Writing PVS Proof Strategies*

Sam Owre and Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
{owre, shankar}@csl.sri.com
URL: http://www.csl.sri.com/{ owre, shankar}
Phone: +1 (650) 859-{5114, 5272} Fax: +1 (650) 859-2844

Abstract. PVS (Prototype Verification System) is a comprehensive framework for writing formal logical specifications and constructing proofs. An interactive proof checker is a key component of PVS. The capabilities of this proof checker can be extended by defining proof strategies that are similar to LCF-style tactics. Commonly used proof strategies include those for discharging typechecking proof obligations, simplification and rewriting using decision procedures, and various forms of induction. We describe the basic building blocks of PVS proof strategies and provide a pragmatic guide for writing sophisticated strategies.

1 Introduction

Writing correct proofs is an activity that combines creativity and tedium. The creative aspect of proof development is in the construction of definitions, lemmas, and theorems, the choice of high-level proof ideas, and in recovering gracefully from failed proof attempts. The tedium is in checking that all the low-level details have been worked out correctly. Automated proof checkers are meant to verify the low-level proof steps corresponding to the high-level proof guidance given interactively. Automated theorem provers, on the other hand, are required to discover both the high-level outline and the low-level details required to prove or refute a given conjecture. Such theorem provers have yet to achieve the level of sophistication needed to reliably tackle conjectures with interesting mathematical content. Early proof checkers required proofs to be given entirely in terms of low-level inferences (such as modus ponens or instantiation) [McC62,dB80,BB74]. The second generation of proof checkers included a language for defining compound proof steps that could be justified solely in terms of primitive inferences. PVS builds on these prior approaches. PVS employs an expressive specification language based on higher-order logic with a type system that includes predicate subtypes, dependent types, and abstract datatypes.

* Funded by Naval Research Laboratory Contract N00173-00-C-2086 and National Aeronautical and Space Agency Contract NASA NAG-1-02101.

These features not only allow mathematical ideas to be captured with cogency, but they also interact synergistically with the inference procedures. PVS allows complex proof strategies to be built up from quite sophisticated primitive inference steps that employ arithmetic decision procedures, rewriting, and simplification. The advantage of the PVS approach is that it exploits the efficiency of modern automated deduction technologies in the construction of powerful and flexible proof strategies. The drawback is that the trusted code base is fairly large since it includes the typechecker and several complex inference procedures.

PVS has a simple language for defining proof strategies. A number of PVS users have used the PVS strategy language for defining customized proof strategies for a variety of applications. Typically, a user builds up a significant body of domain knowledge in a field like finite set theory, analysis, graph theory, algebra, or trigonometry. Proofs in specific applications use this domain knowledge in a stylized format. Proof strategies are defined to package such patterns of usage so that they can be used by non-experts. The PC/DC system [SS94] provided a front-end to PVS that contained various proof strategies for reasoning with a real-time interval temporal logic called the duration calculus. The TAME system [Arc00] from the US Naval Research Laboratory provides a collection of custom proof strategies for carrying out proofs of I/O automata at a level of detail that is reasonably faithful to the original hand proofs. The LOOP project [vdBJ01] at the University of Nijmegen is another example of a substantial investment in PVS proof strategies for automating proofs of Java code. Work on PVS strategies at the NASA Langley Research Center has yielded the *Manip* package [Vit02] for algebraic simplification strategies and the `Field` package [MM01] (modeled on the eponymous Coq library) for simplifying subgoals involving real arithmetic.

User-defined proof strategies are thus an important mechanism for customizing the proof-checking capabilities of PVS toward specific domains. This paper is a brief tutorial on writing advanced proof strategies in PVS. It is directed primarily at PVS users who are interested in achieving greater levels of automation and customization. We first provide some background on proof checking in general (Section 2), and on PVS in particular (Section 3). Some of the PVS internal data structures are reviewed in Section 4. Section 5 introduces the strategy language. We explain the construction of some simple proof strategies in Section 6, and cover more advanced techniques in Section 7. Conclusions and future directions are sketched in Section 8. Due to space limitations, the discussion of strategies and PVS interfaces here contains many gaps. A larger document [OS03] covering the PVS application programmer interface is currently under development.

