NASA/CP–2003–212448

Design and Application of Strategies/Tactics in Higher Order Logics

Edited by

Myla Archer
Naval Research Laboratory, Washington, D.C.

Ben Di Vito
Langley Research Center, Hampton, Virginia

César Muñoz
National Institute of Aerospace, Hampton, Virginia

September 2003

NASA/CP-2003-212448

Design and Application of Strategies/Tactics in Higher Order Logics

Edited by

Myla Archer
Naval Research Laboratory, Washington, D.C.

Ben Di Vito
Langley Research Center, Hampton, Virginia

César Muñoz
National Institute of Aerospace, Hampton, Virginia

Proceedings of a workshop sponsored by the Naval Research Laboratory, the National Aeronautics and Space Administration, and the National Institute of Aerospace and held in Rome, Italy
September 8, 2003

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

September 2003

Preface

The topic of theorem proving strategies has generated considerable interest in the theorem proving community. In particular, there is an ongoing series of workshops associated with CADE entitled Strategies in Automated Deduction, as well as a special issue of *Annals of Mathematics and Artificial Intelligence* with the same title. More information on this workshop series and the special issue can be found at http://www.logic.at/strategies/. Much of the emphasis in the Strategies in Automated Deduction workshops has been on proof strategies and heuristic based proof search/planning in first-order automatic theorem proving.

In principle, first-order theorem proving does not require user guidance; strategies in this context aim at faster proof discovery. By contrast, theorem proving in higher-order logics typically requires interactive user guidance. Thus, although strategies in higher-order logics are also sometimes intended for automatic theorem proving, they are also commonly aimed at providing bases for more efficient interactive proof construction.

Higher order logic theorem provers support interactive reasoning by providing the user with a set of basic proof commands. For reasons of efficiency, the effects of these commands may include reasoning based on decision procedures. This can make the precise effects of these commands on proof goals sometimes hard to anticipate, and thus interfere with fine control in user strategies; however, it does allow simple strategies to be relatively powerful. Moreover, considerable support for fine control in user strategies in higher-order logic theorem provers is provided by their expressive proof scripting languages that include powerful capabilities, e.g., pattern matching formulas in a sequent, or as in PVS, tracking the history of formulas through formula labels.

As a result of the different flavor of theorem proving in higher-order logics versus theorem proving in first order logic, the art of strategy writing in PVS and other higher-order logic theorem provers is worth studying in its own right. Current knowledge about writing PVS strategies in particular has been limited to folklore, despite the powerful capabilities provided in PVS for implementing user-defined strategies. Thus, our first Workshop on The Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003) is focused on PVS. For PVS, we wish to distill what is known from successful efforts, move beyond the folklore stage, and spawn more widespread practice of the strategic arts.

This Proceedings includes both a paper from the implementors of PVS providing guidance for PVS strategy writers and a tutorial on PVS strategy writing distilled from the experience of three PVS users who have written extensive sets of PVS user strategies. Following these are three full papers from the higher-order logic theorem proving community that discuss PVS strategies to enhance arithmetic and other interactive reasoning in PVS; implementing first-order tactics in higher-order provers; and a proposed technique for specifying small step semantics that can be used in multiple higher order logic theorem provers, with illustrations from both Coq and PVS. The Proceedings concludes with three position papers for a panel session that discuss three settings in which development of PVS strategies is worth while.

Organization

STRATA 2003 is organized by the Naval Research Laboratory (NRL), NASA Langley Research Center (NASA LaRC), and the National Institute of Aerospace (NIA).

STRATA 2003 Organizing Committee

Conference Chair:	Myla Archer (NRL)
	Ben Di Vito (NASA)
	César A. Muñoz (NIA)

Table of Contents

Writing PVS Proof Strategies*

Sam Owre and Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
{owre, shankar}@csl.sri.com
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. 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

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

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

[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.*

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 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

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

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 nonterminal, usually either `'type-expr` or `'expr`. The operation `typecheck` typechecks 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.

Developing User Strategies in PVS: A Tutorial

Myla Archer[1], Ben Di Vito[2], and César Muñoz[3]*

[1] Naval Research Laboratory, Washington, DC 20375, USA
archer@itd.nrl.navy.mil
[2] NASA Langley Research Center, Hampton, VA 23681, USA
b.l.divito@larc.nasa.gov
[3] National Institute of Aerospace, Hampton, VA 23666, USA
munoz@nianet.org
http://research.nianet.org/~munoz

Abstract. This tutorial provides an overview of the PVS strategy language, and explains how to define new PVS strategies and load them into PVS, and how to create a strategy package. It then discusses several useful techniques that can be used in developing user strategies, and provides examples that illustrate many of these techniques.

1 Introduction

Why use strategies in PVS? There are several compelling reasons for doing so. We offer a few scenarios below that illustrate productive uses for strategies.

PVS provides a core set of inference rules supplemented by decision procedures and other simplification heuristics. Continuing enhancements to the theorem prover gradually increase the automation available to interactive users. Nevertheless, the level of automation perceived by users is still much lower than desired. This is not a problem peculiar to PVS; similar provers suffer the same limitations. In fact, PVS is among the most automatic of provers in its class.

Strategies provide an accessible means of increasing the automation available to users of the PVS prover. This can be done in generic form, suitable for a wide range of proving tasks, or in specific problem domains, yielding specialized tools suitable only in narrow contexts. Development of strategies can be performed by end users or specialists whose role is to create strategies for use by others. Over time, strategy development can lead to a reusable body of "deductive middleware." An effective division of labor in the overall conduct of mechanical theorem proving is a possible outcome of this process.

In the following, we provide several examples of strategies that are likely to be beneficial to PVS users.

– *Modest strategies to streamline prover use.* This is the simplest category of strategies, typically involving rules with just a few lines of definition. An example would be introducing rules to invoke frequently occurring sequences of proof commands. Consider the sequence `(LIFT-IF)`, `(SPLIT)`, and `(ASSERT)`. One could introduce a strategy named `IF-SPLIT` to carry out this sequence. Such strategies are easy to create, although their benefit is limited to saving the effort of repetitive typing.
– *Extended forms of predefined rules.* A slightly more advanced approach is to identify commonly needed inferences that are guided by user input. By writing strategies that accept arguments, it is possible to create enhanced versions or combinations of rules that already exist in the predefined set provided by PVS. In fact, many of the higher level predefined rules were created using the strategy mechanism. Consider, for example, a rule to claim that the lefthand sides of two formulae are equal, then invoke the appropriate `CASE` command. We might apply such a strategy using `(CLAIM-EQ -1 -3)` where `CLAIM-EQ` is the new proof rule and `-1` and `-3` are the numbers of the sequent formulae to be considered.
– *Algebraic manipulation and arithmetic simplification.* The PVS decision procedures handle linear arithmetic well, but have more difficulty with nonlinear expressions. In such cases, users must apply lemmas from the prelude or other sources. Strategies can be effective at manipulating arithmetic expressions when guided by user input. The package Manip [5], for instance, provides strategies for conducting user-directed manipulations of real-valued expressions. Similarly, the package Field [6] carries out higher level arithmetic reduction with considerable automation.

* Funded under NASA Cooperative Agreement NCC-1-02043.

- *Deduction support for specialized models or specifications.* Verification or analysis tasks based on theorem proving often take place in the context of a specialized model of computation, such as state machines, hybrid automata, etc. Proofs in such contexts often have a stylized character that lends itself to automated proof. By capturing the proof steps and decision processes in the form of strategies, it is possible to provide a great deal of targeted automation to the proof effort. TAME is an example of such an approach within the domain of timed automata.
- *Interfaces to external proof support tools.* Occasionally it is desirable to make use of additional tools that support the prover in the construction of large or difficult proofs. Strategies in this role can be used as a means of accessing the current proof state and exporting information to an external tool. After computing its result, the external tool can supply information to be acted on in some way, such as submitting prover commands. An example would be a tool that performs database searches, then returns the names of suitable lemmas for possible invocation. PVS's `musimp`, `model-check`, and `abstract-and-model-check` strategies are also examples of this approach.
- *Interfaces to support external components through proving.* The support relationship can work in the other direction as well. Under some arrangements, the prover can be used to provide support to an external process. For example, a computer algebra system might wish to consult a theorem prover to confirm that a transformation it needs to perform is valid under certain conditions. This request could be posed as a set of conjectures sent to the prover, where a strategy-guided proof process would attempt to settle the question and return a result.

