PVS generates several TCCs for `basic_defs`, including the following one.

```
% Existence TCC generated (line 31) for aircraft: [flight -> plane]

aircraft_TCC1: OBLIGATION (EXISTS (x: [flight -> plane]): TRUE);
```

[2] Piotr Rudnicki has developed a treatment of this example in the Mizar system. The example, and information about Mizar, are available from `http://web.cs.ualberta.ca/~piotr/Mizar/FLT_DB/`. Appendix B to the present report develops a PVS specification in the spirit of Rudnicki's Mizar treatment.

[3] The remaining combination, an uninterpreted type and interpreted constant (e.g., `foo: TYPE FROM nat` and `bar: foo = 99`), is not meaningful and always generates unprovable TCCs.

This is due to the following declaration.

```
aircraft: [flight -> plane]
```

Here we are asserting the existence of a function of type `[flight -> plane]`; now, a function type is nonempty if its range type is nonempty (or if both its domain and range are empty, but that is seldom an interesting case unless dependent types are involved), so we need to ensure that the uninterpreted type `plane` is declared as a `NONEMPTY_TYPE`.

As Ricky Butler noted, typechecking `ops.pvs` also generates a TCC from the declaration of the function `Next_seat` requiring us to show that its function type is nonempty.

```
% Existence TCC generated (line 22) for
    % Next_seat: [flt_db, flight, preference -> [row, position]]

Next_seat_TCC1: OBLIGATION
     (EXISTS (x: [[flt_db, flight, preference] -> [row, position]]): TRUE);
```

Ricky Butler discharged this TCC by constructing a suitable function, but we think it is neater to establish that the range type `[row, position]` is nonempty, thereby allowing PVS to suppress the TCC. To do this, the types `row` and `position` must be shown to be nonempty and, since these are interpreted, we can help PVS to do this by adding `CONTAINING 1` clauses to their declarations in `basic_defs.pvs`. This will cause PVS to generate TCCs to prove that `1` is a member of both these types, and the standard proof strategy invoked by the `M-x tcp` command is able to discharge those proof obligations automatically.

However, we are not done yet, because we find that typechecking `ops.pvs` generates yet another TCC. This particular TCC does not occur in Ricky Butler's description because the definition of `Lookup` is part of one of the examples he left to the reader.

```
% Existence TCC generated (line 47) for epsilon

Lookup_TCC1: OBLIGATION (EXISTS (x: seat_assignment): TRUE);
```

The function `epsilon` in the definition of `Lookup` returns a `seat_assignment`, and this type is therefore required to be nonempty. Inspecting the type `seat_assignment`, we see that it is a record consisting of a `[row, position]` pair (which we have just ensured is known to be nonempty), and a `passenger`. The latter is an uninterpreted type, so we modify its declaration to be `NONEMPTY_TYPE` and `ops.pvs` no longer generates TCCs.

2.4.2 Choose vs. Epsilon

The function `Lookup(flt, pas, db)`, which Ricky Butler (at the end of his Section 3.5) challenges the "ambitious reader" to add to the specification, is intended to return the `[row, position]` pair for the seat of passenger `pas` on flight `flt`, as recorded in the database `db`.

Now `db(flt)` is the set of seat assignments for flight `flt`, so the seat of passenger `pas` is `seat(a)`, where `a` is the seat assignment such that `member(a, db(flt))` and `pass(a)=pas`. We know that if the passenger is on the flight, there will be exactly one seat assignment `a` with this property, but this is not self-evident from the specification (it has to be established by proving the several "invariant" theorems). However, we can identify the set of relevant seat assignments as

```
    { a | member(a, db(flt)) AND pass(a) = pas }                          1
```

and can then choose one member of that set. (We "know" that the set will be a singleton, and the choice will therefore be deterministic.) A choice function `choose` is defined in the PVS prelude, so it looks as though we can write the specification of Lookup as follows.

```
  Lookup(flt, pas, db):  [row,position] =                                 2
          seat(choose( { a | member(a, db(flt)) AND pass(a) = pas } ))
```

The problem with this specification, which is the one proposed by Ricky Butler, is that it does not deal with the case where the passenger is not booked on the flight. The set in box $\boxed{1}$ will be empty in this case, and it is not obvious how to apply a choice function to an empty set. In fact, the choice function `choose` may be applied *only* when the set concerned is nonempty, and a TCC is generated to ensure this. Thus, Ricky Butler's specification generates the following unprovable TCC from the definition $\boxed{2}$.[4]

```
% Subtype TCC generated (line 51) for  {a | member(a, db(flt)) AND pass(a) = pas}

Lookup_TCC1: OBLIGATION
      (FORALL (db, flt, pas):
         nonempty?[seat_assignment]({a | member[seat_assignment](a, db(flt))
                                          AND pass(a) = pas}));
```

There are three ways to deal with this difficulty. One is to cause `Lookup` to return some specific, fictitious seat, such as `(0, 0)` when the passenger is not booked on the flight; the second is to cause it to return an arbitrary seat in this circumstance; and the third is to constrain `Lookup` so that it can be applied only when the passenger is known to be on the flight. The first of these has nothing to recommend it, the second and third are described below.

The PVS prelude provides two "choice" functions: `choose` and `epsilon`. Of these, `epsilon` is more basic, and less restrictive.[5] If `p` is a predicate (or, equivalently, a set) on some type `T`, then `epsilon(p)` is a value of type `T`; furthermore, if `p` is satisfiable (nonempty), then `epsilon(p)` satisfies (is a member of) it. This is specified in the defining axiom for `epsilon`, given in the PVS prelude as follows.

[4]Drew Dean of Princeton University first reported this.

[5]The name comes from Hilbert's use of the symbol ε (epsilon) for this operator [Lei69].

```
epsilon_ax: AXIOM (EXISTS x: p(x)) => p(epsilon(p))
```

Notice that if `p` is unsatisfiable (or, interpreted as a set, is empty), then `epsilon(p)` is some arbitrary value of type `T`.

When we know that `p` *is* satisfiable, it can be preferable to use `choose` instead of `epsilon`; `choose` is defined as follows.

```
choose(p: (nonempty?)): (p) = epsilon(p)
```

Notice that the return type specified is `(p)`; this is shorthand for the predicate subtype `{x:T | p(x)}`—so the fact that `choose(p)` satisfies `p` is embedded in its type, where the theorem prover can make automatic use of it, rather than requiring invocation of `epsilon_ax`. The price for this convenience is the need to establish during typechecking that the predicate `p` is indeed satisfiable (or, interpreted as a set, nonempty).

Since we do not know that the set given by specification $\boxed{1}$ is nonempty, we cannot use `choose` and must revert to `epsilon` as follows.

```
Lookup(flt, pas, db) : [row, position] =
   seat(epsilon( {a | member(a, db(flt)) AND pass(a) = pas} ))
```

With this definition, which generates no TCCs, `Lookup` returns an arbitrary seat when the passenger has no seat assignment; this might not be what is expected, but it is sufficient to prove the challenge theorem `Lookup_putative`, in which the relevant seat assignment is sure to exist (because it is explicitly constructed by `Make_assn`).

```
Lookup_putative: THEOREM
   NOT (pref_filled(db, flt, pref) OR pass_on_flight(pas, flt, db)) IMPLIES
       meets_pref(aircraft(flt),
                  Lookup(flt, pas, Make_assn(flt,pas,pref,db)),
                  pref)
```

This definition is the one used in this report. It may be considered unsatisfactory, however, to return an arbitrary seat when the passenger is not on the flight. An alternative approach is to disallow application of `Lookup` in this circumstance. This can be accomplished by modifying the argument types in the specification of `Lookup` as follows.

```
                                                                     3
Lookup(flt,
       pas,
       (db: {d : flt_db | pass_on_flight(pas, flt, d)}) ): [row, position] =
  seat(choose( {a | member(a, db(flt)) AND pass(a) = pas} ))
```

The function `Lookup` is said to be *dependently typed* in this version, since the *type* of the argument `db` depends on the *values* of the earlier arguments `pas` and `flt`: specifically, it is restricted to the subtype of `flt_db` consisting of those databases in which passenger `pas` is on flight `flt`. This definition generates the following TCC to establish that `choose` is applied to a nonempty argument.

```
% Subtype TCC generated (line 54) for {a | member(a, db(flt)) AND pass(a) = pas}

Lookup_TCC1: OBLIGATION
      (FORALL (flt, pas, db: {d: flt_db | pass_on_flight(pas, flt, d)}):
         nonempty?[seat_assignment]({a | member[seat_assignment](a, db(flt))
                                      AND pass(a) = pas}));
```

The dependent typing ensures that this TCC is true—and, in fact, it is proved automatically by the default TCC proof strategy. A consequence of the dependent typing in this version of `Lookup` is that TCCs are generated to ensure that the typing is respected whenever it is applied. Thus, the following TCC is generated from the theorem `Lookup_putative`.

```
% Subtype TCC generated (line 147) for  Make_assn(flt, pas, pref, db)

Lookup_putative_TCC1: OBLIGATION
      (FORALL (db: flt_db, flt: flight, pas: passenger, pref: preference):
         NOT((pref_filled(db, flt, pref) OR pass_on_flight(pas, flt, db)))
             IMPLIES pass_on_flight(pas, flt, Make_assn(flt, pas, pref, db)));
```

This TCC is also discharged by the default TCC proof strategy.

2.4.3 Definitional version of `Next_seat`

We noted earlier that to show that the axiomatic specification of `Next_seat` is consistent, we can provide a definitional version of the function and show that it satisfies all of the axioms asserted of the uninterpreted function. We could develop a truly constructive definition for `Next_seat`, but since our purpose here is merely to demonstrate consistency, we use a definition based on the `epsilon` operator. This is a convenient approach because we have three properties we wish the returned seat to have and do not care which seat the function returns when none satisfy these conditions.

We use here a syntactic variation of `epsilon`—`epsilon!`. The expression `epsilon! (x : T) : p(x)` is equivalent to, and we suggest more readable than, the standard forms `epsilon({x : T | p(x)})` and `epsilon(lambda (x:T): p(x))`. All functions that take a predicate as their argument can use this variant form, which transforms their syntax to that of variable-binding constructions such as quantifier and λ expressions.

```
Next_seat_defn(db,flt,pref) : [row, position] =
epsilon! (seat : [row,position]) : seat_exists(aircraft(flt),seat)
       AND (FORALL a : member(a,db(flt)) IMPLIES seat(a) /= seat)
       AND meets_pref(aircraft(flt),seat,pref)
```

We now reformulate our three axioms about `Next_seat` as theorems about `Next_seat_defn`.[6]

[6] This is clumsy and rather tedious; a forthcoming version of PVS will provide theory interpretations to simplify and automate this process.

```
Next_seat_th : THEOREM
        NOT pref_filled(db, flt, pref) IMPLIES
                seat_exists(aircraft(flt),Next_seat_defn(db,flt,pref))

Next_seat_th_2: THEOREM
        NOT pref_filled(db, flt, pref) IMPLIES
           (FORALL a: member(a,db(flt)) IMPLIES
                seat(a) /= Next_seat_defn(db,flt,pref))

Next_seat_th_3: THEOREM
        NOT pref_filled(db, flt, pref) IMPLIES
                meets_pref(aircraft(flt),Next_seat_defn(db,flt,pref),pref)
```

It turns out that these are not true theorems—there is an assumption underlying our specification that needs to be made explicit before these theorems can be proved. We use the attempted proof of the first theorem to demonstrate how theorem proving can assist the development of formal specifications by highlighting such missing assumptions. Since we have not yet introduced the powerful proof commands that are the subject of the latter part of this chapter, we perform this proof in rather small steps.

Our proof begins with the following sequent.

```
Next_seat_th :

  |-------
{1}    (FORALL (db: flt_db, flt: flight, pref: preference):
         NOT pref_filled(db, flt, pref)
             IMPLIES seat_exists(aircraft(flt), Next_seat_defn(db, flt, pref)))
```

We use (skosimp) to Skolemize the universal quantifier, and to simplify the implication:

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

  |-------
{1}    pref_filled(db!1, flt!1, pref!1)
{2}    seat_exists(aircraft(flt!1), Next_seat_defn(db!1, flt!1, pref!1))
```

Next, we expand the definitions of pref_filled and Next_seat_defn and use (skosimp) again.

```
Rule? (then (expand* "pref_filled" "Next_seat_defn")(skosimp))
Expanding the definition(s) of (pref_filled Next_seat_defn),
this simplifies to:
... intermediate sequent omitted
Skolemizing and flattening,
this simplifies to:
Next_seat_th :

{-1}    meets_pref(aircraft(flt!1), seat!1, pref!1)
  |-------
{1}    (EXISTS (a: seat_assignment): member(a, db!1(flt!1)) AND seat(a) = seat!1)
[2]    seat_exists(aircraft(flt!1),
                  epsilon! (seat: [row, position]):
                    seat_exists(aircraft(flt!1), seat)
                        AND
                      (FORALL (a: seat_assignment):
                         member(a, db!1(flt!1)) IMPLIES seat(a) /= seat)
                          AND meets_pref(aircraft(flt!1), seat, pref!1))
```

There are now no more useful expansions to perform, so we introduce the `epsilon_ax` axiom to reason about the application of the `epsilon!` operator.

```
Rule? (use "epsilon_ax[[row,position]]")
Using lemma epsilon_ax[[row,position]],
this simplifies to:
Next_seat_th :

{-1}    (EXISTS (x: [row, position]):
         (LAMBDA (seat: [row, position]):
            seat_exists(aircraft(flt!1), seat)
                AND
              (FORALL (a: seat_assignment):
                 member(a, db!1(flt!1)) IMPLIES seat(a) /= seat)
                  AND meets_pref(aircraft(flt!1), seat, pref!1))(x))
          =>
        (LAMBDA (seat: [row, position]):
           seat_exists(aircraft(flt!1), seat)
               AND
             (FORALL (a: seat_assignment):
                member(a, db!1(flt!1)) IMPLIES seat(a) /= seat)
                 AND
               meets_pref(aircraft(flt!1), seat,
                          pref!1))(epsilon(LAMBDA (seat: [row, position]):
                                             seat_exists(aircraft(flt!1), seat)
                                                 AND
                                               (FORALL (a: seat_assignment):
                                                  member(a, db!1(flt!1))
                                                      IMPLIES seat(a) /= seat)
                                                   AND
                                                 meets_pref(aircraft(flt!1),
                                                            seat, pref!1)))
[-2]    meets_pref(aircraft(flt!1), seat!1, pref!1)
  |-------
[1]    (EXISTS (a: seat_assignment): member(a, db!1(flt!1)) AND seat(a) = seat!1)
[2]    seat_exists(aircraft(flt!1),
                  epsilon! (seat: [row, position]):
                    seat_exists(aircraft(flt!1), seat)
                        AND
                      (FORALL (a: seat_assignment):
                         member(a, db!1(flt!1)) IMPLIES seat(a) /= seat)
                          AND meets_pref(aircraft(flt!1), seat, pref!1))
```

The square brackets around formula numbers `-2`, `1`, and `2` indicate that these are unchanged from the previous sequent; the curly braces around `-1` indicate that it is new, and a good place to focus our attention. Using `(ground)` to simplify formula `-1` generates two subgoals; the first of these is trivial and is discharged with another `(ground)`, leaving us with the following sequent.

```
Rule? (repeat (ground))
Applying propositional simplification and decision procedures,
this yields  2 subgoals:
... intermediate sequent omitted
Applying propositional simplification and decision procedures,

This completes the proof of Next_seat_th.1.

Next_seat_th.2 :

[-1]    meets_pref(aircraft(flt!1), seat!1, pref!1)
  |-------
{1}    (EXISTS (x: [row, position]):
         seat_exists(aircraft(flt!1), x)
             AND
           (FORALL (a: seat_assignment):
              member(a, db!1(flt!1)) IMPLIES seat(a) /= x)
               AND meets_pref(aircraft(flt!1), x, pref!1))
[2]    (EXISTS (a: seat_assignment): member(a, db!1(flt!1)) AND seat(a) = seat!1)
[3]    seat_exists(aircraft(flt!1),
                  epsilon! (seat: [row, position]):
                    seat_exists(aircraft(flt!1), seat)
                        AND
                      (FORALL (a: seat_assignment):
                         member(a, db!1(flt!1)) IMPLIES seat(a) /= seat)
                          AND meets_pref(aircraft(flt!1), seat, pref!1))
```

Formula 3 is no longer needed, so we hide it and then, comparing formulas `-1` and `1`, select `seat!1` to instantiate the existential quantifier in formula `1`.

```
Rule? (then (hide 3)(inst 1 "seat!1"))
Hiding formulas:  3,
this simplifies to:
... intermediate sequent omitted
Instantiating the top quantifier in 1 with the terms:
 seat!1,
this simplifies to:
Next_seat_th.2 :

[-1]    meets_pref(aircraft(flt!1), seat!1, pref!1)
  |-------
{1}    seat_exists(aircraft(flt!1), seat!1)
          AND
        (FORALL (a: seat_assignment):
           member(a, db!1(flt!1)) IMPLIES seat(a) /= seat!1)
            AND meets_pref(aircraft(flt!1), seat!1, pref!1)
[2]    (EXISTS (a: seat_assignment): member(a, db!1(flt!1)) AND seat(a) = seat!1)
```

We use (prop) to simplify the conjunction in formula 1, which generates 2 subgoals. We postpone examination of the first and examine the second.

```
Rule? (prop)
Applying propositional simplification,
this yields  2 subgoals:
... intermediate sequent omitted
Rule? (postpone)
Postponing Next_seat_th.2.1.

Next_seat_th.2.2 :

[-1]    meets_pref(aircraft(flt!1), seat!1, pref!1)
  |-------
{1}    (FORALL (a: seat_assignment):
         member(a, db!1(flt!1)) IMPLIES seat(a) /= seat!1)
[2]    (EXISTS (a: seat_assignment): member(a, db!1(flt!1)) AND seat(a) = seat!1)
```

We notice that formulas 1 and 2 are quite similar, so use (skolem!) then (inst?) to Skolemize the universal quantifier in 1, and then use the generated Skolem constant in 2

```
Rule? (then (skolem!)(inst?))
Skolemizing,
this simplifies to:
... intermediate sequent omitted
Found substitution:
a gets a!1,
Instantiating quantified variables,
this simplifies to:
Next_seat_th.2.2 :

[-1]    meets_pref(aircraft(flt!1), seat!1, pref!1)
  |-------
[1]    member(a!1, db!1(flt!1)) IMPLIES seat(a!1) /= seat!1
{2}    member(a!1, db!1(flt!1)) AND seat(a!1) = seat!1
```

This subgoal now completes with (prop), and the postponed first subgoal returns.

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

This completes the proof of Next_seat_th.2.2.

Next_seat_th.2.1 :

[-1]    meets_pref(aircraft(flt!1), seat!1, pref!1)
  |-------
{1}    seat_exists(aircraft(flt!1), seat!1)
[2]    (EXISTS (a: seat_assignment): member(a, db!1(flt!1)) AND seat(a) = seat!1)
```

Examining this sequent, we can see no clear way to proceed. Formula `2` (and the previously hidden formula `3`) provide little of interest, and we conclude that we really need to deduce that formula `1` follows from `-1`—that is, if a seat meets a preference, then that seat really exists on the aircraft type concerned. The specification provides no way to deduce this—it seems to be an implicit assumption. In order to complete the proof, we need to make the assumption explicit. We do this by adding a new axiom to the specification.

```
new_ax: AXIOM meets_pref(aircraft(flt), seat, pref)                    4
                  IMPLIES seat_exists(aircraft(flt), seat)
```

We could do this by abandoning the current proof, modifying the specification, and then rerunning the proof to return to the current state. This would be obviously inefficient, so PVS allows declarations to be modified and added to the specification mid-proof. Here, we position the cursor in the specification buffer above **Next_seat_th**, and give the command **M-x add-declaration**. A new buffer is created and we type the declaration **new_ax** from $\boxed{4}$ into it, indicating when we are finished by `C-c C-c`. PVS parses and typechecks the declaration and incorporates it into the specification. We can then return to the proof buffer and make use of this new axiom.

```
Rule? (use "new_ax")
Using lemma new_ax,
this simplifies to:
Next_seat_th.2.1 :

{-1}    meets_pref(aircraft(flt!1), seat!1, pref!1)
          IMPLIES seat_exists(aircraft(flt!1), seat!1)
[-2]    meets_pref(aircraft(flt!1), seat!1, pref!1)
  |-------
[1]    seat_exists(aircraft(flt!1), seat!1)
[2]    (EXISTS (a: seat_assignment): member(a, db!1(flt!1)) AND seat(a) = seat!1)
```

The proof then completes with `(prop)`.

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

This completes the proof of Next_seat_th.2.1.


This completes the proof of Next_seat_th.2.

Q.E.D.