2 Background

Automated proof checking has an illustrious history. In the seventeenth century, Gottfried Leibniz had already conceived of a language in which knowledge could be systematized so that a logic engine could be used to resolve arguments. A similar fancy inspired Boole in the development of Boolean algebra. The mechanization of mathematics started to seem more realistic with the formalization of various branches of mathematics at the dawn of the twentieth century through the work of Dedekind, Peano, Cantor, Frege, Russell, Whitehead, and Hilbert. At the beginning of his celebrated article on the incompleteness theorem [Göd92], Gödel explicitly acknowledges the possibility of mechanically checking mathematical proofs. Turing's article *Computing Machinery and Intelligence* [Tur63] also proposed the use of computers as proof engines. Bush's famous article *As We May Think* [Bus45] asserts the centrality of verified reasoning in scientific computing.[1]

Automated reasoning was actively investigated in the 1950s through the work of Davis, Newell, Shaw, and Simon, Wang, Gilmore, and Prawitz. These works were not concerned with proof checking. The earliest work on this topic is due to McCarthy [McC62] in the 1960s. The AUTOMATH project was initiated by de Bruijn [dB80,NGdV94] in the mid-1960s and introduced many key ideas. Jutting [vBJ79] used AUTOMATH to verify Landau's *Foundations of Analysis* [Lan60]. Bledsoe's IMPLY system [BB74] was developed during the late 1960s and early 1970s and applied to proofs in set theory and analysis. The LCF family of systems [GMW79] includes such systems as Nuprl [CAB+86], HOL [GM93], Coq [CCF+95], Isabelle [Pau94], HOL-Lite [Har00], and LEGO [LP92]. LCF is best known for introducing the ML programming language [GMM+77,MTH90] as a way of defining proof tactics and tacticals. The Mizar proof checker [Rud92] constitutes one of the most sustained and coordinated efforts at mechanizing a large body of mathematics.

3 Brief Overview of PVS

Work on the PVS proof checker began at SRI International in 1990. PVS has been strongly influenced in its design by its immediate predecessor, the EHDM system [EHD93]. PVS also builds on the prior work in automated proof checking, especially the work of Bledsoe and the LCF family of systems, the work by Shostak [Sho84] and Nelson and Oppen [NO79] on ground decision procedures, and the proof strategies employed by the Boyer–Moore theorem prover [BM79,BM88]. Like HOL and EHDM, the PVS specification language is based on classical higher-order logic but with added features like predicate subtypes, dependent types, and abstract datatypes. Features similar to subtypes and dependent types also appear in other logics, but in PVS, the decision procedures provide crucial support for processing specifications that exploit these features. With predicate subtyping, typechecking is undecidable in general, but the PVS typechecker verifies simple type correctness and generates proof obligations corresponding to the subtypes. These proof obligations can be proved

[1] To quote Bush: *Logic can become enormously difficult, and it would undoubtedly be well to produce more assurance in its use. ... We may some day click off arguments on a machine with the same assurance that we now enter sales on a cash register.*

automatically or interactively, and the majority of them succumb easily to simple proof strategies that rely heavily on the PVS decision procedures.

We will use the simple example of the language equivalence between deterministic and nondeterministic finite automata to illustrate both the PVS language and proof strategies. A PVS specification is a collection of theories. A theory is a list of declarations of types, constants, and formulas. The declarations of types and constants can include definitions. Declarations without definitions are said to be *uninterpreted*. A theory can also take parameters that are types, individuals, or (instances of) theories. The `DATATYPE` declaration `list` introduces an abstract datatype with two constructors: `null` representing the empty list, and `cons` which adds an element to the front of a list. The accessors corresponding to `cons` are `car`, which returns the leading element, and `cdr` which represents the remainder of the list minus the leading element. The `list` datatype when typechecked, generates several theories that contain a various axioms and operations, including induction principles and recursion operators. The list datatype is introduced in the PVS prelude which contains formalizations of a number of basic datatypes.

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

The theory `DFA` formalizes deterministic automata where the number of states is not necessarily finite. The states of the automata are drawn from the uninterpreted type `state` in which there is a distinguished `start` state, and a designated set of final states `final?`. The type `set[state]` is an abbreviation for the predicate type `[state -> bool]`. The automaton operates on an alphabet `Sigma`, and the transition function `delta` maps a given alphabet and source state to a target state. The operation `DELTA` iterates `delta` and is defined to take a `string` of alphabets from `Sigma` and a source state and return a target state. `DAccept?` is a predicate that accepts a string if the final state returned by `DELTA` is a valid final state.

```
DFA : THEORY
  BEGIN
       Sigma : TYPE
       state : TYPE
       start : state
       delta : [Sigma -> [state -> state]]
       final? : set[state]

   DELTA((string : list[Sigma]))((S : state)):
             RECURSIVE state =
     (CASES string OF
         null : S,
         cons(a, x): delta(a)(DELTA(x)(S))
      ENDCASES)
     MEASURE length(string)

   DAccept?((string : list[Sigma])) : bool =
       final?(DELTA(string)(start))

  END DFA
```