These suggested uses of PVS are by no means exhaustive. They are realistic, however. Each of these uses has either been implemented or is currently under development. No doubt other applications will be discovered. It is our hope that this tutorial might lead others to investigate new possibilities.

The remainder of this tutorial is organized as follows. Section 2 provides the basic information needed for defining your own strategies and making them available in PVS. Section 3 describes and illustrates a set of techniques that can be used in the development of user strategies. Section 4 provides examples that demonstrate how to use various techniques to develop both strategies that facilitate user interaction with PVS and automatic strategies. Finally, Section 5 discusses some additional support that would be useful in to developers of PVS user strategies.

2 The basics

2.1 PVS commands.

PVS commands can be either *rules* or *strategies.* A *rule* is a command that can be invoked by name and (if appropriate) applied to arguments. Rules execute as atomic steps in the PVS prover. A *strategy* is a command created by using zero or more PVS strategy-building commands to combine rule applications and other strategies. Thus, every rule application is also a (degenerate) strategy. Executing a strategy in the PVS prover causes execution of the sequence of atomic steps needed by the strategy for the current subgoal. On the syntactic level, the heart of a strategy definition is a *strategy expression* built by using strategy-building command names to combine rule names (applied to arguments, which may involve variable names) and other strategy expressions.

A representative set of PVS strategy-building commands is listed in Table 1. For short, we will refer to these commands as *strategicals*, in analogy to the *tacticals* in Coq, HOL, and other theorem provers that are used to combine simpler tactics into more complex ones.

A simple example strategy that is sometimes useful is:

```
(THEN (LIFT-IF) (PROP) (ASSERT) (FAIL))                    (1)
```

Strategy (1) is useful in determining whether straightforward simplification combined with the PVS decision procedures will achieve a goal; if it does not, then the intended behavior of this strategy is to return to the proof subgoal in which it is invoked, without generating any new subgoals. Most simple sequential strategies do not use `(FAIL)`; because it does so, Strategy (1) can behave badly. In particular, it causes full or partial proof failure if none of `(LIFT-IF)`, `(PROP)`, and `(ASSERT)` has an effect. One way to ensure the intended

Strategical	Description
`(APPLY` *step*`)`	Turns *step* into a defined rule.
`(THEN` $step_1$ `...` $step_n$`)`	Applies $step_1$ to $step_n$ in order down all branches.
`(THEN@` $step_1$ `...` $step_n$`)`	Applies $step_1$ to $step_n$ in order down the main proof branch.
`(IF` *lisp-expr* $step_1$ $step_2$`)`	If *lisp-expr* evaluates to *true* then applies $step_1$. Otherwise, applies $step_2$.
`(TRY` $step_1$ $step_2$ $step_3$`)`	Tries $step_1$; if it modifies the proof state then applies $step_2$. Otherwise, applies $step_3$.
`(ELSE` $step_1$ $step_2$`)`	Behaves as `(TRY` $step_1$ `(SKIP)` $step_2$`)`.
`(SPREAD` *step* `(`$step_1$ `...` $step_n$`))`	Applies *step* and spreads $step_1$ to $step_n$ over the new subgoals.
`(BRANCH` *step* `(`$step_1$ `...` $step_n$`))`	Like `SPREAD` but reuses $step_n$ on any extra subgoals.
`(REPEAT` *step*`)`	Iterates *step* until it does nothing down the main proof branch.
`(REPEAT*` *step*`)`	Iterates *step* until it does nothing down all branches.
`(WITH-LABELS` *step* `(`$labs_1$ `...` $labs_n$`))`	Applies *step*; then labels all new formulae in the new subgoals with $labs_1$ to $labs_n$.
`(LET ((`v_1 $lisp\text{-}expr_1$`) ...` `(`v_n $lisp\text{-}expr_n$`))` *step*`)`	Applies a new command that is just like `step`, but where v_i has been replaced by the evaluation of $lisp\text{-}expr_i$ for $1 \leq i \leq n$.

Table 1. PVS strategicals

behavior of Strategy (1) is to use the strategy expression in (1) as the body of a defined rule, as described in Section 2.2. Another way is to "wrap" it with the command `APPLY`, as in:

```
(APPLY (THEN (LIFT-IF) (PROP) (ASSERT) (FAIL)))        (2)
```

Finally, one may catch the action of `FAIL` with the command `TRY`. For more on both `TRY` and the use of wrappers, see Section 3.

Note that the two strategicals `IF` and `LET` allow the introduction of Lisp code into a strategy. Strategies that incorporate Lisp code are more sophisticated than Strategies (1) and (2). The Lisp code generally uses information about the current proof state, though a few useful things can be done by using Lisp code to set and observe global variables. Strategies that use information about the proof state are discussed later in Section 3.

2.2 Defined rules and strategies.

PVS proof rules are of two kinds: *primitive* rules and *defined* rules. Both primitive and defined rules behave like atomic steps when applied to appropriate arguments, but, unlike a primitive rule, a defined rule is derived from a strategy expression. The strategy expression corresponding to a defined rule can be observed in PVS by typing:

```
M-x help-pvs-prover-strategy
```

Also, the documentation string for a strategy can be viewed within the prover via the command `HELP`.

A defined rule is created by applying the PVS macro `defstep`. Paraphrased from the PVS Prover Guide [10, 11], the format for `defstep` is:

```
(defstep name
         parameter-list
         strategy-expression                    (3)
         documentation-string
         format-string )
```

The *parameter-list*, whose precise description can be found in [10, 11], can contain required arguments plus `&optional` and `&rest` parts, rather like the parameter list in a Lisp function definition. The *documentation-string* is generally used to describe the effect of applying the strategy; it is printed interactively as part of the documentation of proof steps that is printed by the "help" facilities of PVS, e.g., when one types `(HELP` *name*`)` during a proof, or `M-x help-pvs-prover` or `M-x help-pvs-prover-strategy` followed by *name* at any time when using PVS. The *format-string* is printed interactively when the defined rule *name succeeds*, i.e., completes the proof of the current goal, or when it returns one or more subgoals. In addition to creating a new defined rule *name*, the macro `defstep` also creates a named strategy *name$*. Variants of `defstep` include `defhelper`, which does not require the *documentation-string* or *format-string* arguments, and `defstrat`, which does not require the *format-string* argument. The macro `defhelper` is intended for defining "internal" auxiliary steps that can be used in other strategies, while `defstrat` defines a strategy without a corresponding (atomic) defined rule.

Strategy (1) can be turned into the defined rule PROP_PROBE using the definition:

```
(defstep PROP_PROBE ()
         (THEN (LIFT-IF) (PROP) (ASSERT) (FAIL))                    (4)
         "Checks for a trivial proof" "By simple reasoning")
```

Once the definition of PROP_PROBE has been loaded into PVS, the desired effect of Strategy (1) can be accomplished by just typing `(PROP_PROBE)` when prompted by PVS for a proof rule. Because Strategy (1) does not refer to any unbound parameter names, the effect of `(PROP_PROBE)` is equivalent to that of Strategy (1) wrapped in `(APPLY ...)`. The exact effect of Strategy (1), in which one sees all the steps in the reasoning, can be duplicated by typing `(PROP_PROBE$)` when prompted for a rule.

By allowing the possibility of parameters, the macro `defstep` allows a strategy (as well as its corresponding defined rule) to be applied in an environment where the parameter names are bound to specific values. The *format-string* in the definition of a rule with parameters can refer to these parameters: any inclusion of ~a in the format string is replaced by the value of an actual parameter, with successive ~a's picking up successive parameters.

A simple example of a new rule with all these features is the rule suppose, whose definition is in Figure 1.[4] The rule suppose incorporates formula labeling and comments into the simplest version of the PVS command

```
(defstep suppose (x)
  (let ((suppstring (format nil "Suppose ~a" x))
        (nsuppstring
          (format nil "Suppose not [~a]" x)))
  (branch (with-labels (case x) (("Suppose")("Suppose not")))
          ((comment suppstring) (comment nsuppstring))))
  "For doing a simple case split and tracking the cases"
  "First supposing ~a true and then supposing it false")
```

Fig. 1. Definition of a rule with a parameter and a *format-string* that refers to it.