Context was modified in mid-proof.
Would you like to rerun the proof?
```

Since the specification was modified during proof, the proof is regarded as provisional, and PVS offers to rerun it "clean" against the newly expanded specification. It is prudent to do so. No errors detected in this example, and the formula is considered proved.

We have seen how the act of attempting to prove consistency revealed an implicit assumption in the specification. This assumption was noted independently by Piotr Rudnicki in his Mizar treatment. Proofs of `Next_seat_th_2` and `Next_seat_th_3` do not reveal the need for any further assumptions,[7] and we deduce that our axiomatization of the `Next_seat` function is consistent.

2.5 The Proofs

In this section, we develop automated proofs for the theorems and lemmas in the theory `ops`. Ricky Butler's proofs mainly use a fairly restricted set of PVS prover commands: `skosimp*`, `assert`, `expand`, `lift-if`, `lemma`, `inst`, `inst?`, `case`, and `apply-extensionality`. In order to develop higher-level, more automated proofs, it is useful to have an idea of how the various higher-level prover commands are related to each other.

2.5.1 PVS Proof Commands

Below is a list of many of the PVS commands; the most useful are underlined. Note particularly those commands marked with $\surd$; these package the functionality of those that precede them in a convenient way and are the workhorses of automated proofs.

- Using decision procedures and auto-rewrites: <u>`assert`</u>, `simplify`, `do-rewrites`, `record`
- Basic propositional reasoning: <u>`bddsimp`</u>, `prop`, `iff`, `flatten`, `split`

$\surd$ Combine `prop` and `assert`: <u>`ground`</u>

- Simplifying `if-then-else` and `with` structures: <u>`lift-if`</u>

$\surd$ Iterate `lift-if` with `bddsimp`: <u>`smash`</u>

- Case split: <u>`case-replace`</u>, `case` (also `split` in combination with `lift-if` automates many case-splits)
- Note type information: <u>`typepred`</u>
- Skolemization: <u>`skosimp*`</u>, `skosimp`, <u>`skolem-typepred`</u>, `skolem!`, `skolem`

[7] Suitable proofs are

```
(GRIND :IF-MATCH NIL) (USE "epsilon_ax[[row,position]]") (GROUND) (("1" (REDUCE)) ("2"
(INST?) (GROUND) (("1" (USE "new_ax") (ASSERT)) ("2" (SKOSIMP) (INST?) (ASSERT))))))
```

and

```
(GRIND) (USE "epsilon_ax[[row,position]]") (GROUND) (INST?) (USE "new_ax") (REDUCE)
```

respectively.

- Instantiation: `inst?`, `inst`

√ Combine `inst?` with `skosimp*`, `smash`, and `assert`: `bash`

√ Iterate `bash`: `reduce`

- Setting up auto-rewrites: `install-rewrites`
  `auto-rewrite`, `auto-rewrite-theory`, `auto-rewrite-theories`

√ Set up auto-rewrites and then `reduce`: `grind`, `tcc`, `termination-tcc`

- Beta reduction: `beta`
- Lemma introduction: `use*`, `use`, `forward-chain`, `lemma`
- Equality reasoning: `replace`, `replace*`
- Definition expansion: `expand` (also see auto-rewriting)
- Conditional rewriting: `rewrite`, `simplify-with-rewrites` (also see auto-rewriting)
- Extensionality: `replace-extensionality`, `apply-extensionality`, `extensionality`

The commands listed above are sufficient to do the proofs in this chapter. In the third chapter we will meet some of the commands for induction.

- induction: `induct-and-simplify`, `measure-induct-and-simplify`, generalize,
  `measure-induct`, `name-induct`, `induct`

When large formulas (or nonlinear arithmetic) is involved, it can be helpful to name selected terms.

- Naming: `name-replace*`, `name-replace`, `name`, `same-name`

More advanced tutorials will introduce the eta rules and the model-checking commands.

- Eta rule: `replace-eta`, `apply-eta`, `eta`
- CTL model checking and μ-calculus: `model-check`

It is also necessary to know the commands for controlling the main functions of the prover.

- Control: `quit`, `undo`, `postpone`, `rerun`, `help` (also `delete`, `hide`, `reveal`, `copy`)

And for writing or understanding strategies, it is useful to know the combinators.

- Combinators: `apply`, `then`, `repeat`, `try`, `branch`, `spread`, `else`, `let`, `skip`, `fail`, `lisp`

Armed with this list of prover commands, we will now work through the proofs from Ricky Butler's tutorial.

2.5.2 Cancel_assn_inv

This, the first theorem in the specification, asserts that the db_invariant is preserved when a seat assignment is been canceled.

```
Cancel_assn_inv : THEOREM
    db_invariant(db) IMPLIES db_invariant(Cancel_assn(flt,pas,db))
```

The proof session begins with the following sequent.

```
Cancel_assn_inv :

  |-------
{1}   (FORALL (flt: flight, pas: passenger, db: assn_state):
         db_invariant(db) IMPLIES
             db_invariant(Cancel_assn(flt, pas, db)))
```

This theorem is an "obvious" one whose proof simply requires expansion of definitions and straightforward propositional calculus and equality reasoning. PVS 2 provides a strategy called grind to do this. It is based on the tcc strategy from PVS 1 (the default strategy applied to TCCs by the M-x typecheck-prove or M-x tcp command). Grind performs Skolemization, heuristic instantiation, if-lifting, rewriting and propositional simplification. The rewriting and instantiation can be closely controlled, as sometimes grind can be too large a hammer for the nut you are trying to crack. By default, grind sets up all the definitions relevant to the theorem as automatic rewrites, but uses them in a fairly cautious manner: any top-level conditions in a definition (i.e., antecedents to implications, or the condition of an if-then-else) must simplify to true or false in the context of a potential rewrite in order for the rewrite to take effect.

The Emacs interface to the PVS prover provides a number of shortcuts for common commands: rather than type (grind), it is sufficient to simply strike the two keys TAB G. To see a list of all these shortcut keystrokes (which were originally developed by C. Michael Holloway of NASA Langley Research Center), type TAB h. In the present case, the command (grind) fails to prove the theorem; following a flurry of messages about rewriting (these messages can be globally turned off by the prover command (rewrite-msg nil), or made terse by the prover command (rewrite-msg 0) or by the Emacs command M-x set-rewrite-depth), the strategy terminates with the following sequent.

```
Rule? (grind)
... reporting of rewrites omitted
Trying repeated Skolemization, instantiation, and if-lifting,
this yields  3 subgoals:
Cancel_assn_inv.1 :

{-1}   db!1(flt!2)(a!1)
{-2}   db!1(flt!2)(b!1)
{-3}   pass(a!1) = pass(b!1)
  |-------
{1}   db!1(flt!1)(a!1)
{2}   flt!1 = flt!2
{3}   a!1 = b!1
```

We notice that the formulas `-1` and `1` are almost the same, except for the different Skolem constants `flt!1` and `flt!2`. We can inspect the other sequents at the leaves of the proof tree with the command `(postpone)` (its shortcut is `TAB P`).

```
Rule? (postpone)
Postponing Cancel_assn_inv.1.

Cancel_assn_inv.2 :

{-1}   seat_exists(aircraft(flt!1), seat(a!1))
{-2}   db!1(flt!2)(a!1)
  |-------
{1}   flt!1 = flt!2
{2}   seat_exists(aircraft(flt!2), seat(a!1))

Rule? (postpone)
Postponing Cancel_assn_inv.2.

Cancel_assn_inv.3 :

{-1}   db!1(flt!2)(a!1)
  |-------
{1}   db!1(flt!1)(a!1)
{2}   flt!1 = flt!2
{3}   seat_exists(aircraft(flt!2), seat(a!1))

Rule?
```

Again, we see formulas above and below the line that are similar, except for the Skolem constants `flt!1` and `flt!2`. Circumstances like these indicate that the `grind` strategy probably chose the wrong instantiation somewhere along the way. Suitable responses are to try the strategy again, but with an additional argument that can tell `grind` either to leave the instantiation to us or to use a different criterion for choosing instantiations than its default method. Here, we'll try the manual approach. We undo the `grind` proof step with the `(undo)` command (`TAB u`), and then give the command `(grind :if-match nil)`. The `:if-match` token is a *keyword* that is used to indicate that the token following (here `nil`) is the value of an optional argument to `grind` called `if-match`. You can see the available arguments to a PVS strategy by giving a command such as `(help grind)` to the prover `Rule?` prompt, or by giving the Emacs command `M-x pvs-help-prover-command grind` (the Emacs command has completion—just hit the space bar or `?` to finish off, or to see the options for, a partly-typed command). Alternatively, you can give the command `M-x x-prover-commands` to create a persistent mouse-sensitive display of all prover commands, and then click the middle mouse button on `grind`. In all cases, the help given for `grind` is as follows (the vertical dots indicate material deleted for brevity).

```
(GRIND/$ &OPTIONAL (DEFS !) THEORIES REWRITES (IF-MATCH T) EXCLUDE (UPDATES? T)):
    A super-duper strategy.[a]Does auto-rewrite-defs/theories,
auto-rewrite then applies skolem!, inst?, lift-if, bddsimp, and
assert, until nothing works. Here
   ⋮
 IF-MATCH is either NIL (no instantiation), T (yes instantiation),
     ALL (all instances) or BEST (best instance) depending on which version
     of INST? is required.
```

[a] The term "super-duper" here is a reference to [ALW93].

This tells us that the optional arguments to `grind` are `defs`, `theories`, `rewrites`, `exclude`, `if-match`, and `updates?` (upper/lower case distinctions are not important). Parentheses are used to indicate default values (so the default value for `if-match` is `t`). If we wish to supply values for each of these arguments, we can just supply them in order; but if we wish to supply values only for some of them, we must indicate the ones concerned by preceding each value with a keyword derived by prefixing a colon to name of the relevant argument. From the help display, we see that the `:if-match nil` argument tells `grind` to not attempt heuristic instantiation.

Here, `(grind :if-match nil)` generates four subgoals, the first of which is the following.

```
Rule? (grind :if-match nil)
... reporting of rewrites omitted
Trying repeated Skolemization, instantiation, and if-lifting,
this yields  4 subgoals:
Cancel_assn_inv.1 :

{-1}   FORALL (a: [# pass: passenger, seat: [row, position] #]), (flt: flight):
        db!1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a))
{-2}   FORALL (a: [# pass: passenger, seat: [row, position] #]),
      (b: [# pass: passenger, seat: [row, position] #]), (flt: flight):
        db!1(flt)(a) AND db!1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b
{-3}   flt!1 = flt!2
{-4}   db!1(flt!2)(a!1)
{-5}   db!1(flt!2)(b!1)
{-6}   pass(a!1) = pass(b!1)
  |-------
{1}   (pass(b!1) = pas!1)
{2}   a!1 = b!1
```

It looks as though the obvious instantiations for the variables of formula `-2` should take care of this branch and, indeed, the sequence of proof commands `(inst? -2)(inst? -2)(prop)` discharges it and presents us with the next sequent.

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

This completes the proof of Cancel_assn_inv.1.

Cancel_assn_inv.2 :

{-1}   FORALL (a: [# pass: passenger, seat: [row, position] #]), (flt: flight):
        db!1(flt)(a) IMPLIES seat_exists(aircraft(flt), seat(a))
{-2}   FORALL (a: [# pass: passenger, seat: [row, position] #]),
      (b: [# pass: passenger, seat: [row, position] #]), (flt: flight):
        db!1(flt)(a) AND db!1(flt)(b) AND pass(a) = pass(b) IMPLIES a = b
{-3}   db!1(flt!2)(a!1)
{-4}   db!1(flt!2)(b!1)
{-5}   pass(a!1) = pass(b!1)
  |-------
{1}   flt!1 = flt!2
{2}   a!1 = b!1
```

This should follow in the same way, but we note that the `(inst?)` and `(prop)` commands used on the previous sequent are subsumed by the functionality of `(grind)`, so we try `(grind)` instead. It works—and `(grind)` also takes care of the other two proof branches as well. We have now finished the proof, and can inspect the saved proof description using `M-x edit-proof`. This looks as follows.

```
(""
 (GRIND :IF-MATCH NIL)
 (("1" (INST? -2) (INST? -2) (PROP)) ("2" (GRIND)) ("3" (GRIND))
  ("4" (GRIND))))
```

We conjecture that the first branch of the proof tree could also have been discharged by `(grind)`, yielding a more uniform proof. We check this conjecture by starting the proof again, and giving the following single proof command.

```
(then (grind :if-match nil)(grind))
```

In this command, `then` is a proof sequencing strategy—it successively applies the proofs steps supplied as its argument. More particularly, it first applies its first argument, then recursively applies the rest of its arguments to the subgoals so created.

If we examine (with `M-x edit-proof`) the proof saved after running this strategy, we see that it has the following form.

```
(""
 (GRIND :IF-MATCH NIL)
 (("1" (GRIND)) ("2" (GRIND)) ("3" (GRIND)) ("4" (GRIND))))
```

Notice that this is not the command we typed in, but the collection of proof steps that it generated. If we had wished to save the `(then...)` form, we could have run the proof as an atomic step by wrapping it in an `apply` as follows.

```
(apply (then (grind :if-match nil)(grind))).
```

Most of the standard strategies of PVS automatically run and are saved in this atomic manner; they can be caused to run in the expanded, verbose manner by appending a `$` symbol to their names. This can be useful if you want to know how a strategy such as `grind` actually performed a proof at the level of primitive steps. In the present example, we can discover the primitive steps generated by the applications of `grind` by giving the following proof command.

```
(then (grind$ :if-match nil)(grind$)).
```

The result is shown in Figure 2.1. It is sometimes difficult to follow the structure in a fairly long proof such this, and a graphical display can be very helpful. PVS can produce such graphical displays of proof-trees by the command `M-x x-show-current-proof` whilst in the prover, or by `M-x x-show-proof` to see the saved proof for a formula under the cursor. The graphical display can be adjusted interactively and saved as a postscript file that can then be included in a LaTeX document by commands such as the following (using the LaTeX `epsf` style option).

```
\begin{figure}[htp]
\begin{center}
\leavevmode
\epsfxsize=.65\hsize
\epsfbox{ops_Cancel_assn_inv.ps}
\end{center}
\caption{\label{proof-tree}Proof Tree for Theorem {\tt Cancel\_assn\_inv}}
\end{figure}
```

The result is shown in figure 2.2.

We might wonder why the sequence `(then...)` of `grind` steps succeeds in proving this theorem, when `(grind)` on its own does not. The explanation is that `grind` is a fairly simple heuristic that iteratively simplifies and looks for instantiations—and sometimes it finds the wrong instantiations because it looks for them too early. The strategy `(then (grind :if-match nil)(grind))` effectively postpones the search for instantiations until all simplification has been completed. We might consider this a sufficiently useful strategy that we would like to save it for future use. We can do this by placing the following lisp code in the file `pvs-strategies` in our current or home directories.

```
(defstep lazy-grind ( )
   (then (grind$ :if-match nil)(grind$))
  "Equiv. to (grind) with instantiations postponed until after simplification."
  "By skolemization, if-lifting, simplification and instantiation")
```

Notice that the inner calls to `grind` use the `$` (verbose) form—this is so the command `lazy-grind$` will cause those inner commands to run in their expanded form. The normal (non-`$`) form of the `lazy-grind` command will automatically run its inner commands in the terse form. Once this definition of `lazy-grind` has been placed in our `pvs-strategies` file, PVS will automatically load it the next time the prover is called, and our theorem can be proved with just the single command `(lazy-grind)`.

Our definition of `lazy-grind` is really rather crude. If we wish to create a strategy that will be generally useful, we should pay attention to its efficiency and generality. The current definition is rather inefficient because the second invocation of `grind` will repeat the installation of automatic rewrite rules performed by the first. If we examine the definition of `grind` (by clicking right on the `M-x x-prover-commands` display, or by the command `M-x help-pvs-prover-command grind`), we will see that its inner loop is performed by the command `reduce`, so we should use this in place of the second invocation of `grind`.

Also, our current definition of `lazy-grind` is lacking in generality—because it does not provide a way to pass arguments to `grind`. We can rectify these deficiencies in the following improved definition.

```
(""
 (AUTO-REWRITE-DEFS :ALWAYS? T)
 (ASSERT)
 (SKOLEM-TYPEPRED)
 (FLATTEN)
 (ASSERT)
 (BDDSIMP)
 (("1"
   (SKOLEM-TYPEPRED)
   (FLATTEN)
   (LIFT-IF)
   (ASSERT)
   (BDDSIMP)
   (("1"
     (REPLACE*)
     (ASSERT)
     (BDDSIMP)
     (INST? :IF-MATCH T)
     (REPLACE*)
     (ASSERT)
     (INST? :IF-MATCH T)
     (REPLACE*)
     (ASSERT)
     (INST? :IF-MATCH T)
     (REPLACE*)
     (ASSERT))
    ("2"
     (ASSERT)
     (INST? :IF-MATCH T)
     (REPLACE*)
     (ASSERT)
     (INST? :IF-MATCH T)
     (REPLACE*)
     (ASSERT)
     (INST? :IF-MATCH T)
     (REPLACE*)
     (ASSERT))))
  ("2"
   (SKOLEM-TYPEPRED)
   (FLATTEN)
   (LIFT-IF)
   (ASSERT)
   (BDDSIMP)
   (("1" (REPLACE*) (ASSERT) (INST? :IF-MATCH T) (REPLACE*) (ASSERT))
    ("2" (ASSERT) (INST? :IF-MATCH T) (REPLACE*) (ASSERT))))))
```

Figure 2.1: Expanded Proof for `Cancel_assn_inv`

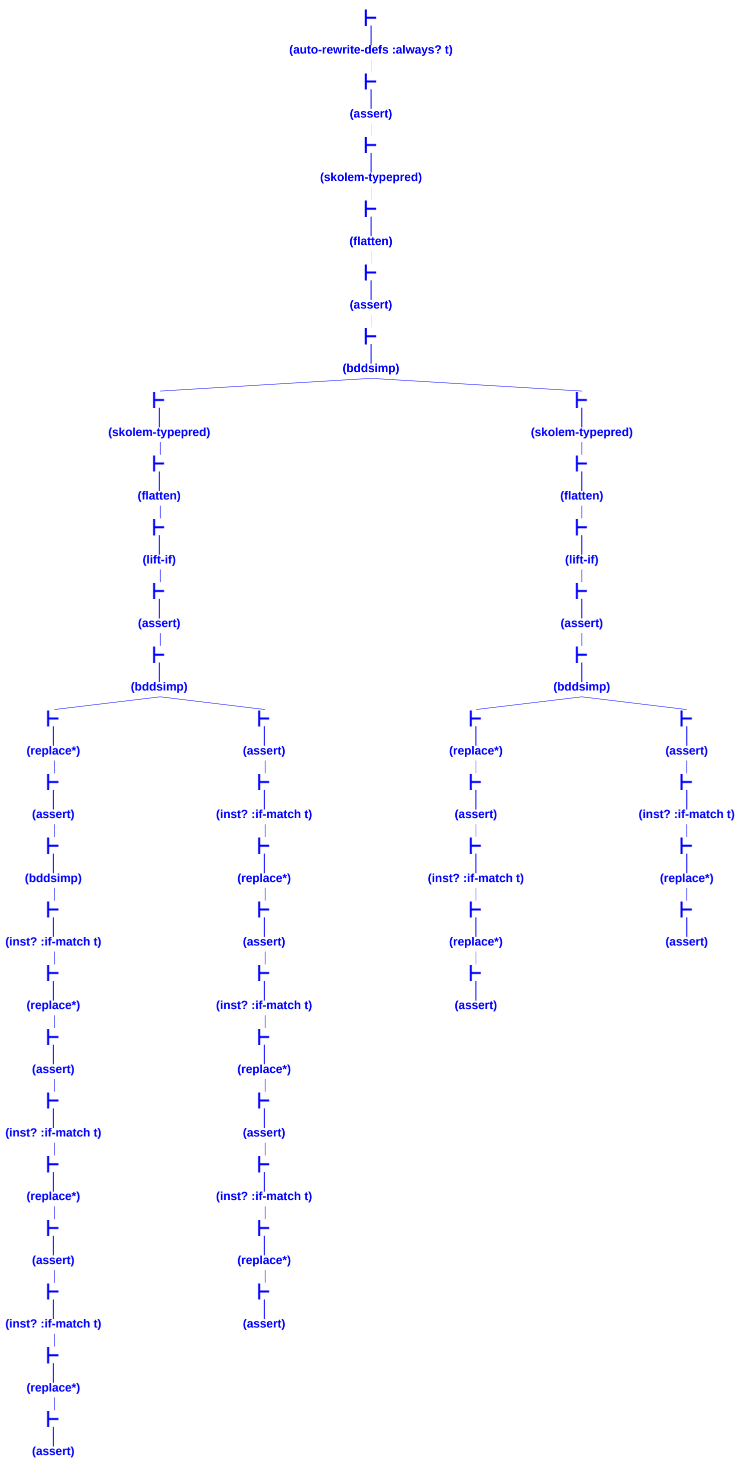


Figure 2.2: Proof Tree for Theorem `Cancel_assn_inv`

```
(defstep lazy-grind  (&optional (if-match t) (defs !)
                       rewrites theories exclude (updates? t))
   (then
     (grind$ :if-match nil :defs defs :rewrites rewrites :theories theories
             :exclude exclude :updates? updates?)
     (reduce$ :if-match if-match :updates? updates?))
  "Equiv. to (grind) with instantiations postponed until after simplification."
  "By skolemization, if-lifting, simplification and instantiation")
```

The identifiers following the `&optional` in the formal argument list indicate the keyword arguments to this command (with optional default values in parentheses—for example, `defs` defaults to `!`, but `rewrites` has no default (or, rather, defaults to `nil`)). When the `lazy-grind` proof command is invoked, the actual arguments supplied become the values of these identifiers. The two strings appearing at the end of the definition are, respectively, the help string and the commentary printed whenever the command is invoked.