The theory `NFA` for nondeterministic automata is similar to `DFA`. The type of `ndelta` differs from that of `delta` in returning a set of states rather than a single state. The recursive operation `NDELTA` similarly processes a string with respect to a state to return a set of states. The nondeterministic automaton accepts this string if the set of states returned by `NDELTA` contains a final state.

```
NFA           : THEORY
  BEGIN
       nSigma : TYPE
       nstate : TYPE
       nstart : nstate
       ndelta : [nSigma -> [nstate -> set[nstate]]]
       nfinal? : set[nstate]

   NDELTA((string : list[nSigma]))((s : nstate)) :
           RECURSIVE set[nstate] =
       (CASES string OF
         null : singleton(s),
         cons(a, x): lub(image(ndelta(a), NDELTA(x)(s)))
        ENDCASES)
     MEASURE length(string)

   Accept?((string : list[nSigma])) : bool =
     (EXISTS (r : (nfinal?)) :
       member(r, NDELTA(string)(nstart)))

  END NFA
```

The language equivalence between the two automaton is captured by the theory `equiv`. The `NFA` theory is imported into the theory `equiv`. The symbols declared in `NFA` are used to create an instance of `DFA` that corresponds to the *subset construction* used to show the equivalence. Here, the alphabet `Sigma` is interpreted as `nSigma`, the `state` type is interpreted as the power set of the type `nstate`, and `start`, `delta`, and `final?` are also suitably defined. The resulting interpretation of the theory `DFA` is used to show the equivalence between `NFA` and `DFA` in two steps. The lemma `main` states the equivalence between `NDELTA` and the interpreted `DELTA` operation. The theorem `equiv` states the equivalence between the strings accepted by the `NFA` and those accepted by the corresponding `DFA`.

```
equiv: THEORY
  BEGIN
   IMPORTING NFA
    NFADFA : THEORY =
             DFA{{Sigma = nSigma,
                 state = set[nstate],
                 start = singleton(nstart),
                 delta((symbol : nSigma))((S : set[nstate])) =
                             lub(image(ndelta(symbol), S)),
                 final?((S : set[nstate]))  =
                       (EXISTS (r : (nfinal?)) : member(r, S))}}


   main: LEMMA
    (FORALL (x : list[nSigma]), (s : nstate):
       NDELTA(x)(s) = DELTA(x)(singleton(s)))

   equiv: THEOREM
    (FORALL (string : list[nSigma]):
       Accept?(string) IFF DAccept?(string))
  END equiv
```

The first proposition, `main`, is proved by invoking the `induct-and-simplify` strategy to employ list induction on the parameter `x`. The second proposition, `equiv`, is is proved by employing the `grind` strategy to apply rewrite rules, simplification using the decision procedures, and heuristic quantifier instantiation.

```
main :

  |-------
{1}   (FORALL (x: list[nSigma]), (s: nstate):
         NDELTA(x)(s) = DELTA(x)(singleton(s)))

Rule? (induct-and-simplify "x")
NDELTA rewrites NDELTA(null)(s!1)
  to singleton(s!1)
DELTA rewrites DELTA(null)(singleton(s!1))
  to singleton(s!1)
NDELTA rewrites NDELTA(cons(cons1_var!1, cons2_var!1))(s!1)
  to lub(image(ndelta(cons1_var!1), NDELTA(cons2_var!1)(s!1)))
DELTA rewrites DELTA(cons(cons1_var!1, cons2_var!1))(singleton(s!1))
  to lub(image(ndelta(cons1_var!1), DELTA(cons2_var!1)(singleton(s!1))))
By induction on x, and by repeatedly rewriting and simplifying,
Q.E.D.
```

```
equiv :

  |-------
{1} (FORALL (string: list[nSigma]): Accept?(string) IFF DAccept?(string))

Rule? (grind :theories "equiv")
main rewrites NDELTA(string)(nstart)
  to DELTA(string)(singleton(nstart))
member rewrites member(r, DELTA(string)(singleton(nstart)))
  to DELTA(string)(singleton(nstart))(r)
Accept? rewrites Accept?(string)
  to EXISTS (r: (nfinal?)): DELTA(string)(singleton(nstart))(r)
member rewrites member(r, DELTA(string)(singleton(nstart)))
  to DELTA(string)(singleton(nstart))(r)
DAccept? rewrites DAccept?(string)
  to EXISTS (r: (nfinal?)): DELTA(string)(singleton(nstart))(r)
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.
```

The inner workings of the `grind` strategy are described in Section 6, and those of `induct-and-simplify` are explained in Section 7.