CASE. The strategy expression body of suppose uses the strategicals LET, WITH-LABELS, and BRANCH. With LET, it incorporates Lisp code that computes two comment strings. Using WITH-LABELS, it applies the labels from the first list ("Suppose") to new formulae in the first new subgoal, and the labels from the second list ("Suppose not") to new formulae in the second new subgoal. Since each of the first and second new subgoals have just one new formula, and these new formulae represent, respectively, the meanings of x and (NOT x), they are labeled appropriately. The second argument of BRANCH is a list of two commands, which will be applied respectively to the new subgoals. Each of these commands adds its argument as a comment in the subgoal to which it is applied; this comment will appear above the sequent when the subgoal is displayed. Each comment will also be recorded in the saved proof at the beginning of the new proof branch starting at its associated proof goal. The use of labels and comments will be discussed further in Section 3.

[4] Though we use a mixture of upper and lower case versions of names in this tutorial, it is safest to use only lower case in actual strategy files; see the PVS release notes at http://pvs.csl.sri.com.

2.3 Adding new rules and strategies to PVS.

Once you have defined one or more new rules using `defstep`, `defstrat`, or `defhelper`, you can make your new rule(s) available in PVS by saving the definition(s) in a file named `pvs-strategies` and putting it in the PVS context where you wish to use the new rules. The file `pvs-strategies` does not need to be a physical file, it can be a link to a file containing your definitions. This way, you can keep a set of definitions consistent across several contexts.

The file `pvs-strategies` is loaded when the first proof in a session is being started, or when a new proof is being started after the content of `pvs-strategies` has been changed. Because `pvs-strategies` is loaded into Lisp, it can contain arbitrary Lisp code—not only rule definitions, but function definitions, global variable initializations, load commands, etc. One use of a load command (that is in fact employed by TAME) is to load a set of strategies specific to one context that can be generated from some theory in that context. Further, if `common_strat` is a file containing a set of strategies that you use in all your developments, you can load those strategies by putting the line

```
(load "<PATH>/common_strat")
```

in the file `pvs-strategies`, where *<PATH>* is the path where the file `common_strat` is found. Section 3 describes some possible uses of functions and global variables.

For testing purposes, one can introduce strategy definitions directly from the command line:

```
(LISP (DEFSTEP strat-name ...))
```

To redefine one later, recall the previous command input using `M-s` or `M-r`, then edit the definition and resubmit it. This technique allows for quick tests or explorations of small strategies.

2.4 Creating a strategy package.

If a set of definitions is general enough to be used in several developments or to be used by other PVS users, you may want to pack them as a prelude library extension. The basic functionality of prelude library extensions has been available in older versions of PVS. However, it became fully operational and simple to use in PVS 3.1. A prelude library extension is a set of PVS theories, strategies, and Lisp code that are available to the user as if they were part of the PVS prelude context. As the developer of a prelude library extension, make a directory *MyPackage* and put the following files in it:

- Files `*.pvs` containing PVS theories that your development requires. These theories become part of the PVS prelude theories; therefore, be careful not to introduce inconsistencies.
- A file *my-strat* containing the new strategies.
- A file `pvs-lib.lisp` containing

```
(in-package :pvs)
;; If your development requires other prelude libraries, then
;; uncomment the following line and modify it as appropriate.
;; (load-prelude-library "OtherPackage")
(libload "my-strat")
```

- A file `pvs-lib.el` containing Emacs Lisp code that is part of your development.

Once you have put all these files together, instruct the users of your prelude extension to

1. Set the variable `PVS_LIBRARY_PATH` to point to *<PATH>*, where

   <PATH>/MyPackage

   is the actual location of your package.
2. Invoke the Emacs command `M-x load-prelude-library` *MyPackage* the first time *MyPackage* is going to be used in a context. Next time that PVS is restarted in the same context, the prelude extension will be automatically reloaded in the environment.

3 Some useful techniques for strategy writing

This section describes a set of techniques that can be used by a strategy developer to create sophisticated PVS strategies. These techniques include:

1. Incorporating backtracking with TRY.
2. Controlling standard PVS steps with appropriate arguments.
3. Observing the proof state.
4. Probing the CLOS structure of the proof state.
5. Defining helper functions in Lisp.
6. Carefully using global variables.
7. Computing a command in Lisp, and then invoking it.
8. Using auxiliary lemmas for rewriting and forward chaining.
9. Using labels and comments.
10. Using functions from PVS.
11. Applying wrappers.
12. Naming subexpressions of complex expressions.
13. Using templates.
14. Comparing proof step definitions using PVS's multiple proof feature.

The TAME [1] strategies and the strategy packages Manip [5] and Field [6] all employ many or all of these techniques. Below, we illustrate how each individual technique can be used to advantage.

3.1 Using TRY for backtracking.

Backtracking is a powerful technique for automatic proof search. It enables the restoring of an original proof state after an unsuccessful proof attempt. In PVS, backtracking is achieved by a careful crafting of TRY, FAIL, and atomic proof rules.

The TRY command in PVS combines a conditional and a backtracking control structure. As a conditional control structure, TRY performs an action based on the progress made by a proof command on the current proof state. For instance, the strategy expression

```
(TRY (THEN (LIFT-IF) (PROP) (ASSERT))
     (COMMENT "Progressing ...")
     (SKIP))
```

applies the proof command (THEN (LIFT-IF) (PROP) (ASSERT)). If it does something, i.e., it modifies the current proof state, the comment "Progressing ..." is added to the new proof state. Otherwise, the strategy expression performs the proof command (SKIP) and does nothing else.

On the other hand, the third argument of TRY is a backtracking alternative to failures signaled in its first argument. Failures in TRY's second and third arguments are propagated out of the command. The following semantics, based on an informal set of rules provided by N. Shankar, exposes some technicalities of the behavior of TRY.

We assume that any proof command evaluates to one of the following states:

- *skip*: If the proof states remains unchanged.
- *failure*: If a failure is signaled.
- *success*: If the current goal is discharged.
- *subgoals*: If new subgoals are generated.
- *backtracking*: If backtracking is required.

The evaluation of SKIP, FAIL, and TRY is given by the function $|.|$ as follows

- $|$(SKIP)$| =$ *skip*.
- $|$(FAIL)$| =$ *failure*.

– $|\texttt{(TRY A B C)}| = \begin{cases} |\texttt{C}| & \text{if } |\texttt{A}| \in \{\textit{skip}, \textit{backtracking}\} \\ |\texttt{A}| & \text{if } |\texttt{A}| \in \{\textit{failure}, \textit{success}\} \\ \textit{backtracking} & \text{if } |\texttt{A}| = \textit{subgoals}, |\texttt{B}| \in \{\textit{failure}, \textit{backtracking}\} \\ \textit{subgoals} & \text{if } |\texttt{A}| = \textit{subgoals}, |\texttt{B}| \in \{\textit{skip}, \textit{subgoals}\} \\ \textit{success} & \text{if } |\texttt{A}| = \textit{subgoals}, |\texttt{B}| = \textit{success} \end{cases}$

To complete the description of TRY's behavior, it is necessary to consider that

– The states *failure* and *backtracking* do not propagate out of atomic proof rules, i.e., if the strategy expression of the atomic proof rule S evaluates to either *failure* or *backtracking*, then |S| = *skip*.
– At the top-level, the state *failure* forces the theorem prover to exit, while the state *backtracking* evaluates to *skip*.

For instance,

– |(TRY (SKIP) (ASSERT) (FAIL))| = *failure*.
– |(TRY (TRY (FAIL) A B) C D)| = *failure*.
– |(TRY (TRY A (FAIL) B) C D)| = |D|, if |A| = *subgoals*.
– |(TRY A (TRY B (FAIL) C) D)| = *backtracking*, if |A| = |B| = *subgoals*.