2.5.3 MAe

Whereas the previous proof showed `grind` being too eager in seeking instantiations, the proof of the next theorem (`MAe`) shows the other aspect of its occasional overeagerness—this time in rewriting. The theorem states that making a new seat assignment will retain the property that all seat assignments on that flight are for seats that really do exist on that type of aircraft.

```
  MAe: THEOREM
        existence(db) IMPLIES existence(Make_assn(flt,pas,pref,db))
```

The proof begins by attacking the theorem with `(grind)`. This gives us five much reduced goals.

```
Rule? (grind)
... reporting of rewrites omitted
Trying repeated skolemization, instantiation, and if-lifting,
this yields  5 subgoals:
MAe.1 :

{-1}   meets_pref(aircraft(flt!2), seat!1, pref!1)
{-2}   flt!1 = flt!2
{-3}   (# seat := Next_seat(db!1, flt!2, pref!1), pass := pas!1 #) = a!1
  |-------
{1}   db!1(flt!2)(a!1)
{2}   seat_exists(aircraft(flt!2), seat(a!1))

Rule?
```

We have several `Next_seat` axioms relating to the `seat_exists` function, but `grind` has rewritten the sequent too far for us to see which axiom is needed. So, we go back to the beginning (with `(undo)`), and try again, but only rewriting the two terms that occur in the `MAe` theorem. The `:defs nil` argument tells `grind` not to install any definitions as automatic rewrites, while the `:rewrites` keyword introduces a list of functions (or, in general, conditional equations) that we do want to be rewritten automatically.

```
Rule? (grind :defs nil :rewrites ("existence" "Make_assn"))
... reporting of rewrites omitted
Trying repeated skolemization, instantiation, and if-lifting,
this simplifies to:
MAe :

{-1}   member(a!1, Make_assn(flt!1, pas!1, pref!1, db!1)(flt!2))
  |-------
{1}   member(a!1, db!1(flt!2))
{2}   seat_exists(aircraft(flt!2), seat(a!1))
```

Note that although `Make_assn` was in the rewrites list, it still appears in the resultant sequent. This is because the definition of `Make_assn` is a *conditional* rewrite (it has a top-level `if-then-else`), whose condition cannot be immediately reduced to `true` or `false`. Rewrites can be made unconditional by placing them in a nested list. For example, the following makes `Make_assn` an unconditional rewrite.

```
Rule? (grind :defs nil :rewrites ("existence" ("Make_assn")))
```

In the present case, however, we can press on by expanding this function with `(expand "Make_assn")` (`TAB e` with the cursor on "`Make_assn`"), then simplifying the resulting `if-then-else` with `(lift-if)` and `(prop)` (the effect of the latter two can be obtained by the more powerful command `(smash)`).

```
Rule? (then (expand "Make_assn")(smash))
... reporting of rewrites omitted
Repeatedly simplifying with BDDs, decision procedures, rewriting,
and if-lifting,
this simplifies to
MAe :

{-1}   flt!1 = flt!2
{-2}   member(a!1,
             add((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #),
                 db!1(flt!1)))
  |-------
[1]   member(a!1, db!1(flt!2))
{2}   pref_filled(db!1, flt!1, pref!1)
{3}   pass_on_flight(pas!1, flt!1, db!1)
[4]   seat_exists(aircraft(flt!2), seat(a!1))

Rule?
```

At this point, it looks as though the axiom `Next_seat_ax` will supply the information necessary to complete the proof. We introduce this axiom by the command `(use "Next_seat_ax")` which extends the `lemma` command by attempting heuristic instantiation of the formula concerned.

```
Rule? (use "Next_seat_ax")
Using lemma Next_seat_ax,,
this simplifies to:
MAe :

{-1}   NOT pref_filled(db!1, flt!1, pref!1)
          IMPLIES seat_exists(aircraft(flt!1), Next_seat(db!1, flt!1, pref!1))
[-2]   flt!1 = flt!2
[-3]   member(a!1,
             add((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #),
                 db!1(flt!1)))
  |-------
[1]   member(a!1, db!1(flt!2))
[2]   pref_filled(db!1, flt!1, pref!1)
[3]   pass_on_flight(pas!1, flt!1, db!1)
[4]   seat_exists(aircraft(flt!2), seat(a!1))
```

It looks like this should follow by trivial reasoning and expansion of definitions, so we hit it with `(grind)` and, indeed, the proof completes.

Now that we have proved this theorem, it will be worth going back to see if we can find a shorter, more automatic and potentially more robust proof. The proof basically came down to straightforward `grind`-like steps, plus use of the axiom `Next_seat_ax`. Such examples can often be proved by first adding the necessary axioms or lemmas to the sequent, and then letting `grind` go to work. This is the case here: the following command proves the theorem.

```
(then (lemma "Next_seat_ax")(grind))
```

As with our previous proof, it may be worth saving this strategy for future use. In fact, it turns out that this strategy can usefully subsume the functionality of `lazy-grind` to yield a strategy called `stew` (because it adds a set of lemmas to the pot and lets them simmer with `grind`).

```
(defstep stew (&optional lazy-match (if-match t) (defs !) rewrites theories
               exclude (updates? t) &rest lemmas)
   (then
     (if lemmas
       (let ((lemmata (if (listp lemmas) lemmas (list lemmas)))
             (x `(then ,@(loop for lemma in lemmata append
                                  `((skosimp*)(use ,lemma))))))
          x)
       (skip))
     (if lazy-match
       (then (grind$ :if-match nil :defs defs :rewrites rewrites
                     :theories theories :exclude exclude :updates? updates?)
             (reduce$ :if-match if-match :updates? updates?))
       (grind$ :if-match if-match :defs defs :rewrites rewrites
                     :theories theories :exclude exclude :updates? updates?)))
"Does a combination of (lemma) and (grind)."
  "~%Grinding away with the supplied lemmas,")
```

The tricky part of this strategy is the `let` construct that builds a list `x` of `skosimp*` and `use` commands from the supplied list of lemmas, and then executes it. Explanation of this construct can be found in the PVS strategy manual (forthcoming). The `lazy-grind` capability is achieved by the `lazy-match` argument to `stew` (i.e., `(stew :lazy-match t)` is equivalent to `(lazy-grind)`). Thus, the previous proof can now be achieved by `(stew :lazy-match t)` and the present one by `(stew :lemmas "Next_seat_ax")`.

2.5.4 MAu

```
  MAu: THEOREM
        uniqueness(db) IMPLIES uniqueness(Make_assn(flt,pas,pref,db))
```

The theorem `MAu` asserts `Make_assn` preserves the property that the same passenger is not booked twice onto the same flight. Since the theorem looks pretty obvious, we start off with `(grind)`.

```
Rule? (grind)
... reporting of rewrites omitted
Trying repeated skolemization, instantiation, and if-lifting,
this yields  2 subgoals:
MAu.1 :

{-1}   meets_pref(aircraft(flt!1), seat!1, pref!1)
{-2}   db!1(flt!2)(a!1)
{-3}   db!1(flt!2)(b!1)
{-4}   pass(a!1) = pass(b!1)
  |-------
{1}   db!1(flt!1)(b!1)
{2}   flt!1 = flt!2
{3}   a!1 = b!1
```

Comparison of formulas `-2` and `-3` with `1` suggests that the strategy found the wrong instantiations. We speculate that it will do better if we postpone these using `(stew :lazy-match t)`. And, indeed, this proves the theorem.

2.5.5 `Make_assn_inv`

```
Make_assn_inv: THEOREM
      db_invariant(db) IMPLIES db_invariant(Make_assn(flt,pas,pref,db))
```

Trying `(grind)` immediately yields 6 subgoals, which doesn't seem very promising. We undo this step and restrict rewriting to just the `db_invariant` that appears in the statement of the theorem.

```
Rule? (grind :defs nil :rewrites ("db_invariant"))
... reporting of rewrites omitted
Trying repeated skolemization, instantiation, and if-lifting,
this yields  2 subgoals:
Make_assn_inv.1 :

{-1}   existence(db!1)
{-2}   uniqueness(db!1)
  |-------
{1}   uniqueness(Make_assn(flt!1, pas!1, pref!1, db!1))
```

This is clearly `MAu`, so we rewrite with that as a lemma.

```
Rule? (rewrite "MAu")
Rewriting using MAu,

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

The other subgoal suffers a similar fate when rewritten with `MAe`.

The proof can be reduced to a single step by including MAu and MAe among the rewrites.

```
(GRIND :DEFS NIL :REWRITES ("db_invariant" "MAu" "MAe"))
```

2.5.6 `initial_state_inv`

```
initial_state_inv: THEOREM
      db_invariant(initial_state)
```

The proof of `initial_state_inv` is trivial and is disposed of by `(grind)`.

2.5.7 Cancel_inv_one_per_seat

`Cancel_inv_one_per_seat` asserts `Cancel_assn` maintains the property that each seat on an aircraft has no more than one occupant.

```
Cancel_inv_one_per_seat: THEOREM
        one_per_seat(db) IMPLIES one_per_seat(Cancel_assn(flt,pas,db))
```

The proof is very similar to the previous one and is dispatched by `(grind)`.

2.5.8 Make_inv_one_per_seat

```
  Make_inv_one_per_seat: THEOREM
        one_per_seat(db) IMPLIES one_per_seat(Make_assn(flt,pas,pref,db))
```

This theorem is very similar to `MAe`: whereas `MAe` ensures that only seats that exist are assigned on a flight, this theorem ensures that a seat is not assigned twice on the same flight.

We speculate that the proof of this theorem will be similar to that of `MAe`, except that the axiom `Next_seat_ax_2` (which says that `Next_seat` does not assign already assigned seats) will be needed instead of `Next_seat_ax` (which says that `Next_seat` does not assign nonexistent seats). Accordingly, we try the proof command `(stew :lemmas "Next_seat_ax_2")` and obtain the following result.

```
Rule? (stew :lemmas "Next_seat_ax_2")
... reporting of rewrites omitted
Grinding away with the supplied lemmas,,
this simplifies to:
Make_inv_one_per_seat :

{-1}   meets_pref(aircraft(flt!1), seat!1, pref!1)
{-2}   db!1(flt!2)(a!1)
{-3}   db!1(flt!2)(b!1)
{-4}   seat(a!1) = seat(b!1)
  |-------
{1}   db!1(flt!1)(a!1)
{2}   flt!1 = flt!2
{3}   a!1 = b!1
```

As in the proof of `MAu`, comparison of formulas `-2` and `-3` with `1` suggests that the strategy has found the wrong instantiations. As before, we speculate that it will do better if we postpone these using `(stew :lemmas "Next_seat_ax_2" :lazy-match t)`. And, indeed, this proves the theorem.

2.5.9 Initial_one_per_seat

```
  initial_one_per_seat: THEOREM
        one_per_seat(initial_state)
```

This is trivial and is discharged with (grind).

2.5.10 Make_Cancel

This theorem asserts that Cancel_assn undoes the operation of Make_assn—i.e., making a new reservation and then canceling it leaves the state unchanged.

```
Make_Cancel: THEOREM
        NOT pass_on_flight(pas,flt,db) IMPLIES
                Cancel_assn(flt,pas,Make_assn(flt,pas,pref,db)) = db
```

Having noticed that the formulas in this specification tend to allow over-rewriting, we start the proof with grind restricted to the functions appearing in the statement of the theorem.

```
Rule? (grind :defs nil :rewrites ("pass_on_flight" "Cancel_assn" "Make_assn"))
...  reporting of rewrites omitted
Trying repeated skolemization, instantiation, and if-lifting,
this simplifies to:

Make_Cancel :

  |-------
{1}   EXISTS (a: [# pass: passenger, seat: [row, position] #]):
        pass(a) = pas!1 AND member(a, db!1(flt!1))
{2}   Make_assn(flt!1, pas!1, pref!1, db!1)
        WITH [(flt!1) :=
                a:
                  [# pass: passenger,
                     seat: [row, position] #]
                |
                member(a,
                       Make_assn(flt!1, pas!1,
                                 pref!1, db!1)(flt!1))
                    AND pass(a) /= pas!1]
          = db!1
```

Notice that Make_assn has not been rewritten, as it would not simplify the sequent.

Now formula 2 requires demonstration that two flt_dbs are equal. These databases are functions, so to prove them equal we must appeal to the principle of extensionality—so that it will then be enough to show that the values of the functions are equal for all arguments.

```
Rule? (apply-extensionality :hide? t)
Applying extensionality,
this simplifies to:
Make_Cancel :

  |-------
{1}   Make_assn(flt!1, pas!1, pref!1, db!1)
        WITH [(flt!1) :=
                a:
                  [# pass: passenger,
                     seat: [row, position] #]
                |
                member(a,
                       Make_assn(flt!1, pas!1,
                                 pref!1, db!1)(flt!1))
                    AND pass(a) /= pas!1](x!1)
          = db!1(x!1)
[2]   EXISTS (a: [# pass: passenger, seat: [row, position] #]):
        pass(a) = pas!1 AND member(a, db!1(flt!1))
```

The `hide? t` argument simply hides the original form of formula `1`, resulting in a less cluttered sequent.

Now the values of a `flt_db` are `flight_assignments`, which are sets of `seat_assignments`. To show two sets are equal, we must again appeal to extensionality, so that it will be enough to show that they have the same members.

```
rule? (apply-extensionality :hide? t)
Applying extensionality,
this simplifies to:
Make_Cancel :

  |-------
{1}   Make_assn(flt!1, pas!1, pref!1, db!1)
        WITH [(flt!1) :=
                a:
                  [# pass: passenger,
                     seat: [row, position] #]
                |
                member(a,
                       Make_assn(flt!1, pas!1,
                                 pref!1, db!1)(flt!1))
                    AND pass(a) /= pas!1](x!1)(x!2)
          = db!1(x!1)(x!2)
[2]   EXISTS (a: [# pass: passenger, seat: [row, position] #]):
        pass(a) = pas!1 AND member(a, db!1(flt!1))
```

(We could have done both these steps with `(repeat (apply-extensionality :hide? t))`.) Optimistically, we try `(grind)` at this point.

```
Rule? (grind)
...  reporting of rewrites omitted
Make_Cancel :

{-1}   flt!1 = x!1
{-2}   (pass(x!2) = pas!1)
{-3}   db!1(x!1)(x!2)
  |-------
{1}   db!1(x!1)((# seat := Next_seat(db!1, x!1, pref!1), pass := pas!1 #))
```

Comparison of formulas `-3` and `1` suggest that an incorrect substitution has been found. We undo this step and try postponing the substitutions with `(stew :lazy-match t)`, and this time the proof succeeds.

We use this proof as an example of the LaTeX facilities of PVS. The command `M-x latex-proof` generates a LaTeX file which produces the following output.[8]

Verbose proof for `Make_Cancel`.

`Make_Cancel`:

$$
\begin{array}{|ll}
\hline
\{1\} & (\forall\,(\mathrm{db}: \mathrm{flt\_db},\ \mathrm{flt}: \mathrm{flight},\ \mathrm{pas}: \mathrm{passenger},\ \mathrm{pref}: \mathrm{preference}):\\
& \quad \neg\ \mathrm{pass\_on\_flight}(\mathrm{pas},\ \mathrm{flt},\ \mathrm{db})\\
& \qquad \supset\ \mathrm{Cancel\_assn}(\mathrm{flt},\ \mathrm{pas},\ \mathrm{Make\_assn}(\mathrm{flt},\ \mathrm{pas},\ \mathrm{pref},\ \mathrm{db})) = \mathrm{db})
\end{array}
$$

Trying repeated skolemization, instantiation, and if-lifting,

`Make_Cancel`:

$$
\begin{array}{|ll}
\hline
\{1\} & \exists\,(a: [\#\ \mathrm{pass}: \mathrm{passenger},\ \mathrm{seat}: [\mathrm{row},\ \mathrm{position}]\ \#]):\\
& \quad \mathrm{pass}(a) = \mathrm{pas!1} \ \wedge\ a \in \mathrm{db!1}(\mathrm{flt!1})\\
\{2\} & \mathrm{Make\_assn}(\mathrm{flt!1},\ \mathrm{pas!1},\ \mathrm{pref!1},\ \mathrm{db!1})\\
& \quad \text{WITH } [(\mathrm{flt!1}) :=\\
& \qquad\qquad \{a: [\#\ \mathrm{pass}: \mathrm{passenger},\ \mathrm{seat}: [\mathrm{row},\ \mathrm{position}]\ \#]\\
& \qquad\qquad\ \ |\\
& \qquad\qquad a \in\\
& \qquad\qquad\ \ \mathrm{Make\_assn}(\mathrm{flt!1},\ \mathrm{pas!1},\\
& \qquad\qquad\ \ \ \ \mathrm{pref!1},\ \mathrm{db!1})(\mathrm{flt!1})\\
& \qquad\qquad\ \ \ \ \ \ \wedge\ \mathrm{pass}(a) \neq \mathrm{pas!1}\}]\\
& \quad = \mathrm{db!1}
\end{array}
$$

Applying extensionality,

[8] Giving an argument to the command (i.e., prefixing it with `C-u`) creates a terse proof, but as this proof is so short, there is no difference in this case.

`Make_Cancel`:

|-------
{1} Make_assn(flt!1, pas!1, pref!1, db!1)
    WITH [(flt!1) :=
        {a : [# pass : passenger, seat : [row, position] #]
        |
        a ∈
          Make_assn(flt!1, pas!1,
            pref!1, db!1)(flt!1)
              ∧ pass(a) ≠ pas!1}](x′)
    = db!1(x′)
{2} ∃ (a : [# pass : passenger, seat : [row, position] #]) :
    pass(a) = pas!1 ∧ a ∈ db!1(flt!1)

Applying extensionality,

`Make_Cancel`:

|-------
{1} Make_assn(flt!1, pas!1, pref!1, db!1)
    WITH [(flt!1) :=
        {a : [# pass : passenger, seat : [row, position] #]
        |
        a ∈
          Make_assn(flt!1, pas!1,
            pref!1, db!1)(flt!1)
              ∧ pass(a) ≠ pas!1}](x′)(x″)
    = db!1(x′)(x″)
{2} ∃ (a : [# pass : passenger, seat : [row, position] #]) :
    pass(a) = pas!1 ∧ a ∈ db!1(flt!1)

Grinding away with the supplied lemmas,,

This completes the proof of `Make_Cancel`.

Q.E.D.

2.5.11 `Cancel_putative`

```
Cancel_putative: THEOREM
        NOT (EXISTS (a: seat_assignment):
           member(a,Cancel_assn(flt,pas,db)(flt)) AND pass(a) = pas)
```

This is trivial and is discharged with `(grind)`.

2.5.12 `Make_putative`

```
Make_putative: THEOREM
      NOT pref_filled(db, flt, pref) IMPLIES
        (EXISTS (x: seat_assignment):
          member(x, Make_assn(flt, pas, pref, db)(flt)) AND pass(x) = pas)
```

This is also trivial and is discharged with `(grind)`.

2.5.13 Lookup_putative

```
Lookup_putative: THEOREM
      NOT (pref_filled(db, flt, pref) OR pass_on_flight(pas,flt,db)) IMPLIES
    meets_pref(aircraft(flt),
               Lookup(flt, pas, Make_assn(flt,pas,pref,db)),
               pref)
```

We begin this proof as usual with (grind), and are left with two subgoals to prove. The first subgoal is the following.

```
Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,
this yields  2 subgoals:
Lookup_putative.1 :

{-1}    meets_pref(aircraft(flt!1), seat!1, pref!1)
{-2}    meets_pref(aircraft(flt!1), seat!2, pref!1)
  |-------
{1}    Next_seat(db!1, flt!1, pref!1) = seat!1
{2}    db!1(flt!1)((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #))
{3}    meets_pref(aircraft(flt!1),
                 seat(epsilon({a: seat_assignment |
                              ((# seat :=
                                    Next_seat(db!1,
                                              flt!1, pref!1),
                                 pass := pas!1 #)
                                  = a
                                  OR db!1(flt!1)(a))
                                 AND pass(a) = pas!1})),
                 pref!1)
```

Notice that this subgoal contains two Skolemized `seat` variables: `seat!1` and `seat!2`. This suggests that `(grind)` has been over-eager in instantiation, as we saw before in the proof of `Cancel_asn_inv`, so we start again and tell `grind` not to perform heuristic instantiation.

```
Rule? (grind :if-match nil)
... reporting of rewrites omitted
Trying repeated skolemization, instantiation, and if-lifting,
this yields  2 subgoals:
Lookup_putative.1 :

{-1}   meets_pref(aircraft(flt!1), seat!1, pref!1)
{-2}   FORALL (seat: [row, position]):
        meets_pref(aircraft(flt!1), seat, pref!1) IMPLIES
           (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat)
  |-------
{1}   (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat!1)
{2}   EXISTS (a: seat_assignment): pass(a) = pas!1 AND db!1(flt!1)(a)
{3}   meets_pref(aircraft(flt!1),
                seat(epsilon({a: seat_assignment |
                               db!1(flt!1)(a) AND pass(a) = pas!1})),
                pref!1)
```

Here, formula `-2`, with substitution `seat!1` would relate `-1` and `1`. To perform the instantiation, and simplification we just issue another `(grind)` command which dispatches this subgoal. We are now left with our second subgoal.

```
Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of Lookup_putative.1.

Lookup_putative.2 :

{-1}    meets_pref(aircraft(flt!1), seat!1, pref!1)
{-2}    meets_pref(aircraft(flt!1), seat!2, pref!1)
  |-------
{1}    (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat!1)
{2}    EXISTS (a: seat_assignment): pass(a) = pas!1 AND db!1(flt!1)(a)
{3}    (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat!2)
{4}    meets_pref(aircraft(flt!1),
                seat(epsilon({a: seat_assignment |
                             ((# seat :=
                                   Next_seat(db!1,
                                             flt!1, pref!1),
                                 pass := pas!1 #)
                                  = a
                                  OR db!1(flt!1)(a))
                                 AND pass(a) = pas!1})),
                pref!1)
```

Clearly here we need to relate either formula `-1` or `-2` with formula `4`. As `4` involves `epsilon`, we introduce the axiom `epsilon_ax` from the prelude. As the `epsilons` prelude theory is parameterized, we must supply the appropriate parameter here.

```
Rule? (use "epsilon_ax[seat_assignment]")
Using lemma epsilon_ax[seat_assignment],
this simplifies to:
Lookup_putative.2 :

{-1}    (EXISTS (x: seat_assignment):
         ({a: seat_assignment |
          ((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #) = a
               OR db!1(flt!1)(a))
              AND pass(a) = pas!1})(x))
          =>
        ({a: seat_assignment |
         ((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #) = a
              OR db!1(flt!1)(a))
             AND pass(a)
               = pas!1})(epsilon({a: seat_assignment |
                                  ((# seat :=
                                        Next_seat(db!1,
                                                  flt!1, pref!1),
                                      pass := pas!1 #)
                                       = a
                                       OR db!1(flt!1)(a))
                                      AND pass(a) = pas!1}))
[-2]    meets_pref(aircraft(flt!1), seat!1, pref!1)
[-3]    meets_pref(aircraft(flt!1), seat!2, pref!1)
  |-------
[1]    (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat!1)
[2]    EXISTS (a: seat_assignment): pass(a) = pas!1 AND db!1(flt!1)(a)
[3]    (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat!2)
[4]    meets_pref(aircraft(flt!1),
                 seat(epsilon({a: seat_assignment |
                               ((# seat :=
                                     Next_seat(db!1,
                                               flt!1, pref!1),
                                   pass := pas!1 #)
                                    = a
                                    OR db!1(flt!1)(a))
                                   AND pass(a) = pas!1})),
                 pref!1)
```

We use `(ground)` to simplify the expression in `-1`.