4 PVS Data Structures

In writing sophisticated PVS strategies, it is useful to have a basic understanding of the way specifications are represented in PVS. Most data are maintained in the form of CLOS (Common Lisp Object System) objects. The appropriate classes are defined using a Lisp macro `(defcl` *`classname`* `(`*`superclasses`*`)` *`slots`*`)`. Typical classes are

1. `module`: Contains declarations and judgements corresponding to a PVS theory. The expression `(get-theory "foo")` returns the theory module named `foo`, and `(show (get-theory "foo"))` displays the slots and their contents.
2. `type-decl`: Type declaration.
3. `formula-decl`: Formula declaration.
4. `funtype`: Function type.
5. `name-expr`: Name expression, i.e., constants or variables.
6. `application`: Application expressions.

4.1 Proof State

PVS proofs employ Gentzen's sequent calculus as the basic representation. A PVS sequent has of the form

$$\begin{array}{cl} \{-1\} & \mathit{antecedentformula}_1 \\ & \vdots \\ [-m] & \mathit{antecedentformula}_m \\ \vdash & \\ \{1\} & \mathit{succedentformula}_1 \\ & \vdots \\ [n] & \mathit{succedentformula}_n \end{array}$$

Here, the negatively numbered formulas are the antecedents of the sequent, and the positively numbered formulas are the succedents. Proofs operate by reducing a goal sequent to subgoal sequents in response to a proof command. Formulas in a subgoal sequent that appear in the parent sequent are numbered within square brackets, and the newly introduced formulas are numbered within braces. Internally, the proof state is a CLOS object with slots including the `current-goal` sequent, the `parent-proofstate`, and the active `current-subgoal`. The current goal is a sequent whose main slot `s-formulas` holds a list of `s-forms`. The `s-forms` are themselves CLOS objects with a `formula` field that contains the PVS expression corresponding to a sequent formulas. The antecedent formulas are those that are negated. The list of `s-forms` interleaves both antecedent and succedent formulas. The proof state also contains fields corresponding to the parent proofstate and the subgoal proof states. The *current* proof state within a proof is accessible through the global variable `*ps*`. The Lisp command `(show ob)` displays the values of the slots of a CLOS object `ob`.

5 The Strategy Language

The core language for defining strategies is quite simple, but this does not cover the large number of syntactic and semantic operations that are required for writing more sophisticated strategies. A PVS proof command is either a primitive proof command such as `flatten`, `split`, `auto-rewrite`, or `simplify`, or a compound strategy that is constructed from smaller proof commands. PVS does allow new primitive inferences to be added, but such additions must be carried out with circumspection since they can introduce unsoundness. Strategies, on the other hand, are conservative, since it is possible to verify the validity of the proof when all the strategies have been expanded into primitive proof steps.

The primitive proof commands in PVS include

1. `flatten` for disjunctive simplification.
2. `split` for conjunctive splitting.
3. `skolem` for eliminating universal-strength quantifiers.
4. `inst` for instantiating existential-strength quantifiers.
5. `auto-rewrite` for installing rewrite rules for use during simplification.
6. `simplify` for simplification using rewriting and ground decision procedures.

PVS strategies can either be in glassbox form so that only the expanded form of the strategy is visible in the resulting subproof, or in blackbox form where it is applied as a single atomic proof step and the internal steps are not recorded. The Common Lisp constructs for defining strategies are:

1. (`defstrat` *name arguments body help-string format-string*): Defines a glassbox strategy named *name* with arguments given in *arguments*. The arguments are given as a list of required and optional arguments, where the optional ones are preceded by the keyword `&optional`. The definition is given in *body*. The *help-string* contains the documentation for the proof command, and *format-string* is a Lisp format control string that is applied to the arguments to generate the commentary that appears when the proof command is applied. The *help-string* and *format-string* are optional.
2. (`defrule` *name arguments body help-string format-string*): Defines a blackbox strategy that is otherwise similar to `defstrat`.
3. (`defstep` *name arguments body help-string format-string*): Defines a blackbox strategy named *name* and a glassbox version named *name$*.

The language in which the strategies are defined involves just a few constructs:

1. (`if` *lisp-expr strat-expr1 strat-expr2*): Returns the value of *strat-expr2* if the evaluation of Common Lisp expression *lisp-expr* (relative to the current proof state) returns `nil`, and the value of *strat-expr1*, otherwise.
2. (`try` *strat-expr1 strat-expr2 strat-expr3*): First applies *strat-expr1* to the current proof state. This could either
   (a) Have no effect, in which case, *strat-expr3* is invoked.

(b) Complete the subproof and *strat-expr2* and *strat-expr3* are not used.
(c) Generate a failure, which is propagated to the parent proof state.
(d) Generate subgoals, and *strat-expr2* is applied to these subgoals, and *strat-expr3* is not evaluated.