The strategy expression

```
(TRY (TRY (THEN (LIFT-IF) (PROP) (ASSERT)) (FAIL) (SKIP))
     step1
     step2)
```

applies the proof command (THEN (LIFT-IF) (PROP) (ASSERT)). If that command discharges the current goal, then it does nothing else. Otherwise, it backtracks to the original proof state and attempts a new proof with the command $\texttt{step}_2$. Since FAIL does not propagate out of atomic proof rules, i.e., it evaluates to *skip*, the logical behavior of the above strategy expression is equivalent to that of the strategy expression (APPLY (THEN (LIFT-IF) (PROP) (ASSERT) (FAIL))) when $\texttt{step}_2$ = (SKIP).

The TRY command is not symmetric: failures signaled in its second argument is not handled in the same way as failures signaled in its third argument. This makes the analysis of failure propagation difficult and error prone. In particular, some PVS commands, such as THEN, ELSE, REPEAT, SPREAD, etc., are implemented with TRY, and their behavior with respect to failure propagation and backtracking is not easy to characterize. For instance, |(THEN $\texttt{step}_1$ ... $\texttt{step}_n$ (FAIL))| is

– *failure*, if $n = 0$ or $|\texttt{step}_i| = \textit{skip}$ for $1 \le i \le n$.
– *backtracking*, otherwise.

In general, it is a good practice to wrap as atomic proof rules the strategy expressions that can generate failures.

For the interested reader, the experimental package Practicals, available at http://research.nianet.org/fm-at-nia/Practicals, provides a redesigned set of strategicals for catching and signaling failures, as well as additional control structures for programming PVS strategies.

3.2 Controlling standard PVS steps.

When one needs finer control in a strategy, one sometimes needs to use variants of the standard PVS steps that do either less or more than the default actions of these steps. For example, the PVS command

(EXPAND *name*)

does not simply expand the definition of *name*, but performs some simplifications as well. This can be inconvenient; e.g., since one of these simplifications can be a (LIFT-IF), it is possible for a quantified formula involving an IF-THEN-ELSE to become an IF-THEN-ELSE with two quantified formulae as branches, complicating a strategy involving skolemization or instantiation. To obtain the effect of simply expanding the definition of *name*, one should instead use the PVS command

```
(EXPAND name :ASSERT? NONE).
```

Other example PVS steps that can be made to do less for finer control are SPLIT and FLATTEN. Using the optional :depth argument, SPLIT can be prevented from producing more subgoals than one desires. One application of this technique is in the definition of the simple strategy modus-ponens:

```
(defstep modus-ponens (formnum)
   (spread (split formnum :depth 1) ((skip)(skip)))
   "Replaces antecedent formulae A and A => B by A and B when
    the formula A => B is labeled by formnum"
   "Performing Modus Ponens")
```

Note that while the PVS rule ASSERT can sometimes be used to discharge the hypothesis of an implication, ASSERT may cause further changes, and it does not discharge a hypothesis that is not a simple expression. The rule modus-ponens permits one to discharge the hypothesis of an implication, without doing more (or less).

Because controlling the number of subgoals in a strategy can be important, being able to apply fine control to SPLIT is useful. However, one can also apply fine control to FLATTEN as well. This is done by replacing it with FLATTEN-DISJUNCT with an appropriate :depth argument.

One case in which the default action of a PVS step may be too limited is in a context where there is extensive use of CASES expressions. The default of ASSERT and SIMPLIFY is to not simplify inside these expressions. This choice often results in more efficient proofs, but experience has shown this may not be true when proofs involve large, complex, and possibly many-layered CASES expressions. In such a case, one may wish to use (ASSERT :CASES-REWRITE? T) and (SIMPLIFY :CASES-REWRITE? T) instead.

3.3 Observing the proof state.

The PVS proof state and related data structures are represented as classes in the Common Lisp Object System (CLOS). In particular, during the execution of any proof in PVS:

- The current proof state is in the global variable *ps*.
- The current proof goal is in the global variable *goal*. It can be also accessed as (current-goal *ps*).
- The list of current sequent formulae, each one an instances of the CLOS class s-formula, can be accessed as (s-forms (current-goal *ps*)).

A more comprehensive list of PVS global variables and data structures and the information they contain can be found in [10, 11].

The proof state (and in fact the value of any Lisp expression) can be observed during a proof using the proof command LISP. Thus, to observe the sequent formulae of the current goal at some point in the proof, one can issue

```
(LISP (s-forms (current-goal *ps*)))                    (5)
```

at the top-level. When making extensive observations about the proof state, it can become inconvenient to have to embed all the Lisp expressions to be evaluated in a LISP command. Another inconvenience of this command is that it interleaves the desired information with repetitions of the current proof goal, making it difficult to make a coherent sequence of observations. (This applies only to PVS versions earlier than 3.1.) An alternative is to send Lisp into a break; this can be done by typing (LISP (BREAK)).

Each s-formula in (s-forms (current-goal *ps*)) corresponds to one of the labeled formulae in the sequent of the current goal. An example of how a list of sequent formulae appears when displayed is:

```
(NOT A B C NOT D E)                    (6)
```

where `A`, `B`, `C`, `D`, and `E` represent particular PVS formulae. The actual members of the list (6) print out as `NOT A, B, C, NOT D, E`. The list (6) represents the sequent:

```
[-1]  A
[-2]  D
|-------                                        (7)
[1]   B
[2]   C
[3]   E
```

(or a variant in which some square brackets are replace by curly braces). In particular, the negative formulae, in order, correspond to the sequent formulae numbered `-1`, `-2`, and so on, while the positive formulae, in order, correspond to the sequent formulae numbered `1`, `2`, and so on. In general, the list of antecedent (negative) formulae and consequent (positive) formulae can be extracted from the proof state as `(n-sforms (current-goal *ps*))` and `(p-sforms (current-goal *ps*))`, respectively.

Note that formulae in the antecedent, such as `A` and `D` in the sequent (7), appear negated in the representation of the PVS proof state. The following Lisp code retrieves a formula in positive form, i.e., as it appears to the user in the PVS theorem prover, from the formula number:[5]

```
; Get formula from current goal (unnegated if antecedent formula)
; Assumes that fnum is a formula number
(defun get-fnum (fnum)
  (let ((index (- (abs fnum) 1))
        (goal (current-goal *ps*)))
    (if (> fnum 0)
        (formula (nth index (p-sforms goal)))
        (argument (formula (nth index (n-sforms goal)))))))
```

To determine that one needs `argument` and `formula` to extract the desired part of an `s-formula` in `(p-sforms goal)` and `(n-sforms goal)`, one can use technique 4 described in Section 3.4.

The inverse of the operation `get-fnum` is to find the formula number or numbers corresponding to formulae with a given property. The PVS Lisp function `(gather-fnums` *`s-forms yes-fnums no-fnums pred`*`)`, described in [10,11], returns the list of formula numbers (taken from *yes-fnums*/*no-fnums*) of sequent formulae in *s-forms* that satisfy *pred*. For example, given the property

```
(defun is-forall (sform) (forall-expr? (formula sform)))
```

the Lisp code:

```
(gather-fnums (s-form *goals*) '* nil #'is-forall)          (8)
```

retrieves all the formula numbers in the current sequent that are universally quantified.

3.4 Using CLOS probes.

Most values manipulated by PVS proof steps are CLOS objects. For instance, `*ps*` is a CLOS object which has a component `current-goal`; in turn, `(current-goal *ps*)` is a CLOS object which has a component `(s-forms (current-goal *ps*))`. To probe the CLOS structure of an object and its components, one can use the Lisp functions `describe` or `show`. Given an object *object*, one can probe its CLOS representation in depth by repeatedly using `describe` to discover components to be probed further:

(describe *object*)

(describe (*component object*))

(describe (*component* (*component object*)))

. . .

[5] More involved versions of this function that take care of special symbols, labels, and error handling are available in the Manip (http://shemesh.larc.nasa.gov/people/bld/manip.html) and Field (http://research.nianet.org/~munoz/Field) packages.

The function `describe` provides explicit names of the component slots in the representations of objects, and these names can then be used like function names to retrieve the elements in these slots, which are themselves objects. The description of *object* starts with a sentence of the form:

object `is an instance of #<STANDARD-CLASS` *object-class*`>`

This information generally tells you that *object-class?* is a recognizer for objects of class *object-class*. An element `x` of class *object-class* can also be recognized by the fact that `(typep x` *object-class*`)` will be true.