```
Rule? (ground)
Applying propositional simplification and decision procedures,
this yields  2 subgoals:
Lookup_putative.2.1 :

{-1}    ((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #)
           =
         epsilon({a: seat_assignment |
                 ((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #)
                      = a
                      OR db!1(flt!1)(a))
                     AND pass(a) = pas!1})
           OR
         db!1(flt!1)(epsilon({a: seat_assignment |
                             ((# seat :=
                                   Next_seat(db!1,
                                             flt!1, pref!1),
                                 pass := pas!1 #)
                                  = a
                                  OR db!1(flt!1)(a))
                                 AND pass(a) = pas!1})))
          AND
        pass(epsilon({a: seat_assignment |
                     ((# seat := Next_seat(db!1, flt!1, pref!1),
                         pass := pas!1 #)
                          = a
                          OR db!1(flt!1)(a))
                         AND pass(a) = pas!1}))
            = pas!1
[-2]    meets_pref(aircraft(flt!1), seat!1, pref!1)
[-3]    meets_pref(aircraft(flt!1), seat!2, pref!1)
  |-------
[1]    (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat!1)
[2]    EXISTS (a: seat_assignment): pass(a) = pas!1 AND db!1(flt!1)(a)
[3]    (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat!2)
[4]    meets_pref(aircraft(flt!1),
                 seat(epsilon({a: seat_assignment |
                              ((# seat :=
                                    Next_seat(db!1,
                                              flt!1, pref!1),
                                  pass := pas!1 #)
                                   = a
                                   OR db!1(flt!1)(a))
                                  AND pass(a) = pas!1})),
                 pref!1)
```

We are still left with a complex propositional expression in `-1`, as `(ground)` does not re-apply propositional simplification after the decision procedures. We thus apply `(ground)` again (we could have done `(repeat (ground))` to the same effect).

```
Rule? (ground)
Applying propositional simplification and decision procedures,
this yields  2 subgoals:
Lookup_putative.2.1.1 :

{-1}    (# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #)
          =
        epsilon({a: seat_assignment |
                ((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #)
                     = a
                     OR db!1(flt!1)(a))
                    AND pass(a) = pas!1})
{-2}    pass(epsilon({a: seat_assignment |
                   ((# seat := Next_seat(db!1, flt!1, pref!1),
                       pass := pas!1 #)
                        = a
                        OR db!1(flt!1)(a))
                       AND pass(a) = pas!1}))
          = pas!1
[-3]    meets_pref(aircraft(flt!1), seat!1, pref!1)
[-4]    meets_pref(aircraft(flt!1), seat!2, pref!1)
  |-------
[1]    (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat!1)
[2]    EXISTS (a: seat_assignment): pass(a) = pas!1 AND db!1(flt!1)(a)
[3]    (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat!2)
[4]    meets_pref(aircraft(flt!1),
                 seat(epsilon({a: seat_assignment |
                              ((# seat :=
                                    Next_seat(db!1,
                                              flt!1, pref!1),
                                  pass := pas!1 #)
                                   = a
                                   OR db!1(flt!1)(a))
                                  AND pass(a) = pas!1})),
                 pref!1)
```

Now we can see that formula `-1`, if used as a right-to-left replace will remove the occurrence of `epsilon` in formulas `-2` and `4`.

```
Rule? (replace -1 :dir rl :hide? t)
Replacing using formula -1,
this simplifies to:
Lookup_putative.2.1.1 :

{-1}    pass((# seat := Next_seat(db!1, flt!1, pref!1), pass := pas!1 #)) = pas!1
[-2]    meets_pref(aircraft(flt!1), seat!1, pref!1)
[-3]    meets_pref(aircraft(flt!1), seat!2, pref!1)
  |-------
[1]    (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat!1)
[2]    EXISTS (a: seat_assignment): pass(a) = pas!1 AND db!1(flt!1)(a)
[3]    (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat!2)
{4}    meets_pref(aircraft(flt!1),
                 seat((# seat := Next_seat(db!1, flt!1, pref!1),
                         pass := pas!1 #)),
                 pref!1)
```

Notice, in formula 4, the application of the field selector `seat`, to a record whose fields are given explicitly. This can be simplified via beta reduction.

```
Rule? (beta)
Applying beta-reduction,
this simplifies to:
Lookup_putative.2.1.1 :

{-1}    pas!1 = pas!1
[-2]    meets_pref(aircraft(flt!1), seat!1, pref!1)
[-3]    meets_pref(aircraft(flt!1), seat!2, pref!1)
  |-------
[1]    (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat!1)
[2]    EXISTS (a: seat_assignment): pass(a) = pas!1 AND db!1(flt!1)(a)
[3]    (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat!2)
{4}    meets_pref(aircraft(flt!1), Next_seat(db!1, flt!1, pref!1), pref!1)
```

Now we must appeal to the specification again, to show equivalence between either formulas `-2` or `-3` and formula `4`. The appropriate axiom is `Next_seat_ax_3`, which we introduce with a `(use)` command.

```
Rule? (use "Next_seat_ax_3")
Using lemma Next_seat_ax_3,
this simplifies to:
Lookup_putative.2.1.1 :

{-1}    NOT pref_filled(db!1, flt!1, pref!1)
          IMPLIES
        meets_pref(aircraft(flt!1), Next_seat(db!1, flt!1, pref!1), pref!1)
[-2]    pas!1 = pas!1
[-3]    meets_pref(aircraft(flt!1), seat!1, pref!1)
[-4]    meets_pref(aircraft(flt!1), seat!2, pref!1)
  |-------
[1]    (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat!1)
[2]    EXISTS (a: seat_assignment): pass(a) = pas!1 AND db!1(flt!1)(a)
[3]    (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat!2)
[4]    meets_pref(aircraft(flt!1), Next_seat(db!1, flt!1, pref!1), pref!1)
```

Applying (ground) again to simplify the sequent, we get the following.

```
Rule? (ground)
member rewrites member(a, db!1(flt!1))
  to  db!1(flt!1)(a)
pref_filled rewrites pref_filled(db!1, flt!1, pref!1)
  to  FORALL (seat: [row, position]):
        meets_pref(aircraft(flt!1), seat, pref!1)
            IMPLIES
          (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat)
Applying propositional simplification and decision procedures,
this simplifies to:
Lookup_putative.2.1.1 :

{-1}    FORALL (seat: [row, position]):
        meets_pref(aircraft(flt!1), seat, pref!1)
            IMPLIES
          (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat)
[-2]    pas!1 = pas!1
[-3]    meets_pref(aircraft(flt!1), seat!1, pref!1)
[-4]    meets_pref(aircraft(flt!1), seat!2, pref!1)
  |-------
[1]    (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat!1)
[2]    EXISTS (a: seat_assignment): pass(a) = pas!1 AND db!1(flt!1)(a)
[3]    (EXISTS (a: seat_assignment): db!1(flt!1)(a) AND seat(a) = seat!2)
[4]    meets_pref(aircraft(flt!1), Next_seat(db!1, flt!1, pref!1), pref!1)
```

Now we can see that formula `-1`, instantiated with `seat!1`, taken together with `-3` will give us formula `1`. Thus, `(inst?)` followed by `(prop)` completes this subgoal. The further remaining two subgoals are trivial, and discharged easily with `(grind)`. We thus have the following proof for `Lookup_putative`.

```
(""
 (GRIND :IF-MATCH NIL)
 (("1" (GRIND))
  ("2"
   (USE "epsilon_ax[seat_assignment]")
   (GROUND)
   (("1"
     (GROUND)
     (("1"
       (REPLACE -1 :DIR RL :HIDE? T)
       (ASSERT)
       (USE "Next_seat_ax_3")
       (GROUND)
       (INST?)
       (PROP))
      ("2" (GRIND))))
    ("2" (GRIND))))))
```

Notice that this combines `(grind)`, some of the common component steps of `(grind)`, and the `(use)` command. Earlier in this report we generated a strategy called `(stew)` that performs these functions, and indeed the following command will discharge the proof in one step.[9]

```
(stew :lemmas ("Next_seat_ax_3" "epsilon_ax[seat_assignment]"))
```

2.6 Summary

In this chapter we have presented one of the major proof tools in PVS 2—the powerful strategy `(grind)`—and have demonstrated how its behavior may be controlled where required. We have also seen how to build a yet more powerful strategy `stew` on top of `grind`, and have encountered the strategies `use` and `apply-extensionality`. We have also seen examples of the LaTeX and PostScript generating facilities provided by PVS.

Here is the output of the PVS command `M-x status-proof-theory` for theory `ops`.

```
Proof summary for theory ops
    Cancel_assn_inv.......................................proved - complete
    MAe...................................................proved - complete
    MAu...................................................proved - complete
    Make_assn_inv.........................................proved - complete
    initial_state_inv.....................................proved - complete
    Cancel_inv_one_per_seat...............................proved - complete
    Make_inv_one_per_seat.................................proved - complete
    initial_one_per_seat..................................proved - complete
```

[9] The version of this formula that uses the dependently-typed variant of `Lookup` (recall box $\boxed{3}$ on page 14) is proved by the following command: `(stew :lemmas ("Next_seat_ax_3" "choose_member"))`.

```
    Make_Cancel...........................................proved - complete
    Cancel_putative.......................................proved - complete
    Make_putative.........................................proved - complete
    Lookup_putative.......................................proved - complete
    Theory totals: 12 formulas, 12 attempted, 12 succeeded.
```

And here is the output of the PVS command `M-x show-proofs-theory` for theory `ops`.

```
Proof scripts for theory ops:

ops.Cancel_assn_inv: proved - complete

("" (STEW :LAZY-MATCH T))


ops.MAe: proved - complete

("" (STEW :LEMMAS "Next_seat_ax"))


ops.MAu: proved - complete

("" (STEW :LAZY-MATCH T))


ops.Make_assn_inv: proved - complete

("" (GRIND :DEFS NIL :REWRITES ("db_invariant" "MAu" "MAe")))


ops.initial_state_inv: proved - complete

("" (GRIND))


ops.Cancel_inv_one_per_seat: proved - complete

("" (GRIND))


ops.Make_inv_one_per_seat: proved - complete

("" (STEW :LEMMAS "Next_seat_ax_2" :LAZY-MATCH T))


ops.initial_one_per_seat: proved - complete

("" (GRIND))
```

```
ops.Make_Cancel: proved - complete

(""
 (GRIND :DEFS NIL :REWRITES
  ("Cancel_assn" "pass_on_flight" "Make_assn"))
 (APPLY-EXTENSIONALITY :HIDE? T)
 (APPLY-EXTENSIONALITY :HIDE? T)
 (STEW :LAZY-MATCH T))


ops.Cancel_putative: proved - complete

("" (GRIND))


ops.Make_putative: proved - complete

("" (GRIND))


ops.Lookup_putative: proved - complete

("" (STEW :LEMMAS ("Next_seat_ax_3" "epsilon_ax[seat_assignment]")))
```

Chapter 3

Noninterference and the Unwinding Theorem

The example undertaken in this chapter is a verification of the unwinding theorem for noninterference security policies [GM84]. One purpose of this example is to demonstrate use of PVS for a specification involving recursive functions and a proof by induction, neither of which were required for the previous example. A second purpose is to illustrate how the facilities of the PVS language and prover can be used to follow an existing mathematical development quite closely. A third purpose is to show how simple this example is: developers and users of verification systems are prone to describing how difficult are the applications they have performed—as if difficulty indicated the strength of their tool—whereas we believe that a good tool is one that makes the task easy. Using automation to reduce labor, we are able to verify this example with just seven user-supplied proof commands.

Noninterference was introduced by Goguen and Meseguer [GM82] to provide a formal foundation for the specification and analysis of security policies that are concerned with "information flow," rather than mere access control. The idea of noninterference is attractively simple: a security domain u is noninterfering with domain v if no action performed by u can influence subsequent outputs seen by v. The unwinding theorem reduces this property of sequences of actions to conditions on individual actions.

The following sections develop noninterference and give a proof of the unwinding theorem in the style of a conventional mathematical development. Each definition or proof is followed by its corresponding treatment in PVS. Our derivation is not based on the original presentation of Goguen and Meseguer, but rather follows that of Haigh and Young [HY87].

3.1 Machines

We model a computer system by a conventional finite-state automaton.

Definition 1 A *system* (or *machine*) M is composed of

- a set S of *states*, with an *initial state* $s_0 \in S$,
- a set A of *actions*, and
- a set O of *outputs*,

together with the functions *step* and *output*:

- $step: S \times A \to S$,
- $output: S \times A \to O$.

We generally use the letters $\ldots s, t, \ldots$ to denote states, letters $a, b, \ldots$ from the front of the alphabet to denote actions, and Greek letters $\alpha, \beta, \ldots$ to denote sequences of actions.

Actions can be thought of as "inputs," or "commands," or "instructions" to be performed by the machine; $step(s, a)$ denotes the next state of the system when action a is applied in state s, while $output(s, a)$ denotes the result returned by the action.

We derive a function *run*

- $run: S \times A^* \to S$,

the natural extension of *step* to sequences of actions, by the equations

$$\begin{aligned} run(s, \Lambda) &= s, \text{ and} \\ run(s, a \circ \alpha) &= step(run(s, \alpha), a), \end{aligned}$$

where Λ denotes the empty sequence and $\circ$ denotes concatenation.

Observe that this definition implies that the actions of a sequence are processed in order from right to left.

Because we will frequently use expressions of the form $output(run(s_0, \alpha), a)$, it is convenient to introduce the functions *do* and *test* to abbreviate these forms. We define these functions

- $do: A^* \to S$
- $test: A^* \times A \to O$

by the equations

$$\begin{aligned} do(\alpha) &= run(s_0, \alpha), \text{ and} \\ test(\alpha, a) &= output(do(\alpha), a). \end{aligned}$$

□

PVS Treatment

We model *states*, *actions*, and *outputs* by nonempty, uninterpreted types, and declare the initial state as a constant `s0` of type *state* and the variables s, t, and a, b as variables of the appropriate types. The functions *step* and *output* and are modeled as uninterpreted functions.

```
state, action, output: TYPE+
s0: state
s, t: VAR state
a, b: VAR action

step(s, a): state          % equivalent to step: [state, action -> state]
output(s, a): output       % equivalent to output: [state, action -> output]
```

Notice there is no confusion caused by overloading the identifier `output` to be both a type and a function.

The function *run* was defined to extend *step* to sequences of actions, so we need to find a suitable PVS representation for the notion of *sequence*. This notion is not precisely defined in semiformal mathematics, and in deciding how to represent it in PVS we need to look at how it is actually used in the example concerned. Here, we see that we need an *empty* sequence, and the operation of constructing a sequence by concatenating an individual action to another sequence (as in $a \circ \alpha$). These properties are provided by the PVS data type *list*.[1]

Lists are defined in the PVS prelude as the following abstract data type.

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

This PVS *datatype* definition says that the *constructors* for the `list` datatype are `null` and `cons`, with `null?` and `cons?` as the predicate *recognizers* for the corresponding subtypes of the list type, and that the *accessors* for a `cons`-type list are `car` and `cdr`. PVS datatypes such as this are a convenient way to specify certain data structures that are "freely generated" by a collection of constructor operations [Sha93a]. Here, *lists* are freely generated by the constructors *null* and *cons*. Similarly, the abstract datatype *stacks* is freely generated by the constructors *empty* and *push* (in fact, *stacks* and *lists* are isomorphic). *Set* provides an example of a data structure that is not freely generated (e.g., by *emptyset* and *add*), because different sequences of additions of elements can yield equivalent sets. The datatype *queues* is freely generated by *emptyqueue* and *enqueue*, but it cannot be directly defined

[1] A different representation, suitable for other interpretations of *sequence*, is a function whose domain is (an initial segment of) the natural numbers. See the PVS prelude theories `sequences` and `finite_sequences` for these representations.

by the PVS abstract datatype mechanism because it is not recursive: that is, the accessors *front* and *dequeue* are not inverses of the constructors.

PVS datatype declarations expand into several theories containing axioms and definitions that specify the properties of the datatype concerned and also define several standard functions and predicates on it. See the prelude theories `list_adt`, `list_adt_reduce` and `list_adt_map` for those generated from `list` (use the PVS command `M-X view-prelude-theory`, or `M-X vpt` for short). Datatypes also communicate information to the PVS theorem prover, and their constructors may appear in the `CASES` construct of the PVS specification language.

Given the `list` datatype, we can specify a list of actions as the type `list[action]`, and can then define the function `run` as a recursive function whose body uses the `CASES` construct to provide pattern-matching selection over the constructors of the *list* datatype.

```
action_list: TYPE = list[action]
alpha, beta: VAR action_list

run(s, beta): RECURSIVE state =
  CASES beta OF
    null: s,
    cons(a, alpha): step(run(s, alpha), a)
  ENDCASES
MEASURE length(beta)
```

All recursive functions defined in PVS must be shown to terminate by exhibiting a *measure* of their arguments that decreases across recursive calls. The built-in `length` function (from the prelude theory `list_props`) provides a suitable measure here. PVS generates the following TCC to ensure that the proposed measure does decrease in the manner required (if a measure function returns a natural number then, by default, the $<$ relation provides the notion of "decreases"). This TCC is proved automatically by the standard `(termination-tcc)` strategy.

```
% Termination TCC generated (line 20) for run

 run_TCC1: OBLIGATION
     (FORALL (b: action, beta: list[action], s, alpha):
        alpha = cons[action](b, beta)
            IMPLIES length[action](beta) < length[action](alpha));
```

More complex termination arguments may require measures on to the ordinals (see the prelude theory `ordinals`) or more elaborate notions of "decreases," such as those provided by the subterm ordering predicates `<<` generated from datatype definitions. Using this approach, an alternative way to show that the `run` function terminates is with `MEASURE beta BY <<`. This generates the following TCC, which is proved automatically by the default strategy.

```
% Termination TCC generated (line 27) for  run

 run_TCC2: OBLIGATION
    (FORALL (hd: action, tl: list[action], beta, s):
       beta = cons[action](a, alpha) IMPLIES alpha << beta);
```

However, it also generates the obligation to show that the `<<` relation is well-founded. This TCC requires the axiom `list_well_founded` to be cited from the `list_adt` theory and is not proved automatically.

```
% Well-founded TCC generated (line 29) for  <<

 run_TCC1: OBLIGATION
  well_founded?[list[action]](LAMBDA (x: action_list, y: action_list): x << y)
```

One way to discover the existence of the `list_well_founded` axiom is to place the cursor at that start of `well_founded?` in the TCC buffer and to type `M-;`. This will bring up a buffer of all declarations that mention this identifier; the `v` key can then be used to view each declaration (use the `q` key to leave this mode). After introducing this axiom with the `lemma` command, we arrive at the following sequent.

```
Rule? (LEMMA "list_well_founded[action]")
Applying list_well_founded[action] where
this simplifies to:
run_TCC1 :

{-1}    well_founded?[list[action]](<<)
  |-------
[1]    well_founded?(LAMBDA (x: action_list, y: action_list): x << y)
```

To complete the proof, we need to establish that `<<` and `LAMBDA (x: action_list, y: action_list): x << y` are equivalent. This is the η-rule of lambda calculus: $f = \lambda x.f(x)$. PVS actually has several eta-rules (for functions, records, tuples, and abstract datatypes); all are invoked by the proof commands `(apply-eta)` and `replace-eta`. In this example, the following command finishes the proof.

```
Rule? (REPLACE-ETA "list_adt[action].<<")
Applying eta axiom scheme to list_adt[action].<< and then replacing
Q.E.D.
```

For each datatype, it is usual to define a recursive function that returns a natural number (or an ordinal) representing its “length” or “size.” The subterm ordering predicate is used to prove termination of this function, but subsequent recursive definitions can then be shown to terminate more simply using the length or size function, as in our original treatment of the termination argument for `run`. See the prelude theory `list_props` for the definition of the `length` function used here.

Although the `list` datatype gives us an adequate semantic treatment for the sequences of actions used in the informal development, it does not reproduce its notation: instead of the $a \circ \alpha$ of the ordinary mathematical presentation, we must write `cons(a, alpha)`. To help reproduce traditional notation, PVS provides a number of prefix, infix, and outfix operators such as `<>` and `[]` (prefix), `|-`, `^`, and `o` (infix), and `[|  |]` and `|[  ]|` (outfix). You can see the whole list with `M-x pvs-help-language`. Infix operators can also be used in a prefix function-application form, and *must* be used in this form when defining new meanings for them. The infix `o` operator is convenient for our purposes, so we overload any existing meaning it may have with the following definition.

```
; o(a, alpha): action_list = cons(a, alpha)
```

(The semicolon at the beginning of this line serves to terminate the previous declaration; because `o` is an infix operator, the one-symbol lookahead of the PVS parser may otherwise confuse the start of this declaration with a continuation of the previous one.) Given this definition, we can write expressions such as `a o alpha`; during proof, we will expand or rewrite with `o` to recover the underlying `cons(a, alpha)` form. Note, however, that we cannot use `a o alpha` as a label in a datatype `CASES` expression, since PVS allows only datatype constructors in this context.

The functions *do* and *test* are specified in the obvious way.

```
do(alpha): state = run(s0, alpha)                                    5

test(alpha, a): output = output(do(alpha), a)
```

□

3.2 Security

Definition 2 In order to discuss security, we require some set of security "domains" and a policy that restricts the allowable flow of information among those domains. Thus, we assume

- a set D of security domains

and use letters $\ldots u, v, w, \ldots$ to denote domains.

A *security policy* is then specified by a reflexive relation $\rightsquigarrow$ on D. We use $\not\rightsquigarrow$ to denote the complement relation, that is

$$\not\rightsquigarrow = (D \times D) \backslash \rightsquigarrow$$

where $\backslash$ denotes set difference. □

PVS Treatment

We introduce *domain* as another uninterpreted, nonempty type, then specify `security_policy` as an uninterpreted relation on *domain*s and constrain it to be reflexive.