3. `(let ((`*var1 lisp-expr1*`)...(`*var1 lisp-expr1*`))` *strat-expr*`)`: Binds *vari* to the value of *lisp-expri* in *strat-expr*.
4. `(skip)`: Does nothing.
5. `(fail)`: Signals failure to trigger backtracking.
6. `(quote` *strat-expr*`)`: Evaluates to *strat-expr* but is useful when the strategy is constructed as a Lisp s-expression.

Note that `(try (skip) A B)` is equivalent to `B`, whereas `(try (try (fail) A B) C D)` is equivalent to `D`. Definitions can also involve recursion. There are some simple strategies that are analogous to LCF tacticals in that they are used to direct other strategies. The `else` strategy applies `step1`, and backtracks to `step2` if the `step1` does nothing.

```
(defstrat else (step1 step2)
  (try step1 (skip) step2)
  "If step1 fails, then try step2, otherwise behave like step1" )
```

The `repeat` strategy applies `step` to the current goal, and recursively applies the strategy to the first resulting subgoal. It thus repeats a step along the "main" branch of a proof. Recall that the global variable `*ps*` captures the current proofstate relative to which the strategy is being evaluated. The simpler strategy `repeat*` repeats a step along all the branches of a proof. Either of these strategies could fail to terminate so it is important to ensure that they are only applied to steps that eventually do nothing.

```
(defstrat repeat (step)
  (try step (if (equal (get-goalnum *ps*) 1)
                        (repeat step)
                        (skip))
    (skip))
  "Successively apply STEP along main branch until it does nothing.")

(defstrat repeat* (step)
  (try step (repeat* step) (skip))
  "Successively apply STEP until it does nothing.")
```

The propositional simplification strategy applies disjunctive flattening to the sequent and recursively invokes itself on the subgoals. When disjunctive flattening is exhausted, then conjunctive splitting is employed, and again, the strategy is recursively invoked until there are no further top-level disjunctive or conjunctive connectives in the sequent. The recursive invocation of `prop` uses the expansive prop$. This makes it easier to observe the internal behavior by invoking the expansive strategy prop$.

```
(defstep prop ()
  (try (flatten) (prop$) (try (split)(prop$) (skip)))
  "A black-box rule for propositional simplification."
  "Applying propositional simplification")
```

6 Simple Proof Strategies

We now examine the construction of the grind strategy as an instance of a simple proof strategy that combines a number of smaller proof steps. This strategy takes a number of optional arguments with possible default values. The strategy installs rewrite rules from the definitions in the current sequent (and, transitively, the definitions used in these), the given theories and rewrites, but excluding those listed in exclude. This is followed by propositional simplification using the bddsimp command, and assert which carries out simplification using the ground decision procedures and the installed rewrite rules. The command replace* is used to apply the antecedent equalities in the sequent as rewrites. The reduce command (described below) is invoked with a number of arguments in keyword form. In a call to the strategy, the required arguments must be given in order but the optional arguments can be given in keyword form, as illustrated in the call to reduce$.

```
(defstep grind (&optional (defs !)
                  theories rewrites exclude (if-match t)
                 (updates? t) polarity? (instantiator inst?)
               (let-reduce? t))
  (then
   (install-rewrites$ :defs defs :theories theories
                      :rewrites rewrites :exclude exclude)
    (then (bddsimp)(assert :let-reduce? let-reduce?))
    (replace*)
    (reduce$ :if-match if-match :updates? updates?
             :polarity? polarity? :instantiator instantiator
             :let-reduce? let-reduce?))
    "..."
    "Trying repeated skolemization, instantiation, and if-lifting")
```

The reduce command repeatedly applies the bash command and then executes replace* on any subgoals.

```
(defstep reduce (&optional (if-match t)(updates? t) polarity?
                            (instantiator inst?) (let-reduce? t))
    (repeat* (try (bash$ :if-match if-match :updates? updates?
                          :polarity? polarity? :instantiator instantiator
                          :let-reduce? let-reduce?)
              (replace*)
              (skip)))
    "..."
    "Repeatedly simplifying with decision procedures, rewriting,
  propositional reasoning, quantifier instantiation, skolemization,
 if-lifting and equality replacement")
```

The bash command is the core of reduce. It first executes assert, and then uses the if construct to selectively use an instantiator to instantiate any existential-strength quantifiers. The repeat loop contains the command skolem-typepred that introduces constants for universal-strength quantifiers followed by disjunctive flattening. Any embedded conditionals are then lifted to the top level of the sequent with the lift-if command. The updates? flag converts update expressions into conditional form.

```
(defstep bash (&optional (if-match t)(updates? t) polarity?
              (instantiator inst?) (let-reduce? t))
  (then (assert :let-reduce? let-reduce?)(bddsimp)
        (if if-match (let ((command (generate-instantiator-command
                                      if-match polarity? instantiator)))
                       command)(skip))
        (repeat (then (skolem-typepred)(flatten)))
        (lift-if :updates? updates?))
  "..."
  "Simplifying with decision procedures, rewriting, propositional
reasoning, quantifier instantiation, skolemization, if-lifting.")
```

7 Advanced Proof Strategies

We first examine a strategy that while simple still illustrates features that are basic to the more advanced strategies. The strategy replace-extensionality replaces all occurrences of a term f by a term g, where the equality $f = g$ holds by extensionality. The type of f and g must be either a function, record, tuple, or a datatype in order for a suitable extensionality scheme to be available. The optional argument expected? is there in the rare event that the type of f is ambiguous. The optional argument keep? is given as T when the equality $f = g$ is to be retained at the end of the step.