When one needs a shortcut to a sequence of CLOS probes, or when one cannot be sure of the sequence or sequences needed, one can use the function `mapobject`. The function `mapobject` provides an analog for objects of `mapcar` for lists: it traverses (most of) the object structure, applying a given function to each component. Thus, to determine whether an s-formula `sform` contains a universal or existential quantifier, one can use the predicate `has-quantifier`, defined as:

```
(defun has-quantifier (sform)
  (let ((has-quant nil))
       (mapobject #'(lambda (x) (if has-quant t
                                    (when (or (forall-expr? x)
                                              (exists-expr? x))
                                          (setq has-quant t) t)))
                  sform)
  has-quant))
```

3.5 Defining helper functions.

Helper functions from Lisp are useful for writing strategy expressions that involve Lisp code, i.e., those using either `LET` or `IF`. They generally involve CLOS probes into the current proof state; thus, we have already seen the following examples of potential helper functions in Sections 3.3 and 3.4:

- `get-fnum`
- `is-forall`
- `has-quantifier`

The helper function `get-fnum` is used in a `LET` in the strategy `add-eq` in Figure 12 below in Section 4.1. Examples of definition and use of additional helper functions can be found below in Section 4.2.

One can classify Lisp helper functions into general purpose and special purpose functions. General purpose helper functions include functions such as `get-fnum` and `is-forall`, which can be applied, respectively, to any valid formula number (or label) and to any valid s-formula. An example of a special purpose helper function is the function `get_sk_constructor_exprs` from Figure 18 in Section 4.2. The function `get_sk_constructor_exprs` will cause a Lisp break if it is called incorrectly; it must be called only on s-formulae of a very limited form. Special purpose helper functions generally use CLOS probes that are either unusual or grouped in a long series, making them hard to match. Thus, extra care must be taken when these functions are used: they should either be used in a context where they are known to be valid (as in the example in Section 4.2, or else a strategy should test the classes of a CLOS structure and its substructures before applying them.

Alternatively, helper functions can take advantage of Common Lisp's exception handling features to deal with errors. While the language specification [12] explains these features in full detail, the following idiom based on the `handler-case` macro is sufficient for most applications:

```
(handler-case
   <expression>
  (error (condition) <alt value/action>))
```

If the evaluation of `<expression>` proceeds normally, its value is returned as the value of the `handler-case` construct. If the evaluation of `<expression>` raises any type of Lisp error, it will be caught and the `<alt value/action>` will be returned/performed.

3.6 Using global variables.

As in any type of programming, global variables must be used carefully in PVS. Clearly, two rules should be followed:

1. Choose variable names not already in use;
2. *Never* change a predefined PVS global variable, such as `*ps*` or `*goal*`.

Towards satisfying rule 1, one can easily test whether a variable `x` is currently in use: either type the command `(LISP x)` when the prover is running, or else type `x` into the `*pvs*` buffer when the prover is not running. For run-time use, the Lisp functions `boundp` and `fboundp` are available to test whether a symbol is currently bound as a variable or a function. Note that if one violates rule 2 by changing `*ps*`, even if the new value of `*ps*` is a valid proof state object, one is creating a nonconservative extension of PVS, and losing PVS's soundness guarantees.

In general, global variables should be avoided. However, they can be useful as switches. In TAME, for example, the user can control whether saved proofs will be in verbose form (recording specific facts introduced in the proofs), or in bare-bones, nonverbose form, by invoking the rules `(VERBOSE)` and `(NONVERBOSE)`. These rules work simply by setting a specific global variable to `t` or `nil`.

3.7 Computing the command to be invoked.

When a strategy definition has parameters, it can happen that the proof step the strategy is to implement depends on some information that must be computed from the parameter values.

A typical example is when the strategy definition has an `&rest` parameter. When the strategy (or corresponding defined rule) is applied, the `&rest` parameter is bound to a list of actual parameters. The strategy will typically need to extract the `car` and `cdr` of this list as it proceeds. Because proof rules cannot be applied directly to `car` or `cdr` expressions, commands involving the application of proof rules to the `car` or `cdr` of a list of actual parameters must be first computed and then called. Examples where this technique is used are in the definitions of the strategies `apply-lemma`, `else*`, and `rewrite-one` in Figures 7, 8, and 9, respectively, in Section 4.1. (Note that `apply-lemma` computes two commands, `lemma-step` and `inst-step`, though actually, only `inst-step`, which depends on the `&rest` parameter, needs to be computed.)

Another example in Section 4.1 in which commands are computed is in the strategy `add-eq` in Figure 12. Here, two commands `case-step` and `steplist` are computed. Because `case-step` applies `CASE` to values computed from its formula-number arguments, it *must* be computed. Here again, one of the steps, `steplist`, need not be computed. However, note that "unnecessary" computation of a step often adds to the readability of a strategy definition, particularly when companion steps must be computed.

3.8 Rewriting and forward chaining with lemmas.

PVS provides a variety of steps for controlling the use of rewrites. An example of a strategy that takes advantage of PVS's `REWRITE` rule is `rewrite-one` in Figure 9 on page 33. The strategy `rewrite-one` does rewriting once using its lemma arguments as the rewrite rules.

For automatic or "large step" strategies, it is useful to do auto-rewriting. Auto-rewriting on a set of lemmas can be initiated by calling `AUTO-REWRITE` on a list of the lemmas. Similarly, auto-rewriting on a set of lemmas can be terminated by calling `STOP-REWRITE` on a list of the lemmas. Rather than explicitly listing lemmas, it can be convenient to collect a set of rewrites into a theory, and calling `AUTO-REWRITE-THEORY` (and `STOP-REWRITE-THEORY`) on that theory. Any lemmas installed as auto-rewrites will be used as rewrites whenever `DO-REWRITE` is called. Since `ASSERT` and `SIMPLIFY` call `DO-REWRITE`, these two PVS strategies also cause auto-rewrites to be performed. Auto-rewrites must clearly be used carefully, to avoid possible nontermination of rewriting.

Rewrites in PVS can be *conditional* rewrites, where a rewrite rule is applied only if its condition simplifies to `TRUE`. Lemmas with conditions (i.e., hypotheses) can also be used for *forward chaining*, in which the (possibly parameterized) hypothesis is matched to some formula or formulae in the current sequent. Any match defines an instance of the conclusion, that is then added as an antecedent formula to the current sequent. The PVS rule `FORWARD-CHAIN` allows forward chaining on a lemma (or on a formula in the current sequent). Note that using `REPEAT` or `REPEAT*` in combination with `FORWARD-CHAIN` can lead to nontermination if the conclusion of the lemma used for forward chaining matches its hypothesis; therefore, care must also be taken in using repeated forward chaining. There is currently no FORWARD-CHAIN-THEORY, although one is expected to be available in the near future [9].

There are many uses for rewriting and forward chaining; for example, TAME uses both auto-rewriting and forward chaining to automate certain reasoning about the relationships between constructor and accessor functions in DATATYPEs that is not handled by ASSERT or GRIND.

3.9 Using labels and comments.

A simple use of comments and labels in a strategy has already been illustrated in Figure 1, which shows the definition of the strategy suppose. This strategy uses WITH-LABELS to introduce a set of labels simultaneously, and the command COMMENT is for introducing comments. There is also a command LABEL for introducing a single label.