```
domain: TYPE+
u, v: VAR domain

security_policy(u, v): bool

policy_refl: AXIOM reflexive?(security_policy)
```

□

The higher-order predicate `reflexive?` comes from the prelude theory `relations` (it is higher-order because it is a predicate that takes another predicate—actually a relation, which in PVS is just a predicate on a two-tuple—as its argument). Notice that by asserting, in the axiom `policy_refl`, that the uninterpreted relation `security_policy` is reflexive, we are implicitly assuming that the predicate `reflexive?` is satisfiable on this class of relations. This is obviously so, but what if we had required that the `security_policy` relation be both reflexive and asymmetric? No relation (on a nonempty type) can satisfy both these properties, so by asserting them of the `security_policy` relation, we would have created an inconsistent specification.

It is always possible to make mistakes when writing specifications, but some mistakes are worse than others. Mistakes that create inconsistent specifications are particularly egregious because we can prove absolutely anything from such specifications—they are, essentially, meaningless. PVS is carefully designed so that the only way to introduce an inconsistency into a PVS specification (provided all its TCCs have been proved) is with an `AXIOM`. Hence, we should always be very careful when using `AXIOM`s, and should generally avoid introducing them gratuitously.

In this specification, the `AXIOM` can be avoided as follows. Instead of first introducing `security_policy` as a relation on `domain` and then asserting, via an `AXIOM`, that it is reflexive, we can introduce `security_policy` as a constant of the reflexive subtype of the type of relations on `domain`. In PVS, if `p` is a predicate on a type `T`, then `(p)` denotes the predicate subtype of `T` satisfying `p`. Thus, `(reflexive?[domain])` denotes the type of reflexive relations on `domain`, and we can then introduce `security_policy` as follows.

```
security_policy: (reflexive?[domain])
```

We saw in Chapter 2 that PVS allows constants to be declared only for those types that are known to be nonempty. Here, declaration of the constant `security_policy` requires that its type `(reflexive?[domain])` is nonempty, and PVS generates the following TCC to ensure this fact.

```
% Existence TCC generated (line 30) for security_policy: (reflexive?[domain])

 security_policy_TCC1: OBLIGATION (EXISTS (x: (reflexive?[domain])): TRUE);
```

This can be proved by exhibiting the equality relation on `domain` as such a predicate using the proof command `(inst 1 "eq[domain]")` and then finishing off the proof with `(grind)`.

Alternatively, we could separate declaration of the constant `security_policy` from that of its type and provide a `CONTAINING` clause in the type declaration as a hint to the prover.

```
 refl_rel: TYPE = (reflexive?[domain]) CONTAINING eq[domain]
 security_policy: refl_rel
```

When the declarations are given in this form, PVS is able to discharge the TCC automatically.

It turns out that the specification we will construct for security only makes sense if the security policy $\leadsto$ is transitive, in addition to reflexive. (This point is discussed at length in [Rus92].) Thus, we really need to amend the declaration of `security_policy` to read something like the following.[2]

```
 refl_trans_rel: TYPE = (reflexive?[domain] AND transitive?[domain])          6
                                                     CONTAINING eq[domain]
 security_policy: refl_trans_rel
```

Unfortunately, the specification of `refl_trans_rel` is not type-correct: the `AND` connective properly applies to booleans whereas here we have applied it to predicates of type `pred[pred[[domain,domain]]]` (i.e., to predicates on relations on `domain`). We can correct this using the following `LAMBDA` form

```
 refl_trans_rel: TYPE = (LAMBDA (r: pred[[domain,domain]]):                   7
        reflexive?(r) AND transitive?(r)) CONTAINING eq[domain]
```

but the result is ugly and not easy to read.

In fact, our original intuition was reasonable—we simply need to *overload* `AND` so that it applies to predicates as well as to simple booleans. All infix operators like `AND` are really functions and also have a prefix form (e.g., `a AND b` can also be written as `AND(a, b)`) and can be overloaded by defining additional types and interpretations for the prefix form. Thus, we could define

```
; AND(x,y: pred[pred[[domain,domain]]])(r: pred[[domain, domain]]): bool =
       x(r) AND y(r)
```

[2] A relation that is both reflexive and transitive is called a *preorder*. This is defined in the PVS prelude, so that we could simply say `security_policy: (preorder?[domain])`. However, the construction used here allows us to demonstrate some additional aspects of PVS

and then the treatment in box $\boxed{6}$ becomes type-correct. However, this redefinition of `AND` is also rather ugly, and very specific to this single application. A better solution is to define a generic theory that overloads several of the propositional connectives by "lifting" them to apply to predicates as follows.

```
lifted_predicates[T:type]: THEORY
BEGIN
  t: VAR T
  p,q: VAR pred[T]

; AND(p,q)(t): bool = p(t) AND q(t);
; OR(p,q)(t): bool = p(t) OR q(t);
; IMPLIES(p,q)(t): bool = p(t) IMPLIES q(t);
END lifted_predicates
```

Then our specification can take the following form.

```
IMPORTING lifted_predicates

refl_trans_rel: TYPE = (reflexive?[domain] AND transitive?[domain])
                                                   CONTAINING eq[domain]
security_policy: refl_trans_rel
```

PVS's type-inference is capable of determining the correct instance of `lifted_predicates` (it is `lifted_predicates[pred[[domain,domain]]]`) needed to provide a correct type for the `AND` in `refl_trans_rel`.

Even this treatment is a little crude, because it only applies to the propositional connectives explicitly mentioned in the `lifted_predicates` theory. It is possible to give a totally general treatment using PVS *conversions*. The simple kind of conversion is a function that is applied automatically to convert a type-incorrect expression to a type correct one. If, for example, we wished to omit explicit mention of the function *do*, so that an action sequence and the state that it leads to are punned together, we could change the declarations of $\boxed{5}$ to the following form.

```
do(alpha): state = run(s0, alpha)

CONVERSION do

test(alpha, a): output = output(alpha, a)
```

The function `output` requires a state as its first argument, not an action sequence; because `do` is declared as a `CONVERSION`, the PVS typechecker is able to insert an application of `do`, so that the action sequence `alpha` is replaced by the state `do(alpha)`, thereby replacing the expression `output(alpha, a)` by the type-correct form `output(do(alpha), a)`. This transformation is revealed by the Emacs command `M-x ppe` (prettyprint-expanded).

Whereas this simple kind of conversion turns a type-incorrect term `a` into a type-correct one by inserting a function `f` to produce `f(a)`, a dual kind of conversion turns a type-incorrect function `f` into a type-correct application by supplying an *argument* `a` to produce `f(a)`. This conversion is more complex because the `a` will be a variable and therefore needs to occur in some binding construct (e.g., `FORALL` or `LAMBDA`). The circumstances in which PVS will perform such a conversion are restricted to the case where a type incorrect application `f(x, y)` can be made type-correct by transforming it to `(LAMBDA (a): f(x(a), y(a)))`. Now it may be that not all the parameters to such a function `f` need to be applied to an argument `a` (e.g., `(LAMBDA (a): f(x(a),y))` may be type-correct), and we can provide for this by causing the *K combinator*,[3] which has the definition `K(p)(q) = p`, to be available as a conversion. The original `f(x, y)` can then be converted first to `(LAMBDA (a): f(x(a), y(a)))`, then to `(LAMBDA (a): f(x(a), K(y)(a)))`, and finally reduced (by application of the definition of `K`) to the type-correct `(LAMBDA (a): f(x(a), y))`. To avoid invoking it accidentally, the step that starts this sequence of conversions is performed only in contexts where a function having the form of the K combinator has been declared as a conversion (even if K is not needed in the sequence concerned).

In the present case, the desired (and type-correct) interpretation of `refl_trans_rel` in $\boxed{6}$ is that shown in $\boxed{7}$, which has *exactly* the form produced by the second kind of conversion. We can cause this to be applied by importing the following theory.

```
K_conversion[T1, T2: TYPE]: THEORY
BEGIN
  t1: VAR T1
  t2: VAR T2

  K(t1)(t2): T1 = t1

  CONVERSION K

END K_conversion
```

The following text is then acceptable to PVS.

```
 IMPORTING K_conversion

 refl_trans_rel: TYPE = (reflexive?[domain] AND transitive?[domain])
                                                   CONTAINING eq[domain]
 security_policy: refl_rel
```

`M-x ppe` reveals that the `refl_trans_type` declaration is converted to the following form (which is identical to $\boxed{7}$).

```
 refl_rel: TYPE =
      (LAMBDA (x: PRED[[domain, domain]]):
              reflexive?[domain](x) AND transitive?[domain](x))
        CONTAINING eq[domain]
```

[3]This terminology comes from Combinatory Logic.

The following TCC (which is proved automatically) is generated to ensure that the type is inhabited. (The same TCC is generated for both the lifted_predicates and K_conversion treatments.)

```
% Subtype TCC generated (line 57) for  eq[domain]

 refl_rel_TCC1: OBLIGATION
        reflexive?[domain](eq[domain]) AND transitive?[domain](eq[domain]);
```

Given the definition of $\rightsquigarrow$ as security_policy, we can define $\not\rightsquigarrow$ as its negation. In order to approximate the notation of the traditional mathematical development, we will use an infix operator for $\not\rightsquigarrow$; |> seems to be the nearest approximation available in PVS.

```
 |>(u, v): bool = NOT security_policy(u, v)
```

□

3.3 Information Flow

We wish to define security in terms of "interference" or information flow, so the next step is to capture these ideas formally. The key observation is that information can be said to flow from a domain u to a domain v exactly when actions submitted by domain u cause the behavior of the system perceived by domain v to be different from that perceived when those actions are not present. We therefore define a function that removes, or "purges," from an action sequence all those actions submitted by a given domain. We can then say that information flows from domain u to domain v, or that u *interferes* with domain v if the latter can distinguish between the state of the machine after it has processed a given action sequence, and the state after processing the same action sequence purged of actions from domain u. We formalize this idea as follows.

Definition 3 We first assume that each action is associated with an agent (or "subject") of a specific security domain and introduce the function $dom(a)$ to identify the security domain of action a.

Then, for $u \in D$ and α an action sequence in A^*, we define α/u (α "purged" by u) to be the subsequence of α formed by deleting all actions associated with domain u. That is:

$$\begin{aligned} \Lambda/u &= \Lambda \\ (a \circ \alpha)/u &= \begin{cases} \alpha/u & \text{if } dom(a) = u \\ a \circ (\alpha/u) & \text{otherwise.} \end{cases} \end{aligned}$$

Domain u is said to *interfere* with domain v if there exists an action sequence α and an action a with $dom(a) = v$ such that

$$output(run(s_0, \alpha), a) \neq output(run(s_0, \alpha/u), a);$$

that is,

$$test(\alpha, a) \neq test(\alpha/u, a).$$

We identify security with the requirement that u should be noninterfering with v whenever the policy specifies $u \not\rightsquigarrow v$. That is, we say that a system is *secure* for the policy $\rightsquigarrow$ if

$$u \not\rightsquigarrow dom(a) \supset test(\alpha, a) = test(\alpha/u, a).^4$$

The intuition here is that the machine starts off in the initial state s_0 and is presented with a sequence $\alpha \in A^*$ of actions. This causes the machine to produce a series of outputs and to progress through a series of states, eventually reaching the state $do(\alpha)$. At that point the action a is performed, and the corresponding output $test(\alpha, a)$ is observed. We can think of presentation of the action a and observation of its output as an experiment performed by $dom(a)$ in order to learn something about the action sequence α. If $dom(a)$ can distinguish between the action sequences α and α/u by such experiments, then u has "interfered" with $dom(a)$ and the system is not secure with respect to policies that specify $u \not\rightsquigarrow dom(a)$.

As mentioned earlier, this definition of security (and in particular, the definition of the *purge* function) only makes sense if the relation $\rightsquigarrow$ is transitive. See [Rus92] for an extended discussion of this topic and a formally verified treatment of intransitive interference policies. □

PVS Treatment

We introduce *dom* as an uninterpreted function, then define a recursive function that performs the "purge" operation; in order to reproduce the traditional notation, we overload the infix operator / to be the name of this function.

```
dom(a): domain             % equivalent to dom: [action -> domain]

/(beta, u): RECURSIVE action_list =
  CASES beta OF
    null: null,
    cons(a, alpha): IF dom(a) = u THEN alpha / u
                    ELSE a o (alpha / u) ENDIF
  ENDCASES
 MEASURE length(beta)
```

Security is specified by the Boolean constant secure.

```
security: bool = FORALL a, u, alpha:
 u |> dom(a) IMPLIES test(alpha, a) = test(alpha / u, a)
```

[4]Formulas such as these are to be read as universally quantified over their free variables (here u, a, and α); we use $\supset$ to denote implication.

There are several alternative ways to state the security requirement. Instead of a Boolean constant, we could have used a formula, as follows.

```
security: FORMULA FORALL a, u, alpha:
 u |> dom(a) IMPLIES test(alpha, a) = test(alpha / u, a)
```

The main difference between these two formulations is that a formula cannot be used as a component in any other linguistic construction: it can only be proved, or cited in proofs. We will eventually wish to state and prove an "unwinding" theorem that will have the form

$$\textit{some simpler conditions} \supset \textit{security} \tag{3.1}$$

and this cannot be stated directly if `security` is given as a `FORMULA`.[5] The Boolean constant seems the better approach here, but we should examine this choice in a little more detail.

It is likely that proving a theorem of the form (3.1) will require induction on the length of the action sequence appearing in the definition of `security`. As currently specified, `security` is a defined Boolean constant whose body is a closed formula. We will need to expand the definition, therefore, to expose the induction variable `alpha`. This need for fine-grained manipulation will limit the likely effectiveness of automated proof strategies. A possibly better treatment, therefore, is one that leaves the variables to which we are likely to need access exposed as arguments. This can be done by changing from a Boolean constant, `security`, to a Boolean function, `secure`, that takes the action sequence, α, as an argument.

```
secure(alpha): bool = FORALL a, u:
 u |> dom(a) IMPLIES test(alpha, a) = test(alpha / u, a)
```

We will use this form for the time being. □

3.4 Unwinding

The noninterference definition of security is expressed in terms of sequences of actions and state transitions; in order to obtain straightforward techniques for verifying the security of systems, we would like to derive conditions on individual state transitions. The first step in this development is to partition the states of the system into equivalence classes which all "appear identical" to a given domain. The verification technique will then be to prove that each domain's view of the system is unaffected by the actions of domains that are required to be noninterfering with it.

[5] We could prove the `FORMULA` *security* using "*some simpler conditions*" as lemmas; by the deduction theorem, this is logically equivalent to the other form, but it is linguistically less direct and would not be explicit in the specification.

Definition 4 A system M is *view-partitioned* if, for each domain $u \in D$, there is an equivalence relation $\stackrel{u}{\sim}$ on S. These equivalence relations are said to be *output-consistent* if

$$s \stackrel{dom(a)}{\sim} t \supset output(s, a) = output(t, a).$$

□

Output consistency is required in order to ensure that two states s and t that appear identical to a given domain really are indistinguishable in terms of the outputs they produce in response to actions from that domain.

The definition of security requires that the outputs seen by one domain are unaffected by the actions of other domains that are specified to be noninterfering with the first. The next result shows that, for an output consistent system, security is achieved if "views" are similarly unaffected.

Lemma 1 *Let $\rightsquigarrow$ be a policy and M a view-partitioned, output-consistent system such that*

$$u \not\rightsquigarrow v \supset do(\alpha) \stackrel{v}{\sim} do(\alpha/u).$$

Then M is secure for $\rightsquigarrow$.

Proof: Let $u \not\rightsquigarrow dom(a)$. The hypothesis to the lemma then provides

$$do(\alpha) \stackrel{dom(a)}{\sim} do(\alpha/u).$$

Output consistency then ensures

$$output(do(\alpha), a) = output(do(\alpha/u), a).$$

But this is simply

$$test(\alpha, a) = test(\alpha/u, a),$$

which is the definition of security for $\rightsquigarrow$ given by Definition 3. □

PVS Treatment

One way to specify in PVS that each domain induces an equivalence relation on states is as follows.

```
view_equiv(u): (equivalence?[state])
```

The trouble with this approach is that PVS does not provide highly automated reasoning support for equivalence relations.[6] Another way to specify this property is to hypothesize a function $view(s, u)$ that gives the abstract "view" of state s, as seen by domain u. Then we can state that two states s and t are *view_equiv* as far as u is concerned if $view(s, u) = view(t, u)$. Using this approach, the properties of *view_equiv* will follow by equality reasoning (which PVS automates very effectively). We specify this treatment in PVS as follows.

```
V : TYPE+

view(s, u): V

view_equiv(u)(s, t): bool = view(s, u) = view(t, u)
```

Next, we can specify *output_consistent* and give a PVS rendition of Lemma 1. The main condition of Lemma 1, namely

$$u \not\rightsquigarrow v \supset do(\alpha) \overset{v}{\sim} do(\alpha/u),$$

is likely to be needed again, so we name it *view_consistent* and refer to it in the specification of Lemma 1. *Output_consistent* is specified as a Boolean constant, and *view_consistent* as a Boolean function. Recall the specification of *secure* for discussion of these choices.

```
                                                                    8
output_consistent: bool = FORALL a, s, t:
  view_equiv(dom(a))(s, t) IMPLIES output(s, a) = output(t, a)

view_consistent(alpha) : bool = FORALL u, v:
  u |> v IMPLIES view_equiv(v)(do(alpha), do(alpha / u))

lemma1: LEMMA
  output_consistent AND view_consistent(alpha) IMPLIES secure(alpha)
```

The lemma is proved by `(grind)`.[7] □

Continuing the ordinary mathematical development, we next define constraints on individual state transitions.

Definition 5 Let M be a view-partitioned system and $\rightsquigarrow$ a policy. We say that M *locally respects* $\rightsquigarrow$ if

$$dom(a) \not\rightsquigarrow v \supset s \overset{v}{\sim} step(s, a)$$

and that M is *step-consistent* if

$$s \overset{v}{\sim} t \supset step(s, a) \overset{v}{\sim} step(t, a).$$

□

[6] It should, and a future version of PVS will do so.

[7] Experts may notice a potential problem in the way `alpha` is quantified in this form of `lemma1`; we discuss this point later.

We now have the local conditions on individual state transitions that are sufficient to guarantee security. This result is a version of the unwinding theorem of Goguen and Meseguer[GM84].

Theorem 1 *(Unwinding Theorem)* Let $\rightsquigarrow$ be a policy and M a view-partitioned system that is

1. output-consistent,
2. step-consistent, and
3. locally respects $\rightsquigarrow$.

Then M is secure for $\rightsquigarrow$.

Proof: We use proof by induction on the length of α to establish

$$u \not\rightsquigarrow v \supset do(\alpha) \stackrel{v}{\sim} do(\alpha/u). \tag{3.2}$$

The result then follows by the previous lemma. The basis is the case $\alpha = \Lambda$ and is elementary. For the inductive step, we assume the inductive hypothesis for α of length n and consider $a \circ \alpha$. We now need to prove

$$u \not\rightsquigarrow v \supset do(a \circ \alpha) \stackrel{v}{\sim} do((a \circ \alpha)/u). \tag{3.3}$$

We assume $u \not\rightsquigarrow v$ and consider two cases.

Case 1: $dom(a) = u$. In this case, the definition of / provides

$$do((a \circ \alpha)/u) = do(\alpha/u)$$

and the inductive hypothesis gives

$$do(\alpha/u) \stackrel{v}{\sim} do(\alpha).$$

The facts that $dom(a) \not\rightsquigarrow v$ and that M locally respects $\rightsquigarrow$ ensure

$$do(\alpha) \stackrel{v}{\sim} step(do(\alpha), a)$$

and we also have, by definition,

$$step(do(\alpha), a) = do(a \circ \alpha). \tag{3.4}$$

Since $\stackrel{v}{\sim}$ is an equivalence relation, (3.3) follows and we conclude the inductive step in this case.

Case 2: $dom(a) \neq u$**.** In this case, the definition of / provides

$$do((a \circ \alpha)/u) = do(a \circ (\alpha/u)).$$

The inductive hypothesis gives

$$do(\alpha/u) \overset{v}{\sim} do(\alpha),$$

from which step consistency allows us to deduce

$$do(a \circ (\alpha/u)) \overset{v}{\sim} do(a \circ \alpha)$$

and thereby (3.3), to conclude the inductive step in this case.

□

PVS Treatment

We define `local_respect` and `step_consistent` as Boolean constants similar to the way `output_consistent` was defined.

```
local_respect: bool = FORALL v, s, a:
  dom(a) |> v IMPLIES view_equiv(v)(s, step(s,a))

step_consistent : bool = FORALL u, s, t, a:
  view_equiv(u)(s,t) IMPLIES view_equiv(u)(step(s,a), step(t,a))
```

The informal proof of the unwinding theorem used an implicit lemma to establish its equation 3.4, and the heart of the proof was an inductive argument that was used to establish formula 3.2. For the PVS treatment, it is convenient to break these out as explicit `LEMMA`s which we call `lemma2` and `lemma3`, respectively. The Unwinding Theorem is then stated as the `THEOREM` `unwinding`.

```
lemma2: LEMMA step(do(alpha),a) = do(a o alpha)

lemma3: LEMMA
  local_respect AND step_consistent IMPLIES view_consistent(alpha)

unwinding: THEOREM
  local_respect AND step_consistent AND output_consistent
    IMPLIES secure(alpha)
```

`Lemma2` is proved by `(grind)`, but the proof of `lemma3` is more involved.

As with the informal proof, we begin the proof of `lemma3` by inducting on `alpha`.