Arguments to strategies that are PVS expressions can be either in the form of concrete syntax as a string or as abstract syntax which is already parsed or even typechecked. Strategies invoked directly by the user often contain arguments in the form of concrete syntax, but those invoked from another strategy may have their arguments in a parsed and typechecked form. The operation `pc-parse` parses the expression if needed and its second argument is the expected non-terminal, usually either `'type-expr` or `'expr`. The operation `typecheck` type-checks the parsed expression relative to a given context. The global variable `*current-context*` binds the context corresponding to the current goal. The function `pc-typecheck` is a variant of `typecheck` that first looks for an occurrence of the given expression in the current sequent. Since the input expression is likely to occur in the sequent, this saves the expense of typechecking. The strategy applies `extensionality` step to the given expected type, if there is one. Otherwise, `extensionality` is applied to the type of the first or second argument. If the extensionality step succeeds, then it adds the appropriate extensionality axiom as the first antecedent formula. This formula is then instantiated with the typechecked forms of the f and g arguments. The instantiated axiom is then subject to conjunctive splitting. The first branch corresponds to the conclusion equality between f and g. The `replace` command is applied to this equality. If the `keep?` argument is `nil`, which is its default value, then equality formula is deleted. The remaining subgoals correspond to the conditions on the instance of the extensionality axiom, and these are discharged by successive applications of `skolem!`, `beta`, and `assert`. The instantiation step might have generated TCCs, and the `assert` step is applied to the subgoals corresponding to these TCCs.

```
(defstep replace-extensionality (f g &optional expected keep?)
  (let ((tt (when expected (typecheck (pc-parse expected 'type-expr)
                                      :context *current-context*))))
    (let ((ff (pc-typecheck (pc-parse f 'expr)
                            :expected tt))
          (gg (pc-typecheck (pc-parse g 'expr)
                            :expected tt)))
      (let ((tf (type ff))
            (tg (type gg)))
        (try (if tt (extensionality tt)
                 (try (extensionality tf)(skip)
                      (extensionality tg)))
             (branch (inst - ff gg)
                     ((branch (split -1)
                              ((then (replace -1)
                                     (if keep? (skip)
                                         (delete -1)))
                               (then* (skolem! 1)
                                      (beta 1);;changed from + to 1.
                                      (assert 1))))
                      (assert)))
             (skip)))))
  "..."
  "Replacing ~a by ~a using extensionality")
```

The apply-extensionality strategy is used to prove a sequent with a consequent equality by employing replace-extensionality to replace the left-hand side of the equality by its right-hand side. The optional argument fnum is + (indicating the consequent formulas) by default. The command first selects the s-forms corresponding to fnum using select-seq. The first equality among these formulas is used as the candidate for applying replace-extensionality. The replace-extensionality step can generate subgoals corresponding to TCCs, and the candidate formula can be deleted from these when the hide? flag is T. The skip-msg is a variant of skip that generates a comment.

```
(defstep apply-extensionality (&optional  (fnum +) keep? hide?)
  (let ((sforms (select-seq (s-forms (current-goal *ps*))
                            (if (memq fnum '(* + -)) fnum
                                      (list fnum))))
        (fmla (loop for sf in sforms thereis
                    (when (equation? (formula sf))
                      (formula sf))))
        (lhs (when fmla (args1 fmla)))
        (rhs (when fmla (args2 fmla))))
    (if fmla
        (try (replace-extensionality$  lhs rhs :keep? keep?)
             (then
              (let ((fnums (find-all-sformnums (s-forms
                                                (current-goal *ps*))
                                               '+
                                               #'(lambda (x)
                                                   (eq x fmla))))
                    (fnum (if fnums (car fnums) nil)))
                (if (and hide? fnum) (delete fnum) (skip)))
              (assert))
             (skip-msg "Couldn't find a suitable extensionality rule."))
        (skip-msg "Couldn't find suitable formula for applying ~
                   extensionality.")))
  "..."
  "Applying extensionality")
```