Labels are applied to formulae. Once a formula has a label, it can be referred to by that label. This fact has many uses in strategies. For example, a labeled formula can be hidden and revealed by calling HIDE and REVEAL on its label. One use of this device is to prevent expansion of definitions in the labeled formula except when such expansion is desired. Another example use for labels is to coordinate skolemization of one quantified formula with instantiation of another. It is possible to give a formula multiple labels by using the optional argument :push? T with either WITH-LABELS or LABEL. This allows all information in original labels to be retained, while adding new information, so that formulae can, if desired, be included in multiple categories for multiple purposes. The use of labels can also increase the stability of strategies. For simplicity, several example strategies in this tutorial use explicit references to formula numbers (see Sections 3.10 and 4). However, provided one knows the number, ordering, and nature of the new formulae that will be created by a command, by wrapping that command using WITH-LABELS and an appropriate list of labels, one can avoid explicit formula number references. On the assumption that the ordering in the set of newly created formula is less likely to change in new PVS versions than the explicit formula numbers that will be assigned to the new formulae, user strategies using WITH-LABELS and label references will be less fragile than those using explicit formula number references. An example of how labels appear in a sequent is shown in Figure 2, which shows a subgoal from a TAME proof for the invariant lemma lemma_5 of *TIP* [4, 3].

```
lemma_5.1 :
;;;Case add_child(addE_action)

{-1,(pre-state-reachable)}
      reachable(prestate)
{-2,(inductive-hypothesis)}
      length(mq(basic(prestate))(e_theorem)) <= 1
{-3,(general-precondition)}
      enabled_general(add_child(addE_action), prestate)
{-4,(specific-precondition)}
      enabled_specific(add_child(addE_action), prestate)
{-5,(post-state-reachable)}
      reachable(poststate)
  |-------
{1,(inductive-conclusion)}
      IF NOT (mq(basic(prestate))(addE_action) = null)
      THEN length(mq(basic(prestate)) WITH
                  [(addE_action) :=
                   cdr(mq(basic(prestate))(addE_action))]
                  (e_theorem))
      ELSE length(mq(basic(prestate)(e_theorem)))
      ENDIF
       <= 1
```

Fig. 2. An example TAME sequent illustrating labels.

In contrast to labels, which attach to formulae, comments attach to subgoals. Note that subgoal in Figure 2 also contains a comment which identifies the case to which the subgoal corresponds. Comments also appear in saved proofs, immediately after the command that introduces them. When a command creates branches, it is possible to "label" the branches in the saved proof with comments by wrapping the command creating the branches in a SPREAD or BRANCH construct that then applies multiple calls to COMMENT to the branches, as illustrated in Figure 1 on page 19. An example saved proof showing how comments can be used to make saved proofs more understandable is shown in Figure 3, which shows the saved TAME proof of the the *TIP* property lemma_5. The subgoal in Figure 2 is the first subgoal of the first branch of the proof in Figure 3, so the comment in this subgoal "labels" the first branch of the proof. The saved proof in Figure 3 illustrates the effect of suppose, and also shows that comments can be used to capture ephemeral information from proof goals, such as facts being used in the reasoning.

```
Inv_5(s:states): bool = (FORALL (e:Edges): length(mq(e,s)) <= 1);
;;; Proof lemma_5-like-hand for formula tip_invariants.lemma_5
(""
 (AUTO_INDUCT)
 (("1" ;;Case add_child(addE_action)
   (APPLY_SPECIFIC_PRECOND)
   ;;Applying the precondition
   ;;init(target(addE_action), prestate)
   ;; & NOT (mq(addE_action, prestate)=null)
   (SUPPOSE "e_theorem = addE_action")
   (("1" ;;Suppose e_theorem = addE_action
     (TRY_SIMP))
    ("2" ;;Suppose not [e_theorem = addE_action]
     (TRY_SIMP))))
  ("2" ;;Case children_known(childV_action)
   (SUPPOSE "source(e_theorem) = childV_action")
   (("1" ;;Suppose source(e_theorem) = childV_action
     (APPLY_SPECIFIC_PRECOND)
     ;;Applying the precondition
     ;;init(childV_action, prestate)
     ;;    &
     ;;  (FORALL (e: Edges):
     ;;     FORALL (f: tov(childV_action)):
     ;;       child(e, prestate) OR child(f, prestate) OR e = f)
     (APPLY_INV_LEMMA "2" "e_theorem")
     ;;Applying the lemma
     ;;(FORALL (e: Edges): init(source(e), prestate)
     ;; => mq(e, prestate)=null)
     (TRY_SIMP))
    ("2" ;;Suppose not [source(e_theorem) = childV_action]
     (TRY_SIMP))))
  ("3" ;;Case ack(ackE_action)
   (SUPPOSE "e_theorem = ackE_action")
   (("1" ;;Suppose e_theorem = ackE_action
     (APPLY_SPECIFIC_PRECOND)
     ;;Applying the precondition
     ;;NOT (init(target(ackE_action), prestate))
     ;;    & NOT (mq(ackE_action, prestate) = null)
     (TRY_SIMP))
    ("2" ;;Suppose not [e_theorem = ackE_action]
     (TRY_SIMP))))))
```

Fig. 3. A verbose TAME proof illustrating comments in a saved proof.

3.10 Using Lisp functions from PVS.

As illustrated in Section 3, one can use PVS Lisp functions documented in [10, 11] in writing Lisp code to be used in strategies.[6] These documented functions can be a convenience in writing Lisp code, but one can generally achieve the same effects in one's Lisp code by combining standard Lisp constructs with CLOS probes. For example, the effect of the code in 8 on page 24, which solves the problem of listing all formula numbers in a goal corresponding to quantified formulae, can also be achieved by the code

```
(gather-fnums-property 'is-forall (current-goal *ps*))          (9)
```

where `gather-fnums-property` is defined by:

```
(defun gather-fnums-property (prop goal)
   (let ((negfnums
          (let ((fnum 0))
          (loop for x in (n-sforms goal) do (setq fnum (- fnum 1))
               when (funcall prop x) collect fnum)))
         (posfnums
          (let ((fnum 0))
          (loop for x in (p-sforms goal) do (setq fnum (+ fnum 1))
               when (funcall prop x) collect fnum))))
     (append negfnums posfnums)))
```

However, there are PVS Lisp functions that are not formally documented that allow one to solve problems in ways not so easily duplicated.

Consider the following problem. PVS expressions that are parameters to proof commands are input as strings. In general, these expressions are built from other expressions in the proof state, where they appear as CLOS structures, and converted to strings with the Lisp function `format`. In some special cases, we may want to perform the inverse operation, i.e., to get a CLOS structure from the string representation of a PVS expression. A simple way to achieve this operation is to bring the PVS expression to the proof state, for example using a harmless `(CASE "`*expr* `=` *expr*`")`, and then observing the CLOS structure of the proof state as explained in Sections 3.3 and 3.4. The following piece of code implements this technique

```
(LET ((casestr (format nil "(~A) = (~A)" expr expr)))
    (THEN
     (CASE casestr)
     (LET ((closexpr (args1 (get-fnum -1))))
      (THEN
       (DELETE -1)
        ;; closexpr is the CLOS representation of expr
       (... closexpr ...)))))
```

The code above (which makes use of the documented PVS Lisp function `args1`) has the side effect of temporarily modifying the proof state. In most cases, the modification has no logical consequences. However, if `expr` generates TCCs, these TCCs will appear in the new proof state.

An alternative, cleaner way to get a CLOS structure of a PVS expression is by using the PVS parser and type-checker functions `pc-parse` and `pc-typecheck` directly. These functions are not properly documented and they must be used with care; otherwise, the PVS prover could get into an unstable state. The function `(pc-parse` ***expr gramtyp***`)` returns a non-type-checked CLOS structure of the expression ***expr***. The parameter ***gramtyp*** is a grammar nonterminal, ***in most cases*** with the same name as the CLOS type of the structure to be parsed. For instance,

```
(pc-parse "(# x:=1, b:=true #)" 'expr)
```

[6] An API document that covers all the Lisp calls needed for strategies and integration with other tools is being written at SRI [7].

returns the CLOS structure corresponding to the PVS record `(# x:=1, b:= true #)`. On the other hand,

```
(pc-parse "[# x:int, b:bool #]" 'type-expr)
```

returns the CLOS structure corresponding to the PVS type record `[# x:int, b:bool #]`. CLOS structures should not be used in a proof state unless they are appropriately type-checked. The function `(pc-typecheck` *`closexpr`*`)` adds PVS type information to the CLOS structure *`closexpr`*. Usually, a call to `pc-parse` is followed by a call to `pc-typecheck`.