```
lemma3 :

  |-------
{1}    (FORALL (alpha: action_list):
         local_respect AND step_consistent => view_consistent(alpha))

Rule? (INDUCT "alpha")
Inducting on alpha,
this yields  2 subgoals:
lemma3.1 :

  |-------
{1}    local_respect AND step_consistent => view_consistent(null)
```

The PVS `induct` strategy tries to infer the correct induction scheme to use from the type of the given induction variable. Here the variable is `alpha`, whose type (`action_list`) is an instance of `list`; this causes PVS to invoke the structural induction scheme for lists. Such structural induction schemes are generated automatically from the specifications for abstract data types. The structural induction scheme for the `list` abstract data type is the following.

```
 list_induction: AXIOM
       (FORALL (p: [list -> boolean]):
          p(null)
              AND
            (FORALL (cons1_var: T, cons2_var: list):
               p(cons2_var) IMPLIES p(cons(cons1_var, cons2_var)))
              IMPLIES (FORALL (list_var: list): p(list_var)));
```

This says that to prove a property `p` true for all lists, it is sufficient to show that it is true of the empty list `null` (the base case) and, assuming it is true of an arbitrary list `cons2_var`, that it will also be true of this formed by `cons`ing an arbitrary element `cons1_var` on the front (the inductive step). PVS has automatically instantiated this general scheme for the predicate of `lemma3` (using higher-order matching) and is inviting us to consider the base case.

```
lemma3.1 :

  |-------
{1}    local_respect AND step_consistent => view_consistent(null)
```

This is easily discharged with `(grind)`, and PVS then presents us with the inductive step.

```
Rule? (GRIND)
··· many rewrites omitted
This completes the proof of lemma3.1.

lemma3.2 :

  |-------
{1}    (FORALL (cons1_var: action, cons2_var: list[action]):
         (local_respect AND step_consistent => view_consistent(cons2_var))
             IMPLIES local_respect AND step_consistent
               => view_consistent(cons(cons1_var, cons2_var)))
```

The definitions `view_consistent` and `view_equiv` are artefacts of our specification and should always be expanded. We instruct the prover to do this by the command `(AUTO-REWRITE "view_consistent" "view_equiv")` and then give the command `(REDUCE)`. The latter invokes the core of the `grind` strategy, but without establishing additional rewrites.[8]

```
Rule? (AUTO-REWRITE "view_consistent" "view_equiv")
···
Rule? (REDUCE)
··· many rewrites omitted
Repeatedly simplifying with decision procedures, rewriting,
  propositional reasoning, quantifier instantiation, skolemization,
 if-lifting and equality replacement,
this simplifies to:
lemma3.2 :

{-1}    view(v!1, do(cons2_var!1)) = view(v!1, do(cons2_var!1 / u!1))
{-2}    local_respect
{-3}    step_consistent
{-4}    u!1 |> v!1
  |-------
{1}    view(v!1, do(cons(cons1_var!1, cons2_var!1)))
          = view(v!1, do(cons(cons1_var!1, cons2_var!1) / u!1))
```

In the proof of an inductive step, the general approach is to expand some of the recursively defined functions appearing below the line so that their components will match some of those appearing above the line. Here the appropriate function to expand is the purge function `/`. We specify that only the instance below the line should be expanded using the `+` qualifier on the `expand` command.

[8]The single command `(grind :defs nil :rewrites ("view_consistent" "view_equiv"))` could replace both these commands.

```
Rule? (EXPAND "/" +)
Expanding the definition of /,
this simplifies to:
lemma3.2 :

[-1]    view(v!1, do(cons2_var!1)) = view(v!1, do(cons2_var!1 / u!1))
[-2]    local_respect
[-3]    step_consistent
[-4]    u!1 |> v!1
  |-------
{1}    (view(v!1, do(cons(cons1_var!1, cons2_var!1)))
           =
         view(v!1,
              IF dom(cons1_var!1) = u!1 THEN do(cons2_var!1 / u!1)
              ELSE do(cons1_var!1 o (cons2_var!1 / u!1))
              ENDIF))
```

This sequent contains both the functions `cons` and `o`, the former from the induction scheme, and the latter from our own specification. For things to match up, we need to ensure that only one for is used. We could expand the infix `o` to a `cons` with either the `expand` or `rewrite` commands, but prefer to "contract" the `cons` to `o` by rewriting the definition of `o` in the reverse (right to left) direction. We specify this with the command `(REWRITE "o" :DIR RL)`.

```
Rule? (REWRITE "o" :DIR RL)
Found matching substitution:
alpha gets cons2_var!1,
a gets cons1_var!1,
Rewriting using o,
this simplifies to:
lemma3.2 :

[-1]    view(v!1, do(cons2_var!1)) = view(v!1, do(cons2_var!1 / u!1))
[-2]    local_respect
[-3]    step_consistent
[-4]    u!1 |> v!1
  |-------
{1}    (view(v!1, do(cons1_var!1 o cons2_var!1))
           =
         view(v!1,
              IF dom(cons1_var!1) = u!1 THEN do(cons2_var!1 / u!1)
              ELSE do(cons1_var!1 o (cons2_var!1 / u!1))
              ENDIF))
```

The condition to the `IF` expression below the line suggests the case split that needs to be performed next. We could cause this to be done explicitly using `lift-if` and `split`, but `smash` will do it automatically, and also apply the decision procedures to simplify the resulting formulas.

```
Rule? (SMASH)
Repeatedly simplifying with BDDs, decision procedures, rewriting,
and if-lifting,
this yields  2 subgoals:
lemma3.2.1 :

[-1]    view(v!1, do(cons2_var!1)) = view(v!1, do(cons2_var!1 / u!1))
[-2]    local_respect
[-3]    step_consistent
[-4]    u!1 |> v!1
{-5}    dom(cons1_var!1) = u!1
  |-------
{1}    (view(v!1, do(cons1_var!1 o cons2_var!1))
          = view(v!1, do(cons2_var!1 / u!1)))
```

The PVS proof is now at a point corresponding to the start of "Case 1" in the informal proof, except that we have already performed the expansion of the purge function `/`. The PVS proof proceeds in the same way as the informal proof by invoking the definition of `local_respect`. Since this definition expands to a quantified formula, we immediately use `reduce` to instantiate its variables and to simplify the result.

```
Rule? (EXPAND "local_respect")
Expanding the definition of local_respect,
...
Rule? (REDUCE)
... rewrites omitted
Repeatedly simplifying with decision procedures, rewriting,
  propositional reasoning, quantifier instantiation, skolemization,
 if-lifting and equality replacement,
this simplifies to:
lemma3.2.1 :

{-1}    view(v!1, do(cons2_var!1))
          = view(v!1, step(do(cons2_var!1), cons1_var!1))
{-2}    view(v!1, do(cons2_var!1 / u!1))
          = view(v!1, step(do(cons2_var!1), cons1_var!1))
[-3]    step_consistent
[-4]    u!1 |> v!1
[-5]    dom(cons1_var!1) = u!1
  |-------
{1}    (view(v!1, do(cons1_var!1 o cons2_var!1))
          = view(v!1, step(do(cons2_var!1), cons1_var!1)))
```

The instances of `step(do(cons2_var!1), cons1_var!1)` appearing in this sequent are equal, by lemma2, to `do(cons1_var!1 o cons2_var!1)`. Performing this rewrite causes the two sides of the formula below the line to become identical, and finishes this branch of the proof.

```
Rule? (REWRITE "lemma2")
Found matching substitution:
a gets cons1_var!1,
alpha gets cons2_var!1,
Rewriting using lemma2,

This completes the proof of lemma3.2.1.
```

We are now at a point corresponding to "Case 2" in the informal proof, following the expansion of the purge function `/`.

```
lemma3.2.2 :

[-1]    view(v!1, do(cons2_var!1)) = view(v!1, do(cons2_var!1 / u!1))
[-2]    local_respect
[-3]    step_consistent
[-4]    u!1 |> v!1
  |-------
{1}    dom(cons1_var!1) = u!1
{2}    (view(v!1, do(cons1_var!1 o cons2_var!1))
           = view(v!1, do(cons1_var!1 o (cons2_var!1 / u!1))))
```

As in the informal proof, we apply the definition of `step_consistent` and reduce the result.

```
Rule? (EXPAND "step_consistent")
Expanding the definition of step_consistent,
...
Rule? (REDUCE)
... rewrites omitted
Repeatedly simplifying with decision procedures, rewriting,
  propositional reasoning, quantifier instantiation, skolemization,
 if-lifting and equality replacement,
this simplifies to:
lemma3.2.2 :

[-1]    view(v!1, do(cons2_var!1)) = view(v!1, do(cons2_var!1 / u!1))
[-2]    local_respect
{-3}    view(v!1, step(do(cons2_var!1), cons1_var!1))
          = view(v!1, step(do(cons2_var!1 / u!1), cons1_var!1))
[-4]    u!1 |> v!1
  |-------
[1]    dom(cons1_var!1) = u!1
[2]    (view(v!1, do(cons1_var!1 o cons2_var!1))
           = view(v!1, do(cons1_var!1 o (cons2_var!1 / u!1))))
```

Here, two applications of `lemma2` will cause formula `-3` to become identical to formula `2` and thereby finish the proof.

```
Rule? (REWRITE "lemma2")
Found matching substitution:
a gets cons1_var!1,
alpha gets cons2_var!1,
Rewriting using lemma2,
...
Rule? (REWRITE "lemma2")
Found matching substitution:
a gets cons1_var!1,
alpha gets cons2_var!1 / u!1,
Rewriting using lemma2,

This completes the proof of lemma3.2.2.

This completes the proof of lemma3.2.
Q.E.D.
```

We performed this proof in PVS in a manner that followed the informal proof quite closely. Notice, however, that apart from the identification of lemmas there was little "intelligence" required—the case split, in particular, arose naturally from the two cases in the definition of the purge function `/`. Inductive proofs of this routine kind can be performed automatically by the `induct-and-simplify` strategy. This takes arguments, similar to those of `grind`, to control the formulas available for rewriting and the way instantiation is performed. The control tactic that PVS uses for automated rewriting in this strategy is usually able to expand the correct definitions in the inductive conclusion. The default selections are adequate for `lemma3`, and the simple command `(induct-and-simplify "alpha")` proves the result. Notice that this does not refer to `lemma2` (it is essentially proved in-line), so that lemma can be deleted from the specification.

The theorem `unwinding` is proved simply by using `lemma1` and `lemma3` as rewrites, with the following command.

```
  (grind :defs nil :rewrites ("lemma1" "lemma3"))
```

3.4.1 Implicit Quantification

It might seem that we are done at this point, but we should revisit the choice that `secure` and `view_consistent` are specified as functions on `alpha`, rather than as closed boolean constants. Our motivation for specifying `secure` in this way was that it would leave the likely induction variable exposed. We have now seen that induction is performed in the proof of `lemma3`, whose conclusion is `view_consistent(alpha)`, rather than in the proof of `unwinding` (where `secure(alpha)` is the conclusion), so our motivation was mistaken. Accordingly, we change the specification of `secure` to the following form

```
 secure: bool = FORALL a, u, alpha:
    u |> dom(a) => test(alpha, a) = test(alpha / u, a)
```

and modify `lemma1` and `unwinding` to correspond.

```
lemma1: LEMMA                                                          9
  output_consistent AND view_consistent(alpha) => secure

unwinding: THEOREM
  local_respect AND step_consistent AND output_consistent => secure
```

When we rerun the proofs of these formulas, we find that `unwinding` succeeds, but the proof for `lemma1` fails, leaving us to contemplate the following sequent.

```
lemma1 :

{-1}    view(dom(a!1), run(s0, alpha!1)) = view(dom(a!1), run(s0, alpha!1 / u!1))
  |-------
{1}    view(dom(a!1), run(s0, alpha!2)) = view(dom(a!1), run(s0, alpha!2 / u!1))
{2}    security_policy(u!1, dom(a!1))
{3}    output(run(s0, alpha!2), a!1) = output(run(s0, alpha!2 / u!1), a!1)

Rule?
```

We notice there are two different Skolem constants present for `alpha`: `alpha!1` and `alpha!2`. This usually indicates an incorrect instantiation, but here it is directing our attention to a larger problem: the quantification is incorrect in `lemma1`.

Free variables in PVS formulas—such as the `alpha` of `lemma1` in 9—are interpreted as universally quantified at the outermost level. When such a variable appears—as here—only in the antecedent to an implication, then it is within the scope of an implicit negation and so an outermost universal quantifier is equivalent to a local existential. This is seldom what is intended. Thus, for example, the `lemma1` of 9 is equivalent to the following, plainly erroneous, specification.

```
lemma1: LEMMA
  output_consistent AND (EXISTS alpha: view_consistent(alpha)) => secure
```

In the original specification of `lemma1` in 8 (page 67), `alpha` appeared in the consequent to the implication as well as the antecedent, so this problem did not arise. However, that specification binds the *same* `alpha` in both `view_consistent` and `secure` and, on reflection, we recognize that this is probably not what we intended either; the following (stronger) result more accurately reflects the informal treatment.

```
lemma1: LEMMA                                                          10
  output_consistent AND (FORALL alpha: view_consistent(alpha))
          => (FORALL alpha: secure(alpha))
```

In fact, this treatment is the same as would be obtained by quantifying `alpha` locally in the definitions of both view_consistent and `secure`.

```
view_consistent : bool = FORALL u, v, alpha:                          11
  u |> v IMPLIES view_equiv(v)(do(alpha), do(alpha / u))

lemma1: LEMMA
  output_consistent AND view_consistent => secure

lemma3: LEMMA
  local_respect AND step_consistent IMPLIES view_consistent
```

These corrected forms of `lemma1` are proved by `(grind)`, and do not affect the proof of `unwinding`. The version of `lemma3` in [11] requires `(expand "view_consistent")` and `(flatten)` to expose `alpha` before induction can be applied. The specifications and proofs presented below and in the appendix use the treatment of [11].

Since `lemma1` has no external significance and the weak form of [8] is adequate to prove the theorem `unwinding`, the preference for [10] and [11] is simply that they better reflect our intent. In cases where the formula concerned forms part of the external specification, however, these issues concerning free variables and the scope and parity of quantification will vitally affect the significance and utility of any results established. There is no foolproof way to be sure these issues are handled correctly: careful introspection and the scrutiny of knowledgeable reviewers are the best safeguards—as they are for other aspects of formal specification. As a general rule, the appearance of a free variable only in the antecedent to an implication is almost certainly incorrect; and the appearance of the same free variable on both sides of an implication should be viewed with suspicion. When in doubt, make the quantification explicit.

3.4.2 LaTeX-printed Specification

The appearance of LaTeX-printed PVS specifications can be adjusted by user-supplied tables that describe how various identifiers should be rendered in LaTeX. The following `noninterference.sub` file reproduces the notation used in the traditional mathematical development.

```
|>                2        1       {#1 \not\leadsto #2}
security_policy   2        1       {#1 \leadsto #2}
view_equiv        (1 2)    1       {#2 \stackrel{#1}{\sim} #3}
o                 2        1       {#1 \circ #2}
null              id       1       {\Lambda}
=>                2        1       {#1 \supset #2}
state             id       1       {\cal S}
action            id       1       {\cal A}
output            id       1       {\cal O}
output            2        1       {\pvsid{output}(#1,#2)}
domain            id       1       {\cal D}
V                 id       1       {\cal V}
action_list       id       1       {{\cal A}^{*}}
```

In this file, the first column gives the PVS identifier concerned, and the fourth its LaTeX translation; the third column specifies the number of character positions the translation will occupy (the LaTeX-printer uses this information to calculate where to make line breaks). Since PVS identifiers may be overloaded, the second column helps identify the instances concerned: `id` refers to a simple identifier, a number to a function application with that many arguments, and a list to a curried application (so `view_equiv` is a function that takes a single argument to yield a function that takes a further two arguments).

The results are shown below. Note that, owing to a bug in the current version of the PVS LaTeX-printer, it is the post-conversion form of `refl_trans_rel` that is printed; owing to another bug, constructors with arguments are not translated in `CASES` expressions (e.g., `cons(a, alpha)` does not become $a \circ \alpha$).

noninterference : THEORY
BEGIN

$\mathcal{S}, \mathcal{A}, \mathcal{O}$: TYPE+

s_0 : $\mathcal{S}$

s, t : VAR $\mathcal{S}$

a, b : VAR $\mathcal{A}$

$\text{step}(s, a)$: $\mathcal{S}$

$\text{output}(s, a)$: $\mathcal{O}$

$\mathcal{A}^*$: TYPE $= \text{list}[\mathcal{A}]$

α, β : VAR $\mathcal{A}^*$

$a \circ \alpha$: $\mathcal{A}^* = \text{cons}(a, \alpha)$

$\text{run}(s, \beta)$:
RECURSIVE $\mathcal{S}$ = CASES β OF Λ : $s, \text{cons}(a, \alpha)$: $\text{step}(\text{run}(s, \alpha), a)$ ENDCASES
MEASURE β BY $\ll$

$\mathcal{D}$: TYPE+

u, v : VAR $\mathcal{D}$

IMPORTING K_conversion

refl_trans_rel :
TYPE =
$(\lambda\ (x : \text{PRED}[[\mathcal{D}, \mathcal{D}]]) : \text{reflexive?}[\mathcal{D}](x) \wedge \text{transitive?}[\mathcal{D}](x))$
CONTAINING $\text{eq}[\mathcal{D}]$

$\rightsquigarrow$: refl_trans_rel

$u \not\rightsquigarrow v$: bool $= \neg u \rightsquigarrow v$

$\mathrm{dom}(a): \mathcal{D}$

$$
\begin{array}{l}
/(\beta, u): \\
\quad \text{RECURSIVE } \mathcal{A}^* \;=\; \text{CASES } \beta \text{ OF} \\
\qquad \Lambda : \Lambda, \\
\qquad \mathrm{cons}(a, \alpha): \\
\qquad\quad \text{IF } \mathrm{dom}(a) = u \text{ THEN } \alpha \,/\, u \\
\qquad\quad\; \text{ELSE } a \circ (\alpha \,/\, u) \\
\qquad\quad\; \text{ENDIF} \\
\qquad \text{ENDCASES} \\
\quad\; \text{MEASURE } \mathrm{length}(\beta)
\end{array}
$$

$\mathrm{do}(\alpha): \mathcal{S} \;=\; \mathrm{run}(s_0, \alpha)$

$\mathrm{test}(\alpha, a): \mathcal{O} \;=\; \mathrm{output}(\mathrm{do}(\alpha), a)$

$$
\begin{array}{l}
\mathrm{secure}: \mathrm{bool} \;= \\
\quad \forall\, a, u, \alpha:\; u \not\rightsquigarrow \mathrm{dom}(a) \supset \mathrm{test}(\alpha, a) = \mathrm{test}(\alpha \,/\, u, a)
\end{array}
$$

$\mathcal{V}: \text{TYPE+}$

$\mathrm{view}(u, s): \mathcal{V}$

$s \overset{u}{\sim} t: \mathrm{bool} \;=\; \mathrm{view}(u, s) = \mathrm{view}(u, t)$

$$
\begin{array}{l}
\mathrm{output\_consistent}: \mathrm{bool} \;= \\
\quad \forall\, a, s, t:\; s \overset{\mathrm{dom}(a)}{\sim} t \supset \mathrm{output}(s, a) = \mathrm{output}(t, a)
\end{array}
$$

$$
\begin{array}{l}
\mathrm{view\_consistent}: \mathrm{bool} \;= \\
\quad \forall\, u, v, \alpha:\; u \not\rightsquigarrow v \supset \mathrm{do}(\alpha) \overset{v}{\sim} \mathrm{do}(\alpha \,/\, u)
\end{array}
$$

$\mathrm{lemma1}: \text{LEMMA } \mathrm{output\_consistent} \wedge \mathrm{view\_consistent} \supset \mathrm{secure}$

$\mathrm{local\_respect}: \mathrm{bool} \;=\; \forall\, v, s, a:\; \mathrm{dom}(a) \not\rightsquigarrow v \supset s \overset{v}{\sim} \mathrm{step}(s, a)$

$$
\begin{array}{l}
\mathrm{step\_consistent}: \mathrm{bool} \;= \\
\quad \forall\, u, s, t, a:\; s \overset{u}{\sim} t \supset \mathrm{step}(s, a) \overset{u}{\sim} \mathrm{step}(t, a)
\end{array}
$$

$\mathrm{lemma3}: \text{LEMMA } \mathrm{local\_respect} \wedge \mathrm{step\_consistent} \supset \mathrm{view\_consistent}$

$\mathrm{unwinding}: \text{THEOREM } \mathrm{local\_respect} \wedge \mathrm{step\_consistent} \wedge \mathrm{output\_consistent} \supset \mathrm{secure}$

END noninterference

3.5 Summary

In this chapter we have presented a formal specification and verification of the unwinding theorem for noninterference security policies. We have shown a translation of the mathematical specification into the PVS specification language, and demonstrated the power of the induction strategies in PVS in discharging the required lemmas and theorems.

Here is the output of the PVS command `M-x status-proof-theory` for theory `noninterference`.

```
 Proof summary for theory noninterference
    run_TCC1..............................................proved - complete
    run_TCC2..............................................proved - complete
    refl_trans_rel_TCC1...................................proved - complete
    divide_TCC1...........................................proved - complete
    divide_TCC2...........................................proved - complete
    lemma1................................................proved - complete
    lemma3................................................proved - complete
    unwinding.............................................proved - complete
    Theory totals: 8 formulas, 8 attempted, 8 succeeded.
```

And here is the output of the PVS command `M-x show-proofs-theory` for theory `noninterference`. Notice that it requires just seven user-supplied proof commands to complete this example. The first two commands are needed only because we used `beta by <<` as the measure for the recursive function `run`. We did this for illustration; had we used `length(beta)` as the measure, the well-founded TCC would not have been generated and the termination TCC would have continued to be proved automatically by the default strategy, thereby reducing the number of proof commands to five.