The last strategy we describe is induct-and-simplify which is used in the DFA-NFA equivalence proof. This strategy is applied to a sequent with a consequent formula that universally quantifies the given variable var. Like grind, the install-rewrites strategy is used to install rewrite rules from the definitions in the formula, the given theories and rewrite rule names. The induct step instantiates the induction scheme: either the one named by name or the one that is appropriate for the variable var, and generates the base and induction steps. These are simplified using repeated application of skolemization, assert, propositional simplification, if-lifting, and instantiation.

```
(defstep induct-and-simplify (var &optional (fnum 1) name
                                  (defs t)
                                  (if-match best)
                                  theories
                                  rewrites
                                  exclude
                                  (instantiator inst?)
                                  )
  (then
   (install-rewrites$ :defs defs :theories theories
                      :rewrites rewrites :exclude exclude)
   (try (induct var fnum name)
        (then
         (skosimp*)
         (assert);;To expand the functions in the induction conclusion
         (repeat (lift-if));;To lift the embedded ifs,
         ;;then simplify, split, then instantiate
         ;;the induction hypothesis.
         (repeat* (then (assert)
                        (bddsimp)
                        (skosimp*)
                        (if if-match
                            (let ((command
                                   (generate-instantiator-command
                                     if-match nil instantiator)))
                              command)
                            (skip))
                        (lift-if))))
        (skip)))
  "..."
  "By induction on ~a, and by repeatedly rewriting and simplifying")
```

The main step in the `induct-and-simplify` is the `induct` command. This strategy first selects the candidate formula using `select-seq` with the input `fnum`. The induction variable is parsed and a new skolem constant is generated for it. This skolem constant is placed in a `skolem-list` corresponding to the outermost bound variables of the formula is generated with blanks (indicated by underscore) for those variables different from `var`. The body of the strategy is described below.

```
(defstep induct (var &optional (fnum 1) name)
  (let ((fmla (let* ((sforms (select-seq (s-forms (current-goal *ps*))
                                         (list fnum))))
                (when sforms
                  (formula (car sforms)))))
        (var (pc-parse var 'name))
        (new-var-symbol (new-sko-symbol var *current-context*))
        (skolem-list (if (forall? fmla)
                         (loop for x in (bindings fmla)
                               collect (if (format-equal var (id x))
                                           new-var-symbol
                                           "_"))
                         nil)))
    [see below])
  "..."
  "Inducting on ~a~@[ on formula ~a~]~@[ using induction scheme ~a~]")
```

If there is a selected formula, the strategy applies `simple-induct` to generate a suitable instance of the induction scheme (determined by the type of `var` or the given `name`). The induction scheme instantiated with the induction formula is beta-reduced using `beta`, instantiated using `inst?`, and conjunctively split using `split`.

```
(if fmla
    (try (simple-induct var fmla name)
         (if *new-fmla-nums*
             (let ((fnum (find-sform (s-forms (current-goal *ps*))
                                     '+
                                     #'(lambda (sform)
                                         (eq (formula sform)
                                             fmla)))))
               (then (beta)
                     (let ((fmla
                            (let ((sforms (select-seq
                                            (s-forms (current-goal *ps*))
                                            (list fnum))))
                              (when sforms (formula (car sforms))))))
                       (then (let ((x (car *new-fmla-nums*)))
                               (then (inst? x)
                                     (split x)))
                             [see below]))))
             (skip))
        (skip-msg "Could not find suitable induction scheme."))
        (let ((msg (format nil "No formula corresponding to fnum ~a"
                     fnum)))
          (skip-msg msg)))
```

The position in the sequent of the original formula where induction was applied, might now be different. This position is recomputed. The formula, which must be universally quantified, is skolemized, and the corresponding universal quantifier in the induction scheme is instantiated with this skolem constant. The induction conclusion is discharged using `prop` leaving the base and induction subgoals. The residue of the induction formula is deleted in these subgoals.

```
(let ((num (find-sform
            (s-forms (current-goal *ps*))
            '+
            #'(lambda (sform)
                (eq (formula sform)
                    fmla)))))
  (if (eql num fnum)
      (then (prop)
            (skolem fnum skolem-list)
            (inst - new-var-symbol)
            (prop))
      (if num (delete num)
          (let ((newnums
                 (loop for n
                       in *new-fmla-nums*
                       when (and (> n 0)
                                 (<= n fnum))
                       collect n))
                (newfnum (+ fnum
                            (length newnums))))
            (delete newfnum)))))
```

8 Conclusions

Proof checkers, like any other usable form of software, must be programmable. User-defined proof strategies are a mechanism for defining common patterns of inference steps as a single proof command. Such defined strategies are conservative since they introduce no new unsoundness into the proof system. PVS proof strategies are thus similar in philosophy to LCF tactics. There are, however, some significant differences with LCF in that the primitive inferences in PVS encompass rewriting and the use of decision procedures. They are therefore much more complex than those typically employed by the LCF family of checkers. The PVS primitive proof commands are neither easily nor efficiently definable by means of tactics. By starting with powerful primitive inferences, it is possible to perform proof construction and strategy definition at a level of detail that is closer to that of a hand-proof.