An example where converting a string to a CLOS structure in this fashion is useful is in defining a strategy whose behavior depends on the type of one or more of its arguments. Provided the string `x` names a valid expression that is type correct in the current proof goal, the value of

```
(type (pc-typecheck (pc-parse x 'expr)))                    (10)
```

will be the (CLOS representation of the) type of that expression. (Note that `type` is a CLOS probe—i.e., the name of a slot or method—rather than a function from PVS.) The string

```
(princ-to-string (type (pc-typecheck (pc-parse x 'expr))))
```

can then be compared to any specific type name represented as a string, or, more safely, the (not yet documented) PVS Lisp function `tc-eq` can be used to compare the type (10) with another (analogously computed) type.

3.11 Applying wrappers.

Wrappers are strategicals that prevent their strategy arguments from causing unintended effects. We have already seen one example use for wrapping: wrapping a command that may lead to failure in `(APPLY ...)` so that any failure caused will be local (undoing the proof only to the subgoal where the command was applied).

Another instance in which one may wish to use a wrapper is when a strategy has potential side effects, for example through the use of auto-rewrites or global variables, and one wishes to be sure no permanent side effects result from execution of the strategy. Even a strategy that ultimately follows every auto-rewrite command with an appropriate corresponding stop-rewrite command can leave "dangling rewrites" active if it produces multiple branches and proves the last branch before it reaches a needed stop-rewrite command. In such a case one can wrap the strategy, together with a "cleanup step" that removes any potential side effects, in the strategical `unwind_protect` defined in Figure 4. To protect against auto-rewrites remaining

```
(defstep unwind-protect (main-step cleanup-step)
  (spread (case "id(true)")
          ((then (delete -1) main-step)
           (then cleanup-step (expand "id" 1))))
  "Invoke MAIN-STEP followed by CLEANUP-STEP, which is performed
even if MAIN-STEP leads to a proof of the current goal."
  "Invoking proof step with cleanup")
```

Fig. 4. An example "safety wrapper" strategical.

unintentionally active, the `cleanup-step` argument to `unwind-protect` can be a strategy that performs the needed sequence of stop-rewrite commands.

3.12 Naming a subexpression.

Field axioms, such as associativity, commutativity, distributivity, etc., are known to the PVS decisions procedures. For instance, the sequent

```
  |-------
{1}  x * x >= 0
```

is automatically discharged by the proof command (GRIND). Surprisingly, the sequent

```
  |-------
{1}  (x - 1) * (x - 1) >= 0
```

is not discharged by (GRIND). In this case, GRIND yields the sequent:

```
  |-------
{1}   1 - x + (x * x - x) >= 0
```

which is not further simplified by the PVS decision procedures.

The reason for this behavior is that the decision procedures always apply fields axioms, and in particular the distributive law, before other simplifications. Since PVS does not provide an explicit mechanism to customize these simplifications, they can be problematic for writing strategies where proof control is fundamental.

One way to avoid certain implicit simplifications, such as the distributive law, is to wrap a subexpression in an application of the identity function, e.g., id(x - 1). This renders the expression ineligible for the distributive law. When this protection is no longer desired, the id function can be expanded to restore the original expression. For simple cases this technique is often adequate.

For more advanced uses, undesired simplification can be avoided by *naming* the expression that should not be simplified. This can be achieved with the commands NAME and REPLACE, or the command NAME-REPLACE. The commands NAME introduce a new name definition to the current sequent. This name is then used by REPLACE to abbreviate the original expression.

Figure 5 illustrates a strategy that blocks the first application of the distributive law in a formula by introducing a new name. The strategy NODISTR uses helper functions get-fnum, get-newname, get-distr-expr, and get-distr-plus. The function get-fnum (see Section 3.3) gets the formula in the formula number fnum. New names are created by the function get-newname, which increments the global variable newname each time a new name is required. Finally, the functions get-distr-expr and get-distr-plus descend the formula tree to find the first expression having the form $(x + y) * z$ or $z * (x + y)$. These functions use PVS functions infix-application? that checks if a formula is an infix application, name-expr? that checks if an operator is a name (as opposed to a lambda expression), and args1 and args2 that projects the first and second argument of an application, respectively.

For instance, (NODISTR 1) applied to the sequent

```
  |-------
{1}  (x - 1) * (x - 1) >= 0
```

yields the sequent[7]

```
{-1}  (x - 1) = v7__
  |-------
{1}   v7__ * v7__ >= 0
```

When strategies introduce new names automatically, there is the possibility of conflicts with user supplied names. To prevent such clashes, we recommend following a naming convention that yields distinctive identifiers. For example, the convention followed by the function get-newname is to create identifiers with two trailing underscore characters.

The strategy NODISTR can be used to improve the automation provided by GRIND on the field of real numbers. For example, the simple strategy GRINOD in Figure 6 discharges, among others, the following sequent

```
  |-------
{1}   FORALL (x: real): (x - 1) * (x - 2) * (x - 1) * (x - 2) >= 0
```

[7] The name of the new variable may be different.

```
;; Strategy definition
(defstrat NODISTR (fnum)
  (LET ((form (get-fnum fnum))
        (name (get-newname))
        (expr (get-distr-expr form))
        (str  (when expr (format nil "~A" expr))))
    (IF str (NAME-REPLACE name str :hide? nil) (SKIP)))
  "Introduces a new name in ~A to block the distributive law")

;; Generating new names
(setq newname 0)

(defun get-newname ()
  (progn (setq newname (+ 1 newname))
  (format nil "v~A__" newname)))

;; Helper functions
(defun get-distr-expr (form)
  (when (and (infix-application? form)
             (name-expr? (operator form)))
    (let ((op (id (operator form))))
      (cond ((member op '(= <= >= < > + - /))
             (or (get-distr-expr (args1 form))
                 (get-distr-expr (args2 form))))
            ((eq op '*)
             (or (get-distr-plus (args1 form))
                 (get-distr-plus (args2 form))))
            (t nil)))))

(defun get-distr-plus (form)
  (when (and (infix-application? form)
             (name-expr? (operator form)))
    (let ((op (id (operator form))))
      (cond ((member op '(+ -)) form)
            ((member op '(* /))
             (or (get-distr-expr (args1 form))
                 (get-distr-expr (args2 form))))
            (t nil)))))
```

Fig. 5. Naming a subexpression to block the distributive law

```
(defstrat GRINOD (fnum)
  (THEN (SKOSIMP fnum)
        (REPEAT (NODISTR fnum))
        (GRIND :theories "real_props"))
  "Blocks the distributive law in ~A before applying GRIND")
```

Fig. 6. Combining `NODISTR` and `GRIND`

3.13 Using templates.

The use of templates is an indirect technique that can be used in strategy development. For example, when one is reasoning in a special domain, one may wish to assume some degree of uniformity either in the objects about which one is reasoning or in the formulations of properties of these objects (or both). Templates allow one to enforce a standard naming scheme for objects and their types or a standard scheme for expressing properties. As a result, strategies based on templates can be based on a certain amount of definite information that allows them to make more reasoning automatic, and thus to achieve larger size proof steps.

Templates for both specifications and lemmas are used to advantage by TAME.

3.14 Using PVS's multiple proof feature.

For proof steps that do a significant amount of automatic reasoning, and which therefore can take a long time to execute, efficiency is an important design goal. Once one has designed a strategy that achieves an intended purpose, one can compare the strategy for efficiency against alternate versions by saving proofs that use the different versions. The saved proofs include run time information that can be used for efficiency comparison.

Comparisons for efficiency should be done over several examples, as there are often tradeoffs in the choice between two near-optimal versions of a strategy. Note that the PVS command `TIME`, which is similar to `APPLY` in that it turns a strategy into an atomic rule, has the additional effect of giving timing information for any branches created by the strategy on which the strategy does not terminate. Thus, `TIME` provides an additional resource in studies of efficiency: it can be used for strategy efficiency comparisons between the cases in the branches a strategy generates.

4 Examples of strategy design

In this section, we provide several examples to further illustrate the kinds of reasoning steps that can be supported with PVS strategies, and to provide new PVS strategy developers with some useful ideas that they may wish to recycle in their own strategies.