```
Proof scripts for theory noninterference:

noninterference.run_TCC1: proved - complete

("" (LEMMA "list_well_founded[action]") (REPLACE-ETA "list_adt[action].<<"))


noninterference.run_TCC2: proved - complete

("" (TERMINATION-TCC))


noninterference.refl_trans_rel_TCC1: proved - complete

("" (SUBTYPE-TCC))


noninterference.divide_TCC1: proved - complete

("" (TERMINATION-TCC))


noninterference.divide_TCC2: proved - complete

("" (TERMINATION-TCC))
```

```
noninterference.lemma1: proved - complete

("" (GRIND))

noninterference.lemma3: proved - complete

("" (EXPAND "view_consistent") (FLATTEN) (INDUCT-AND-SIMPLIFY "alpha"))

noninterference.unwinding: proved - complete

("" (GRIND :DEFS NIL :REWRITES ("lemma1" "lemma3")))
```

Bibliography

[AH96] M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. In *IEEE Real-Time Technology and Applications Symp. (RTAS'96)*, Boston MA, June 1996. IEEE Computer Society Press.

[ALW93] M. Aagaard, M. E. Leeser, and P. J. Windley. Toward a super duper hardware tactic. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications (6th International Workshop, HUG '93)*, pages 399–412, Vancouver, Canada, August 1993. Number 780 in Lecture Notes in Computer Science, Springer-Verlag.

[But93] Ricky W. Butler. An elementary tutorial on formal specification and verification using PVS 2. NASA Technical Memorandum 108991, NASA Langley Research Center, Hampton, VA, June 1993. Revised June 1995. Available, with PVS specification files, from `http://atb-www.larc.nasa.gov/ftp/larc/PVS-tutorial`; use only files marked "revised.".

[CD96] Judith Crow and Ben L. Di Vito. Formalizing Space Shuttle software requirements. In *First Workshop on Formal Methods in Software Practice (FMSP '96)*, pages 40–48, San Diego, CA, January 1996. Association for Computing Machinery.

[CM95] Victor A. Carreño and Paul S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In *HOL95: Eighth International Workshop on Higher-Order Logic Theorem Proving and Its Applications*, Aspen Grove, UT, September 1995. Category B proceedings, available from `http://lal.cs.byu.edu/lal/hol95/Bprocs/indexB.html`.

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995. Available, with specification files, from `http://www.csl.sri.com/wift-tutorial.html`.

[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.

[Di 96] Ben L. Di Vito. Formalizing new navigation requirements for NASA's space shuttle. In *Formal Methods Europe FME '96*, pages 160–178, Oxford, UK, March 1996. Number 1051 in Lecture Notes in Computer Science, Springer-Verlag.

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

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the Symposium on Security and Privacy*, pages 11–20, Oakland, CA, April 1982. IEEE Computer Society.

[GM84] J. A. Goguen and J. Meseguer. Inference control and unwinding. In *Proceedings of the Symposium on Security and Privacy*, pages 75–86, Oakland, CA, April 1984. IEEE Computer Society.

[Har96] John Harrison. A Mizar mode for HOL. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs '96*, pages 203–220, Turku, Finland, August 1996. Number 1125 in Lecture Notes in Computer Science, Springer-Verlag. To appear.

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

[HY87] J. Thomas Haigh and William D. Young. Extending the noninterference version of MLS for SAT. *IEEE Transactions on Software Engineering*, SE-13(2):141–150, February 1987.

[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.

[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.

[Lei69] A. C. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol*. Gordon and Breach Science Publishers, New York, NY, 1969.

[LR93] Patrick Lincoln and John Rushby. Formal verification of an algorithm for interactive consistency under a hybrid fault model. In Courcoubetis [Cou93], pages 292–304.

[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.

[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.

[ORS95] Sam Owre, John Rushby, and Natarajan Shankar. Analyzing tabular and state-transition specifications in PVS. Technical Report SRI-CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1995. Available, with specification files, from `http://www.csl.sri.com/csl-95-12.html`.

[ORSS94] S. Owre, J. M. Rushby, N. Shankar, and M. K. Srivas. A tutorial on using PVS for hardware verification. In Kumar and Kropf [KK94], pages 258–279.

[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 1996.

[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 1996.

[PD96] Seungjoon Park and David L. Dill. Verification of the FLASH cache coherence protocol by aggregation of distributed transactions. In *8th ACM Symposium on Parallel Algorithms and Architectures*, pages 288–296, Padua, Italy, June 1996.

[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.

[RRV95] Sreeranga Rajan, P. Venkat Rangan, and Harrick M. Vin. A formal basis for structured multimedia collaborations. In *Proceedings of the 2nd IEEE International Conference on Multimedia Computing and Systems*, pages 194–201, Washington, DC, May 1995. IEEE Computer Society.

[RSS95] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, pages 84–97, Liege, Belgium, June 1995. Volume 939 of *Lecture Notes in Computer Science*, Springer-Verlag.

[RSS96] H. Rueß, N. Shankar, and M. K. Srivas. Modular verification of SRT division. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, pages 123–134, New Brunswick, NJ, July/August 1996. Number 1102 in Lecture Notes in Computer Science, Springer-Verlag.

[Rud92] Piotr Rudnicki. An overview of the MIZAR project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pages 311–330, Båstad, Sweden, June 1992. The complete proccedings are available from `http://www.cs.chalmers.se/pub/cs-reports/baastad.92/`, this particular paper is also available separately at `http://web.cs.ualberta.ca/~piotr/Mizar/MizarOverview.ps`.

[Rus] John Rushby. *PVS Bibliography*. Menlo Park, CA. Constantly updated; available at `http://www.csl.sri.com/pvs-bib.html`.

[Rus92] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report SRI-CSL-92-2, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1992.

[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. Revised July 1996.

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

[SLR78] Jay M. Spitzen, Karl N. Levitt, and Lawrence Robinson. An example of hierarchical design and proof. *Communications of the ACM*, 21(12):1064–1075, December 1978.

[SM95] Mandayam K. Srivas and Steven P. Miller. Formal verification of the AAMP5 microprocessor. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, chapter 7, pages 125–180. Prentice Hall, Hemel Hempstead, UK, 1995.

[SM96] Mandayam K. Srivas and Steven P. Miller. Applying formal verification to the AAMP5 microprocessor: A case study in the industrial use of formal methods. *Formal Methods in Systems Design*, 8(2):153–188, March 1996.

[SO96] N. Shankar and Sam Owre. *PVS Semantics*. Computer Science Laboratory, SRI International, Menlo Park, CA, 1996. In preparation.

[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 1996.

[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.

Appendix A

Ascii Listings of the Specifications

A.1 Ascii Listing of the Airline Reservation Specifications

These listings were produced by the PVS `M-x alltt-importchain` command.

A.1.1 Theory basic_defs

```
basic_defs: THEORY
BEGIN

  nrows:  posnat            % Max number of rows
  nposits: posnat           % Max number of positions per row

  row:  TYPE = {n: posnat | 1 <= n AND n <= nrows} CONTAINING 1
  position: TYPE = {n: posnat | 1 <= n AND n <= nposits} CONTAINING 1

  flight: TYPE              % Flight identifier
  plane: NONEMPTY_TYPE      % Aircraft type
  preference: TYPE          % Position preference
  passenger: NONEMPTY_TYPE  % Passenger identifier

  seat_assignment: TYPE = [# seat: [row, position],
                             pass: passenger #]

  flight_assignments: TYPE = set[seat_assignment]

  flt_db: TYPE = [flight -> flight_assignments]

  initial_state(flt : flight): flight_assignments =
              emptyset[seat_assignment]
```

```
% ====================================================================
% Definitions that define attributes of a particular airplane
% ====================================================================

  seat_exists: pred[[plane, [row, position]]]
  meets_pref: pred[[plane, [row, position], preference]]
  aircraft: [flight -> plane]

END basic_defs
```

A.1.2 Theory ops

```
ops: THEORY

BEGIN

  IMPORTING basic_defs

  flt:    VAR flight
  pas:    VAR passenger
  db:     VAR flt_db
  a,b:    VAR seat_assignment
  pref:   VAR preference
  seat:   VAR [row,position]

  Cancel_assn(flt,pas,db): flt_db =
          db WITH [(flt) := {a | member(a,db(flt)) AND pass(a) /= pas}]

  pref_filled(db,flt,pref) : bool =
       FORALL seat:  meets_pref(aircraft(flt), seat, pref)
                  IMPLIES (EXISTS a: member(a, db(flt))
                                    AND seat(a) = seat)

  Next_seat: [flt_db, flight, preference -> [row,position]]

  Next_seat_ax: AXIOM
        NOT pref_filled(db, flt, pref) IMPLIES
                seat_exists(aircraft(flt),Next_seat(db,flt,pref))

  Next_seat_ax_2: AXIOM
        (FORALL a: member(a,db(flt)) IMPLIES
                seat(a) /= Next_seat(db,flt,pref))
```

```
  Next_seat_ax_3: AXIOM
        NOT pref_filled(db, flt, pref) IMPLIES
                meets_pref(aircraft(flt),Next_seat(db,flt,pref),pref)

  pass_on_flight(pas,flt,db): bool =
        EXISTS a: pass(a) = pas AND member(a,db(flt))

  Make_assn(flt,pas,pref,db): flt_db =
        IF pref_filled(db, flt, pref) OR pass_on_flight(pas,flt,db) THEN
              db
        ELSE LET a = (# seat := Next_seat(db,flt,pref), pass := pas #) IN
              db WITH [(flt) := add(a, db(flt))]
        ENDIF

  Lookup(flt,pas,db):  [row,position] =
          seat(epsilon( {a | member(a,db(flt)) AND pass(a) = pas}))

% ========================================================================
%                             Invariants
% ========================================================================

  existence(db): bool =
        FORALL a,flt: member(a, db(flt)) IMPLIES
                seat_exists(aircraft(flt), seat(a))

  uniqueness(db): bool =
        FORALL a,b,flt: member(a, db(flt)) AND member(b, db(flt))
           AND pass(a) = pass(b) IMPLIES a = b

  one_per_seat(db): bool =
       FORALL a,b,flt: member(a, db(flt)) AND member(b, db(flt))
           AND seat(a) = seat(b) IMPLIES a = b


  db_invariant(db): bool =
        existence(db) AND uniqueness(db)

  Cancel_assn_inv: THEOREM
        db_invariant(db) IMPLIES db_invariant(Cancel_assn(flt,pas,db))

  MAe: THEOREM
        existence(db) IMPLIES existence(Make_assn(flt,pas,pref,db))
```

```
  MAu: THEOREM
        uniqueness(db) IMPLIES uniqueness(Make_assn(flt,pas,pref,db))

  Make_assn_inv: THEOREM
        db_invariant(db) IMPLIES db_invariant(Make_assn(flt,pas,pref,db))

  initial_state_inv: THEOREM
        db_invariant(initial_state)

% =======================================================================
%                    Invariants Left To Reader
% =======================================================================

  Cancel_inv_one_per_seat: THEOREM
        one_per_seat(db) IMPLIES one_per_seat(Cancel_assn(flt,pas,db))


  Make_inv_one_per_seat: THEOREM
        one_per_seat(db) IMPLIES one_per_seat(Make_assn(flt,pas,pref,db))

  initial_one_per_seat: THEOREM
        one_per_seat(initial_state)

% =======================================================================
%                   Putative Theorems
% =======================================================================

  Make_Cancel: THEOREM
        NOT pass_on_flight(pas,flt,db) IMPLIES
                Cancel_assn(flt,pas,Make_assn(flt,pas,pref,db)) = db


  % <<<<<< Following left to the reader >>>>>>

  Cancel_putative: THEOREM
          NOT (EXISTS (a: seat_assignment):
             member(a,Cancel_assn(flt,pas,db)(flt)) AND pass(a) = pas)

  Make_putative: THEOREM
        NOT pref_filled(db, flt, pref) IMPLIES
          (EXISTS (x: seat_assignment):
            member(x, Make_assn(flt, pas, pref, db)(flt)) AND pass(x) = pas)

  Lp2_lem: LEMMA
```

```
        NOT (pref_filled(db, flt, pref) OR pass_on_flight(pas,flt,db))
                IMPLIES Next_seat(db, flt, pref) =
                        seat(epsilon({a: seat_assignment |
                              Make_assn(flt, pas, pref, db)(flt)(a)
                                AND pass(a) = pas}))

  Lookup_putative: THEOREM
        NOT (pref_filled(db, flt, pref) OR
                                  pass_on_flight(pas,flt,db)) IMPLIES
      meets_pref(aircraft(flt),
                 Lookup(flt, pas, Make_assn(flt,pas,pref,db)),
                 pref)

END ops
```

A.2 Ascii Listing of the Noninterference Specifications

These listings were produced by the PVS `M-x alltt-pvs-file` command.

A.2.1 Theory K_Conversion

```
K_conversion[T1, T2: TYPE]: THEORY
BEGIN
  t1: VAR T1
  t2: VAR T2

  K(t1)(t2): T1 = t1

  CONVERSION  K

END K_conversion
```

A.2.2 Theory `noninterference`

```
noninterference: THEORY
BEGIN

  state, action, output: TYPE+
  s0: state
  s, t: VAR state
  a, b: VAR action
```

```
  step(s, a): state
  output(s, a): output

  action_list: type = list[action]
  alpha, beta: VAR action_list


 ;o(a, alpha): action_list = cons(a, alpha)

  run(s, beta): RECURSIVE state =
    CASES beta OF
      null: s,
      cons(a, alpha): step(run(s, alpha), a)
    ENDCASES
   MEASURE beta by <<

  domain: TYPE+
  u, v: VAR domain

IMPORTING K_conversion
 % IMPORTING  more_preds

   refl_trans_rel: TYPE = (reflexive?[domain] AND transitive?[domain]) CONTAINING eq[domain]

  security_policy: refl_trans_rel
  |>(u, v): bool = NOT  security_policy(u, v)

  dom(a): domain

  /(beta, u): RECURSIVE action_list =
    CASES beta OF
      null: null,
      cons(a, alpha): IF dom(a) = u THEN alpha / u
                      ELSE a o (alpha / u) ENDIF
    ENDCASES
   MEASURE length(beta)

  do(alpha): state = run(s0, alpha)

  test(alpha, a): output = output(do(alpha), a)

  secure: bool = FORALL a, u, alpha:
    u |> dom(a) => test(alpha, a) = test(alpha / u, a)
```

```
  V: TYPE+

  view(u, s): V

  view_equiv(u)(s, t): bool = view(u, s) = view(u, t)

  output_consistent: bool = FORALL a, s, t:
    view_equiv(dom(a))(s, t) => output(s, a) = output(t, a)

  view_consistent: bool = FORALL u, v, alpha:
    u |> v => view_equiv(v)(do(alpha), do(alpha / u))

  lemma1: LEMMA
    output_consistent AND view_consistent => secure

  local_respect: bool = FORALL v, s, a:
    dom(a) |> v => view_equiv(v)(s, step(s, a))

  step_consistent: bool = FORALL u, s, t, a:
    view_equiv(u)(s, t) => view_equiv(u)(step(s, a), step(t, a))

%  lemma2: LEMMA step(do(alpha),a) = do(a o alpha)

  lemma3: LEMMA
    local_respect AND step_consistent => view_consistent

  unwinding: THEOREM
    local_respect AND step_consistent AND output_consistent => secure

END noninterference
```

Appendix B

A More Advanced Specification for the Seat Reservation Problem

Ricky Butler's Seat Reservation Problem is introduced in a report deliberately described as an "elementary tutorial" for PVS [But93]. Chapter 2 of the present report shows how more advanced theorem proving methods can be applied to that example; this appendix further "upgrades" the example by reformulating its specification using more advanced features of the PVS language. The particular specification presented here was stimulated by a version developed by Piotr Rudnicki of the University of Alberta using the "Mizar" system.

Mizar is a very interesting system for mechanized mathematics developed in Poland over a period of 20 years under the leadership of Andrzej Trybulec. The system provides a language for formalized mathematics based on Tarski-Grothendieck set theory, and a proof checker driven by very detailed, but quite readable, proof scripts. A remarkable amount of mathematics has been formalized in Mizar and recorded in the *Journal of Formalized Mathematics*. Information about this journal and about Mizar are available from a Web page maintained by Piotr Rudnicki at `http://web.cs.ualberta.ca/~piotr/Mizar/`. Rudnicki notes that Mizar is "notorious for its lack of documentation"; we have found the most useful and accessible sources to be [Har96] and [Rud92].

Piotr Rudnicki has developed three treatments of the Seat Reservation Problem in Mizar; they are available from URL `http://web.cs.ualberta.ca/~piotr/Mizar/FLT_DB/`. The first follows our rendition of Ricky Butler's original specification (as presented in Chapter 2) quite closely; the second and third use some of the more elaborate types available in Mizar to present a much more sophisticated treatment. Rudnicki asked whether PVS could support a similar approach, and this appendix is our response. It uses the predicate subtypes and dependent types of PVS to develop a specification in which the invariants proposed by Ricky Butler are enforced automatically by the PVS type system (this technique is explained, using a simpler example, in another PVS tutorial [COR+95, pp. 20–24 (example `phone4`)]).

To begin, we introduce `seats`, `flights`, `planes`, `preferences`, and `passengers` as uninterpreted, nonempty types, and also introduce some variables of those types.

```
seats, flights, planes, preferences, passengers: TYPE+

s: VAR seats
flt: VAR flights
p: VAR planes
pref: VAR preferences
pass: VAR passengers
```

Unlike Ricky Butler's treatment, where seat positions are characterized by pairs of the form `[row, position]` and `row` and `position` are positive integers, our new treatment is more abstract and uses the uninterpreted type `seats` for this purpose. The seat positions that exist on a given plane will then be some set of seats that is particular to that kind of plane. We could specify this by an uninterpreted function `seats_on_plane` as follows.

```
seats_on_plane: [plane -> setof[seats]]
```

An equivalent way to write this is the following.

```
seats_on_plane(p): setof[seats]
```
12

You are probably familiar with interpreted functions being specified in this "applicative" form, with an `=` sign and a definition following the type specification, but it can also be used without these to introduce uninterpreted functions.

The specification as given in 12 allows the seat positions on a given plane to be the empty set. This causes difficulty later when we need to show that some of the function types we construct are nonempty. (For example, `Next_seat` will return a seat position; if the plane concerned has none, then we have a contradiction; PVS excludes this possibility by generating TCCs that can only be discharged if `seats_on_plane(p)` is nonempty.) We therefore change the specification to require that all planes have a nonempty set of seat positions. (As well as being necessary, this is also reasonable—if a plane has no seats, why is it present in a database for allocating seats?)

```
seats_on_plane(p): (nonempty?[seats])
```
13

The predicate `nonempty?` is defined in the prelude theory `sets`. The full name of the instance of this predicate that we need here is `sets[seats].nonempty?`, but PVS allows predicate (or other) names from the theory to replace the theory name when no ambiguity results. Now `nonempty?[seats]` is a *predicate* on seats; by enclosing it in parentheses, we change it to a *type*: namely the subtype of seats satisfying the `nonempty?` predicate. Thus, the declaration of 13 is equivalent to the following.

```
seats_on_plane(p): {ss: setof[seats] | nonempty?(ss)}
```

We will use this construction frequently in the rest of this specification.

The requirement that `seats_on_plane(p)` be nonempty generates the following TCC to show that such a function exists.

```
% Existence TCC generated (line 52) for seats_on_plane(p): (nonempty?[seats])
seats_on_plane_TCC1: OBLIGATION
      (EXISTS (x: [planes -> (nonempty?[seats])]): TRUE)
```

It is discharged by the following proof, which in turn requires that `seats` be a nonempty type.

```
(INST 1 "lambda (x:planes): fullset[seats]")
(GRIND)
(INST -1 "epsilon! (x:seats): true")
```

In this proof, the first `(INST...)` supplies the function that associates the set of all seats (`fullset[seats]`, defined in the prelude theory `sets`) with each plane. Then `(GRIND)` expands definitions and simplifies, leaving us to establish that `fullset[seats]` is not empty. This requires that we exhibit a value of type `seats`, which is accomplished by the second `(INST...)`.

Next, we introduce an uninterpreted function called `aircraft` that gives the plane used for a given flight, and then construct the nonempty set `seats_on_flight(flt)` that returns the seat positions that exist on the plane used for flight `flt`.

```
  aircraft(flt): planes
  seats_on_flight(flt): (nonempty?[seats]) = seats_on_plane(aircraft(flt))
```

This definition does not generate a TCC because `seats_on_plane` is already known to have the correct type.

We now declare the function `meets_pref` that takes a plane `p` and a preference `pref` and returns the (possibly empty) set of seat positions on the plane that meet the preference.

```
  meets_pref(p, pref): setof[(seats_on_plane(p))]
```

This is an example of a *dependent* type: the return type of the function depends on the value of its first argument. Notice how, unlike that of Chapter 2, this specification guarantees that only seat positions that exist on the plane concerned are valid members of the set returned by `meets_pref`. Notice, too, that this set may be empty—indicating that no seat positions are acceptable on a given plane.

The database `flight_db` is defined as a function that takes a flight and returns a *partial injection* from seat positions on that flight to passengers.

```
  flight_db:
      TYPE = [flt: flights -> (part_inj[(seats_on_flight(flt)), passengers])]
```

We will look at how the partial injections are defined in terms of the predicate `part_inj` shortly, but the crucial point is that a partial injection associates some of the seat positions on the flight with some of the passengers in such a way that at most one passenger is booked into any given seat (i.e., it is a partial *function* from seats on the plane to passengers), and at most one seat is booked for any given passenger (i.e., it is an *injection*). This pushes the invariants (`existence`, `uniqueness`, and `one_per_seat`) of Ricky Butler's specification into the *type* associated with the database. PVS will then enforce those invariants automatically by generating appropriate TCCs for the functions that construct values of type `flight_db`.

We now turn to the specification of partial injections. PVS is a type theory in which *total* functions are primitive. It is easy to define the total injections as a predicate subtype of the functions (this is done in the PVS prelude), but partial functions are slightly more difficult. One way to define a partial function from `A` to `B`, say, is as the following dependently typed record.

```
part_fun: TYPE = [# dom: setof[A], fun: [(dom) -> B] #]
```

Application of a `part_fun pf` to an argument `x` is then written `fun(pf)(x)`.[1] This can be made more attractive by using a conversion.