The core of the PVS strategy language is quite simple but writing effective strategies requires familiarity with Common Lisp and the underlying PVS data structures. The constructs of the strategy language are inspired by the *recursive waterfall* strategy employed by the theorem provers of Boyer and Moore. The `prop` and `ground` strategies are typical of such recursive waterfalls.

Proof checking continues to pose significant challenges. There is still a lot of tedium associated with proof construction. These challenges can be addressed by identifying useful primitive proof steps for building proofs in specific domains, new techniques for building sound and efficient decision procedures, and systematic studies of the strategies that are used in constructing complex proofs. Proof strategies also need to be integrated with formalized libraries of mathematical knowledge. The PVS strategy language can be enhanced by means of a type system and a formal semantics (see Kirchner [Kir03]).

References

[Arc00] Myla Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):139–181, 2000.

[BB74] W. W. Bledsoe and Peter Bruell. A man-machine theorem-proving system. *Artificial Intelligence*, 5:51–72, 1974.

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

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

[Bus45] Vannevar Bush. As we may think. *The Atlantic Monthly*, 1945. Available at `http://www.theatlantic.com/unbound/flashbks/computer/bushf.htm`.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, Englewood Cliffs, NJ, 1986.

[CCF+95] C. Cornes, J. Courant, J.C. Filliatre, G. Huet, P. Manoury, C Paulin-Mohring, C. Munoz, C. Murthy, C. Parent, A. Saibi, and B. Werner. The Coq proof assistant reference manual, version 5.10. Technical report, INRIA, Rocquencourt, France, February 1995.

[dB80] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 589–606. Academic Press, New York, NY, 1980.

[EHD93] Computer Science Laboratory, SRI International, Menlo Park, CA. *User Guide for the* EHDM *Specification Language and Verification System, Version 6.1*, February 1993. Three volumes.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, UK, 1993.

[GMM+77] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in LCF. Technical Report CSR-16-77, Department of Computer Science, University of Edinburgh, 1977.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Göd92] Kurt Gödel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover Publications, Inc., New York, NY,

1992. Translated by B. Meltzer, with an Introduction by R. B. Braithwaite. Originally published as a book in 1962. Article originally published in 1931.

[Har00] John Harrison. High-level verification using theorem proving and formalized mathematics. In David McAllester, editor, *Automated Deduction—CADE-17*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 1–6, Pittsburgh, PA, June 2000. Springer-Verlag.

[Kir03] Florent Kirchner. Coq tacticals and PVS strategies: A small step semantics. In *STRATA'03*, 2003.

[Lan60] E. Landau. *Foundations of Analysis*. Chelsea, New York, NY, 1960. translated from the German original by F. Steinhardt.

[LP92] Z. Luo and R. Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.

[McC62] J. McCarthy. Computer programs for checking mathematical proofs. In *Recursive Function Theory, Proceedings of a Symposium in Pure Mathematics*, volume V, pages 219–227, Providence, Rhode Island, 1962. American Mathematical Society.

[MM01] C. Muñoz and M. Mayero. Real automation in the field. Technical Report NASA/CR-2001-211271 Interim ICASE Report No. 39, ICASE-NASA Langley, ICASE Mail Stop 132C, NASA Langley Research Center, Hampton VA 23681-2199, USA, December 2001.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[NGdV94] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer. *Selected Papers on Automath*. North-Holland, Amsterdam, 1994.

[NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[OS03] S. Owre and N. Shankar. *PVS API Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 2003.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[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 at `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`.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[SS94] Jens U. Skakkebæk and N. Shankar. Towards a Duration Calculus proof assistant in PVS. In 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*, pages 660–679, Lübeck, Germany, September 1994. Springer-Verlag.

[Tur63] Alan Turing. Computing machinery and intelligence. In E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*. McGraw-Hill Book Company, New York, 1963. Originally published in Mind, Vol. LIX. No.236, October, 1950.

[vBJ79] L. S. van Benthem Jutting. Checking Landau's 'Grundlagen' in the Automath system. Technical report, Mathematical Centre, Amsterdam, 1979. Mathematical Centre Tracts.

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312, Genova, Italy, April 2001. Springer-Verlag.

[Vit02] Ben. L. Di Vito. A PVS prover strategy package for common manipulations. Technical Memorandum NASA/TM-2002-211647, NASA Langley Research Center, Hampton, Virginia, April 2002.