```
pf: VAR part_fun

pfun_appl(pf): [(dom(pf)) -> B] = LAMBDA (x:(dom(pf))): fun(pf)(x)

CONVERSION pfun_apply
```

With this construction, we can write simply `pf(x)`.

It is perfectly feasible to extend this construction to specify the partial injections, but for variety, and to be closer to the Mizar set-theoretic treatment, we will use an alternative approach here. This is specified in the theory `rel_as_fun` shown in Figure B.1. In set theory, (partial) functions are just special kinds of relations, so this theory begins by identifying relations from `A` to `B` with predicates on the pair `[A, B]`. The domain and range of a relation are defined in the obvious manner, and then the predicates `functional` and `injective` are defined. These identify those relations that have the special property of being a function, or an injection, respectively. Those relations that have both these properties are the *partial injections*, specified by the predicate `part_inj`, and its associated predicate subtype `(part_inj)`.

We define the empty partial injection `null_inj` and then the functions `reldel_1` and `reldel_2`. The former takes a partial injection from `A` to `B`, and a value `a` of type `A`, and returns a new partial injection from which all pairs of the form `(a, x)` (there can have been at most one) have been removed; the latter is defined dually for values `x` of type `B`. These three functions generate the following TCCs, which ensure that the values they construct really are partial injections.

[1] PVS will generate a TCC to ensure `member(x, dom(pf))`.

```
rel_as_fun[A: TYPE, B: TYPE]: THEORY
BEGIN

  a, b: VAR A
  x, y: VAR B

  rel: TYPE = pred[[A, B]]
  R: VAR rel

  domain(R): setof[A] = {a | EXISTS x: R(a, x)}
  range(R): setof[B] = {x | EXISTS a: R(a, x)}

  functional(R): bool = FORALL a, x, y: R(a, x) & R(a, y) => x = y
  injective(R): bool = FORALL a, b, x: R(a, x) & R(b, x) => a = b

  part_inj(R): bool = functional(R) AND injective(R)

  null_inj: (part_inj) = emptyset[[A, B]]

  reldel_1((R: (part_inj)), a): (part_inj) = {(b, y) | R(b, y) AND a /= b}
  reldel_2((R: (part_inj)), x): (part_inj) = {(b, y) | R(b, y) AND x /= y}

  apply((R: (part_inj)), (a: (domain(R)))):
      (range(R)) = choose! (x: (range(R))): R(a, x)

  invapply((R: (part_inj)), (x: (range(R)))):
      (domain(R)) = choose! (a: (domain(R))): R(a, x)

  update_ok: LEMMA
      LET newR = add((a, x), R) IN
         part_inj(R) AND NOT member(a, domain(R)) AND NOT member(x, range(R))
              IMPLIES part_inj(newR)
                  AND apply(newR, a) = x AND invapply(newR, x) = a

END rel_as_fun
```

Figure B.1: Partial Injections Defined as a Subtype of Relations

```
% Subtype TCC generated (line 17) for  emptyset[[A, B]]
null_inj_TCC1: OBLIGATION part_inj(emptyset[[A, B]])

% Subtype TCC generated (line 19) for  {(b, y) | R(b, y) AND a /= b}
reldel_1_TCC1: OBLIGATION
      (FORALL (R: (part_inj), a): part_inj({(b, y) | R(b, y) AND a /= b}))

% Subtype TCC generated (line 20) for  {(b, y) | R(b, y) AND x /= y}
reldel_2_TCC1: OBLIGATION
      (FORALL (R: (part_inj), x): part_inj({(b, y) | R(b, y) AND x /= y}))
```

The first of these is proved automatically by the default `(subtype-tcc)` strategy. The other two need a little guidance; a suitable proof for the second is the following.

```
(GRIND :IF-MATCH NIL)
 (("1" (HIDE -1 1 2) (REDUCE))
  ("2" (HIDE -2 1) (REDUCE)))
```

The third is proved similarly. The `(HIDE...)` commands remove formulas that would otherwise cause PVS's heuristic instantiation to pick the wrong match.

The function `apply` takes a partial injection and an element of its domain and returns the corresponding element of its range; `invapply` works dually—given an element of the range, it returns the corresponding element of the domain. These definitions generate the following TCCs (which are proved automatically by the default `(subtype-tcc)` strategy) to ensure that the values they return really are in the range and domain, respectively, of the partial injection concerned.

```
% Subtype TCC generated for  LAMBDA (x: (range(R))): R(a, x)
apply_TCC1: OBLIGATION
      (FORALL (R: (part_inj), a: (domain(R))):
         nonempty?[(range(R))](LAMBDA (x: (range(R))): R(a, x)))

% Subtype TCC generated for  LAMBDA (a: (domain(R))): R(a, x)
invapply_TCC1: OBLIGATION
      (FORALL (R: (part_inj), x: (range(R))):
         nonempty?[(domain(R))](LAMBDA (a: (domain(R))): R(a, x)))
```

The lemma `update_ok` says that if we take a partial injection `R` and values `a` of type `A`, not in the domain of `R`, and `x` of type `B`, not in the range of `R`, and add the association `(a, x)` to `R`, then we obtain a new relation `newR` that *is* a partial injection and, furthermore, `apply(newR,a) = x` and `invapply(newR, x) = a`. (The function `add` is from the prelude theory `sets`.) This lemma generates the following TCCs.

```
% Subtype TCC generated (line 31) for  a
update_ok_TCC1: OBLIGATION
      (FORALL (R: rel, a: A, newR, x: B):
         newR = add[[A, B]]((a, x), R)
             AND
           (part_inj(R)
               AND NOT member[A](a, domain(R)) AND NOT member[B](x, range(R)))
               AND part_inj(newR)
             IMPLIES domain(newR)(a))

% Subtype TCC generated (line 31) for  x
update_ok_TCC2: OBLIGATION
      (FORALL (R: rel, a: A, newR, x: B):
         newR = add[[A, B]]((a, x), R)
             AND
           (part_inj(R)
               AND NOT member[A](a, domain(R)) AND NOT member[B](x, range(R)))
               AND part_inj(newR) AND apply(newR, a) = x
             IMPLIES range(newR)(x))
```

The first of these is discharged by

```
(SKOSIMP) (HIDE -2 -3 1 2) (GRIND)
```

and the second by a similar proof.

The lemma itself requires the following proof. The `stew` strategy disposes of the first conjunct in the conclusion and generates a subgoal for each of the other two conjuncts; these are proved by appeal to the `epsilon_ax` axiom from the prelude.

```
(STEW :LAZY-MATCH T :IF-MATCH ALL)
(("1"
  (USE "epsilon_ax[(domain(add((a!1, x!1), R!1)))]")
  (("1" (REDUCE)) ("2" (INST 1 "a!1") (REDUCE))))
 ("2"
  (HIDE -1 -2)
  (USE "epsilon_ax[(range(add((a!1, x!1), R!1)))]")
  (("1" (REDUCE :IF-MATCH ALL)) ("2" (INST 1 "x!1") (REDUCE)))))
```

Having defined the partial injections, we import the theory `rel_as_fun` and proceed with the main specification. We define an initial database in which every flight has no seats assigned and then define the function `pass_on_flight` which tells us whether a given passenger has a seat booked on a given flight in the database.

```
IMPORTING rel_as_fun

initial_db(flt): (part_inj[(seats_on_flight(flt)), passengers]) = null_inj

db: VAR flight_db

pass_on_flight(pass, flt, db): bool = member(pass, range(db(flt)))

seat_filled_on_flight(flt, db, (s: (seats_on_flight(flt)))): bool =
  member(s, domain(db(flt)))

pref_filled(db, flt, pref): bool =
  FORALL (s: (seats_on_flight(flt))):
    meets_pref(aircraft(flt), pref)(s) => seat_filled_on_flight(flt, db, s)
```

We also define a dual function `seat_filled_on_flight` that tells us whether a given seat has been booked on a given flight in the database. Notice that because of the dependent typing, the arguments to this function are given in a different order than those for `pass_on_flight`. (PVS typechecks from left to right, and therefore needs to encounter the argument `flt` before the dependently typed `(s: (seats_on_flight(flt)))`.) We then define the predicate `pref_filled`, which is `true` when all seats meeting a given preference have been filled on a given flight. This definition generates the following TCC, which is discharged automatically by the default `(subtype-tcc)` strategy.

```
% Subtype TCC generated (line 85) for  s
pref_filled_TCC1: OBLIGATION
      (FORALL (db, flt, s: (seats_on_flight(flt))):
         (seats_on_plane(aircraft(flt)))(s));
```

Now we can define the first of the functions that update the database. `Cancel_assn` deletes any booking for a given passenger on a given flight from the database.

```
Cancel_assn(flt, pass, db): flight_db =
  db WITH [(flt) := reldel_2(db(flt), pass)]
```

Since `reldel_2` is known to return a partial injection, no TCC needs to be generated to ensure that `Cancel_assn` maintains the properties of the database.

To develop the function `Make_assn` that creates a new booking in the database, we begin with the function `Next_seat`, which returns a vacant seat on a given flight matching a given preference (if possible). In Chapter 2, we specified this as an uninterpreted function and then constrained its properties by means of axioms. The danger with axioms is that they may be inconsistent—or, at least, not demonstrably consistent—as was the case in Chapter 2 before we added the `new_ax` axiom. If we were to give a definition to the `Next_seat` function, then we will be assured of its consistency (provided all its TCCs are proved), but may suggest an implementation when we really only want to indicate constraints. Definitions involving choose or epsilon suffer less than others from this disadvantage, but they complicate theorem proving (it is generally necessary either to cite epsilon_ax, or to prevent expansion of the definition and to cite a lemma). Also, the fact that epsilon returns an arbitrary member of its type when the predicate is unsatisfiable is an obstacle to many readers.

Fortunately, the type system of PVS is sufficiently rich that it provides a way to escape the horns of this dilemma: we indicate properties the function should possess in its type. (In essence the return type will be a dependent predicate subtype equivalent to the conjunction of the axioms intended to constrain the function.) Like an axiomatic treatment, this approach indicates constraints without suggesting an implementation; unlike an axiomatic treatment, however, we are assured of soundness because PVS will generate a TCC that forces us to exhibit a function of the required type (which is equivalent to demonstrating consistency of the axioms). The TCC can be discharged by constructions involving choose or epsilon—but unlike a definitional specification, it is clear that these are removed from the main line of the specification and are not suggestive of an implementation. In addition to these benefits, this approach assists automated theorem proving because the properties of the function are recorded in its type, where the theorem prover can make productive use of them.

In the case of Next_seat, we write its specification as follows.

```
  Next_seat(db, flt, (pref: {p:preferences | NOT pref_filled(db,flt,p)})):     14
    { (s: (seats_on_flight(flt))) |
                       meets_pref(aircraft(flt), pref)(s)
                           AND NOT seat_filled_on_flight(flt, db, s)}
```

This says that if not all the seats satisfying the given preference have been booked, then Next_seat returns a seat position that does exist on the plane concerned, that meets the preference, and that is not already booked. Notice the dependent typing for the third argument to Next_seat; this is similar to that which we saw in Chapter 2 for the function Lookup, and is needed for a similar reason: to ensure that the return type is not empty. PVS generates the following two TCCs from this declaration.

```
% Subtype TCC generated (line 89) for  s
Next_seat_TCC1: OBLIGATION
      (FORALL (flt, (s: (seats_on_flight(flt)))):
         (seats_on_plane(aircraft(flt)))(s));

% Existence TCC generated (line 87) for
    %  Next_seat(db, flt,
    %           (pref: {p: preferences | NOT pref_filled(db, flt, p)})):
    %     {((s: (seats_on_flight(flt)))) | meets_pref(aircraft(flt), pref)(s)
    %           AND NOT seat_filled_on_flight(flt, db, s)}
    %
Next_seat_TCC2: OBLIGATION
 (EXISTS (x: [d:
    [db: flight_db, flt: flights, {p: preferences | NOT pref_filled(db, flt, p)}]
         -> {((s: (seats_on_flight(PROJ_2(d)))))
                  | meets_pref(aircraft(PROJ_2(d)), PROJ_3(d))(s)
                       AND NOT seat_filled_on_flight(PROJ_2(d), PROJ_1(d), s)}]):
         TRUE);
```

The first of these is discharged automatically by the default (subtype-tcc) strategy. The second, which requires us to demonstrate that the function type specified for Next_seat is inhabited, is discharged by the following proof.

```
(INST 1 "LAMBDA
  (db, flt, (pref: p:preferences| not pref_filled(db,flt,p))):
        choose! (s: (seats_on_flight(flt))):
            meets_pref(aircraft(flt), pref)(s)
                AND NOT seat_filled_on_flight(flt, db, s)")
(("1" (GRIND)) ("2" (GRIND)))
```

The (INST...) command constructs a suitable function using choose; notice that the substitution can be developed by a simple transformation on the type given for Next_seat. The two (GRIND) commands discharge TCC subgoals generated by the instantiation.

Our specification departs from Piotr Rudnicki's with this treatment of Next_seat. His specification can be approximately rendered as follows,

```
  Next_seat_variant(db, flt, pref): (seats_on_flight(flt)) =
    epsilon! (s: (seats_on_flight(flt))):
      meets_pref(aircraft(flt), pref)(s)
          AND NOT seat_filled_on_flight(flt, db, s)
```

although he uses an auxiliary predicate flight_pref that returns the seat positions on the aircraft satisfying the given preference; his version of Next_seat then removes the seats already booked and chooses one of the remainder. Next_seat_variant does it slightly different order by restricting the initial choice to those seats that are not already filled.

The type given for Next_seat in $\boxed{14}$ ensures that the axioms required in the treatment of Chapter 2 are provable as lemmas here. They can be stated as follows.

```
  Next_seat_ax: LEMMA
        NOT pref_filled(db, flt, pref)
           IMPLIES member(Next_seat(db, flt, pref), seats_on_flight(flt))

  Next_seat_ax_2: LEMMA
        NOT pref_filled(db, flt, pref)
          IMPLIES NOT seat_filled_on_flight(flt, db, Next_seat(db, flt, pref))

  Next_seat_ax_3: LEMMA
        NOT pref_filled(db, flt, pref)
            IMPLIES meets_pref(aircraft(flt), pref)(Next_seat(db, flt, pref))
```

All of these are proved by `(grind)`, though this is something of a sledgehammer given the richness of the type information that PVS has available. More surgical proofs are `(expand "member")` for the first and `(skosimp)(assert)` for the other two.

`Next_seat_ax_3` generates the following TCC.

```
% Subtype TCC generated (line 90) for  Next_seat(db, flt, pref)
Next_seat_ax_3_TCC1: OBLIGATION
      (FORALL (db: flight_db, flt: flights, pref: preferences):
         NOT pref_filled(db, flt, pref)
             IMPLIES
           (seats_on_plane(aircraft(flt)))(Next_seat(db, flt, pref)))
```

This is very similar to `Next_seat_ax` and can be discharged by the following proof.

```
(USE "Next_seat_ax") (EXPAND "seats_on_flight") (EXPAND "member")
```

Finally, we can define the function that adds a seat booking to the database.

```
  Make_assn(flt, pass, pref, db): flight_db =
    IF pref_filled(db, flt, pref) OR pass_on_flight(pass, flt, db)
      THEN db
      ELSE db WITH [(flt) := add((Next_seat(db, flt, pref), pass), db(flt))]
    ENDIF
```

If the preference is filled or the passenger is already on the flight, the database is left unchanged, otherwise the seat returned by `Next_seat` is booked for the passenger on the flight concerned. A TCC is generated to ensure that the type constraints on the application of `Next_seat` are satisfied.

```
% Subtype TCC generated (line 107) for  pref
Make_assn_TCC1: OBLIGATION
      (FORALL (db, flt, pass, pref):
         NOT (pref_filled(db, flt, pref) OR pass_on_flight(pass, flt, db))
             IMPLIES NOT pref_filled(db, flt, pref));
```

This is discharged automatically by the default (subtype-tcc) strategy.

Because the database for each flight is specified to be a partial injection, a second TCC is generated to ensure that this construction preserves the required properties (namely, that each passenger has at most one seat and each seat at most one passenger).

```
% Subtype TCC generated (line 107) for
        add((Next_seat(db, flt, pref), pass), db(flt))
Make_assn_TCC2: OBLIGATION
      (FORALL (db, flt, pass, pref):
         NOT (pref_filled(db, flt, pref) OR pass_on_flight(pass, flt, db))
             IMPLIES
           part_inj[(seats_on_flight(flt)),passengers]
                 (add[[((seats_on_flight(flt))), passengers]]
                         ((Next_seat(db, flt, pref), pass),db(flt))));
```

This TCC is equivalent to Make_assn_inv in the treatment of Chapter 2. Unlike that treatment, however, where we had to realize for ourselves that we ought to check that our specification preserves such an invariant, the present treatment generates it automatically as a proof obligation needed to ensure type-correctness. The proof is a straightforward expansion of definitions followed by appeal to the lemmas update_ok and Next_seat_ax_2.

```
(SKOSIMP)
(TYPEPRED "db!1(flt!1)")
(STEW :EXCLUDE ("domain" "range" "apply" "invapply" "part_inj")
   :LEMMAS ("update_ok[(seats_on_flight(flt!1)), passengers]" "Next_seat_ax_2"))
```

The :EXCLUDE... simply speeds up the proof by preventing rewriting of definitions from the theory rel_as_fun.

The "challenge" theorems are essentially identical to those of Chapter 2, except that Cancel_putative is stated more neatly.

```
Make_Cancel: THEOREM
     NOT pass_on_flight(pass, flt, db)
         IMPLIES Cancel_assn(flt, pass, Make_assn(flt, pass, pref, db)) = db

Cancel_putative: THEOREM
      NOT pass_on_flight(pass, flt, Cancel_assn(flt, pass, db))

Make_putative: THEOREM
      NOT pref_filled(db, flt, pref)
          IMPLIES pass_on_flight(pass, flt, Make_assn(flt, pass, pref, db))

Lookup(flt, pass, (db: d: flight_db | pass_on_flight(pass, flt, d))):
    (seats_on_flight(flt)) = invapply(db(flt), pass)

Lookup_putative: THEOREM
      NOT((pref_filled(db, flt, pref) OR pass_on_flight(pass, flt, db)))
          IMPLIES
        meets_pref(aircraft(flt),
                 pref)(Lookup(flt, pass, Make_assn(flt, pass, pref, db)))
```

The function `Lookup` and the challenge theorem `Lookup_putative` generate the following TCCs. The first ensures that the passenger really is on the flight concerned (which follows by the dependent typing in the definition of `Lookup`); the second ensures that the database update performed by `Make_assn` in `Lookup_putative` satisfies the dependent type restriction in `Lookup`; the third ensures that the seat returned by `Lookup` really does exist on the plane concerned.

```
% Subtype TCC generated (line 122) for  pass
Lookup_TCC1: OBLIGATION
      (FORALL (flt, pass, db: d: flight_db | pass_on_flight(pass, flt, d)):
         range[((seats_on_flight(flt))), passengers](db(flt))(pass));

% Subtype TCC generated (line 128) for  Make_assn(flt, pass, pref, db)
Lookup_putative_TCC1: OBLIGATION
      (FORALL (db: flight_db, flt: flights,
               pass: passengers, pref: preferences):
         NOT(((pref_filled(db, flt, pref) OR pass_on_flight(pass, flt, db))))
             IMPLIES
           pass_on_flight(pass, flt, Make_assn(flt, pass, pref, db)));

% Subtype TCC generated (line 128) for
        Lookup(flt, pass, Make_assn(flt, pass, pref, db))
Lookup_putative_TCC2: OBLIGATION
      (FORALL (db: flight_db, flt: flights,
               pass: passengers, pref: preferences):
         NOT(((pref_filled(db, flt, pref) OR pass_on_flight(pass, flt, db))))
             IMPLIES
           (seats_on_plane(aircraft(flt)))(Lookup(flt,
                                                  pass,
                                                  Make_assn(flt,
                                                            pass, pref, db))));
```

The first of these is proved by the default `(subtype-tcc)` strategy; the second appeals to the theorem `Make_putative`.

```
(SKOSIMP) (USE "Make_putative") (ASSERT)
```

The third follows by the type predicate associated with the return type of `Lookup`.

```
(SKOSIMP)
(TYPEPRED "Lookup(flt!1, pass!1, Make_assn(flt!1, pass!1,pref!1, db!1))")
 (("1" (EXPAND "seats_on_flight"))
  ("2" (USE "Make_putative") (ASSERT)))
```

The challenge theorems themselves are proved in the following manner.

```
new_flight_db.Make_Cancel:

 (SKOSIMP)
 (APPLY-EXTENSIONALITY :HIDE? T)
 (APPLY-EXTENSIONALITY :HIDE? T)
 (TYPEPRED "x!2")
 (GRIND)


new_flight_db.Cancel_putative:

 (GRIND)


new_flight_db.Make_putative:

 (GRIND) (INST? 3 :WHERE 3) (ASSERT) (REDUCE)


new_flight_db.Lookup_putative:

 (SKOSIMP)
 (TYPEPRED "db!1(flt!1)")
 (TYPEPRED "Next_seat(db!1, flt!1, pref!1)")
 (("1"
   (STEW :EXCLUDE ("domain" "range" "apply" "invapply" "part_inj")
     :LEMMAS ("update_ok[(seats_on_flight(flt!1)),passengers]")))
  ("2" (SKOSIMP) (TYPEPRED "s!1") (EXPAND "seats_on_flight")))
```

Some of these proofs are slightly more complicated than the corresponding ones in Chapter 2; the additional complexity is generally due to the need to expand a few definitions in order to expose a term whose type predicate is required, or to discharge a TCC side-condition. In other examples, however, the presence of rich type information often simplifies proofs, and increases their automation. This approach has been exploited to very good effect in PVS libraries developed at NASA Langley by Ricky Butler and Paul Miner.

In general, the style of specification illustrated here is worth mastering. Definitions and lemmas can often be stated more simply when a lot of information is provided implicitly in the types, and the whole process of specification is made less error-prone because the PVS typechecker can provide powerful assistance in the form of TCCs.