4.1 Some small-step strategy examples

The example strategies in this section are geared towards carrying out tasks during interactive proving, and can be viewed as providing slightly more powerful versions of existing prover rules. Included are examples of:

- capturing a commonly used pattern of steps within a single step,
- using `TRY` together with recursion to define a step that iterates a command over the list of arguments to the step,
- forking a "proof obligation" proof to simplify introducing a fact (as a conjecture) on the current proof branch, and
- creating a new arithmetic reasoning step that is not supported by any standard PVS proof step.

Several of these examples also illustrate techniques from Section 3, including computing and then executing a command, use of CLOS probes into the proof state, use of Lisp helper functions, and use of PVS functions.

Figure 7 shows a modest strategy `apply-lemma` that invokes a lemma after accepting a list of expressions for instantiating the variables. The strategy expands into a prover command of the form:

```
(THEN (LEMMA name) (INST -1 expr-1 ... expr-n))
```

Note that the bindings of the LET construct in `apply-lemma` could have been written using Lisp's backquote feature:

```
(let ((lemma-step '(lemma ,lemma))
      (inst-step  '(inst -1 ,@exprs)))
  (then lemma-step inst-step))
```

```
(defstep apply-lemma (lemma &rest exprs)
  (let ((lemma-step (list 'lemma lemma))
        (inst-step  (cons 'inst (cons -1 exprs))))
    (then lemma-step inst-step))
  "Apply a lemma with explicit variable instantiations.
Lemma variables appear in alphabetical order when introduced
by the LEMMA rule.  That order needs to be observed when
entering EXPRS."
  "~%Invoking lemma ~A on given expressions")
```

Fig. 7. Applying a lemma and instantiating its variables.

```
(defstep else* (&rest steps)
  (if (null steps)
      (skip)
      (let ((try-step `(try ,(car steps)
                            (skip)
                            (else*$ ,@(cdr steps)))))
        try-step))
  "Try STEPS in sequence until the first one succeeds."
  "~%Trying steps in sequence")
```

Fig. 8. Generalization of the prover's ELSE strategical.

In many cases this type of notation simplifies the coding effort and improves readability. We will make use of it in the remaining examples.

Figure 8 illustrates a basic strategy pattern for trying a series of actions until one succeeds. When the first step is encountered that has an effect on the proof state, the strategy terminates without attempting any of the remaining steps. ELSE* can be thought of as a generalization of the prover's built-in ELSE strategical. It is likely to be useful as a building block for higher level strategies.

The TRY strategical together with recursive invocation is employed to achieve the effect of conditional iteration. For each element of argument STEPS, if trying the step has no effect, ELSE* is invoked again on the remaining steps. TRY is applied to achieve the following general scheme:

```
(TRY current-step (SKIP) recursive-invocation)
```

Note the use of the strategy form (ELSE*$) rather than the rule form (ELSE*) in the recursive invocation. This is a convention often followed in PVS strategies. It ensures that when the top-level command is invoked as a nonatomic strategy, all subordinate strategies will be as well, resulting in a full expansion into predefined rules.

```
(defstep rewrite-one (fnums &rest lemmas)
  (if (null lemmas)
      (skip)
      (let ((try-step
              `(try (rewrite ,(car lemmas) ,fnums)
                    (skip)
                    (rewrite-one$ ,fnums ,@(cdr lemmas)))))
        try-step))
  "Try rewriting LEMMAS in sequence within FNUMS until the
first one succeeds."
  "~%Trying lemma rewrites in sequence")
```

Fig. 9. Using the pattern for ELSE* in a more concrete setting.

Figure 9 demonstrates how the pattern of ELSE* can be applied to a more concrete objective. Given a list of lemma names, REWRITE-ONE tries to rewrite with each lemma in turn until one is successful. It also provides an argument FNUMS to control which part of the sequent should be subject to rewriting. It follows the same recursive pattern presented in Figure 8. (See Section 3 for more on the use of term rewriting in PVS.)

```
(defstep claim-cond (cond)
  (let ((case-step (list 'case cond))
        (steplist
          (list '(skip)
                '(try (then (grind) (fail))
                      (skip)
                      (skip-msg "Claim justification not proved"
                                t)))))
    (spread case-step steplist))
  "Try claiming a condition holds.  A proof of the justification
step is attempted using (GRIND)."
  "~%Claiming the condition ~A holds")
```

Fig. 10. Claiming a condition and trying to prove its justification.

Figure 10 illustrates a different use for the TRY strategical. In CLAIM-COND, we wish to accept a PVS expression COND as a condition that holds in the current goal and introduce it as a new antecedent formula. We would also like to automatically prove that the condition holds.

To carry out this task, we use CASE to introduce the supposition, then apply GRIND on the second branch generated by CASE to prove that COND holds. If GRIND fails to completely prove the justification, we undo the partial proof and leave it to the user to determine how to proceed. This behavior is obtained using the following scheme on the second branch generated by CASE:

```
(TRY (THEN (GRIND) (FAIL)) (SKIP) (SKIP-MSG message t))
```

Backtracking via FAIL is performed if the subgoal is not completely proved. In this case the SKIP-MSG rule is invoked to display a message to the user that the justification proof did not succeed.

To direct the branching of the proof into subgoals, the SPREAD strategical is used. The first argument to SPREAD is a step that causes branching, which is CASE in this instance. The second argument is a list of steps for the follow-up actions to be performed for each subgoal. The second subgoal represents the justification proof for the claim, where the TRY construct is applied.

```
(defstep equate-terms (lhs rhs)
  (let ((case-eq (list 'case
                       (format nil "(~A) = (~A)" lhs rhs)))
        (steplist
          (list '(replace -1 :hide? t)
                '(try (then (grind) (fail))
                      (skip)
                      (skip-msg "Equate justification not proved"
                                t)))))
    (spread case-eq steplist))
  "Try equating two expressions and replacing the LHS by the RHS.
A proof of the justification step is attempted using (GRIND)."
  "~%Equating two expressions and replacing")
```

Fig. 11. Claiming two terms are equal and carrying out replacement.

Figure 11 follows the same pattern as found in Figure 10. EQUATE-TERMS accepts two PVS expressions that are claimed to be equal, then substitutes one for the other. A new antecedent equality LHS = RHS will be added as a claim. REPLACE is applied to substitute RHS for LHS. Then a justification proof to establish the equality is carried out in the same manner as CLAIM-COND.

Forming the CASE command requires some string manipulation, which is implemented using Lisp's FORMAT function. This is an example of a common operation in strategy writing. LET bindings are introduced to allow Lisp code to compute prover command invocations having whatever arguments are necessary.

```
(defstep add-eq (fnum1 fnum2)
  (let ((formula1  (get-fnum fnum1))
        (formula2  (get-fnum fnum2))
        (left-sum  (format nil "~A + ~A"
                           (args1 formula1) (args1 formula2)))
        (right-sum (format nil "~A + ~A"
                           (args2 formula1) (args2 formula2)))
        (case-step '(case ,(format nil "~A = ~A"
                                   left-sum right-sum)))
        (steplist  '((skip) (then (assert) (assert)))))
    (spread case-step steplist))
  "Given two antecedent equalities a = b and c = d, introduce
a new formula relating their sums, a + c = b + d."
  "~%Adding terms from ~A and ~A to derive a new equality")
```

Fig. 12. Adding two antecedent equalities to generate a third.

Figure 12 illustrates the extraction of expressions from CLOS objects within the current proof state. ADD-EQ accepts two formula numbers for antecedent equalities involving numeric values. It then introduces a new antecedent equality that sums the two equations, i.e., given equations $a = b$ and $c = d$, it forms $a + c = b + d$. The justification proof consists of two applications of ASSERT, which should be sufficient to prove the subgoal.

To extract terms from the proof state, the formula objects are first retrieved using the Lisp function GET-FNUM described earlier. Assuming the formulae are equalities, their left hand and right hand sides can be accessed using the PVS functions ARGS1 and ARGS2. When supplied as values to the FORMAT function, Lisp renders their textual representations as PVS expressions. This allows ordinary string manipulation to be used to construct new PVS expressions from fragments of the current sequent.

Having formed the new antecedent equality as a text string, an application of the CASE rule is used to achieve the desired effect. In a more realistic strategy development effort, error checking code would be inserted at various places to check for invalid inputs. Strategy writers can decide how important such checking is for the intended purpose of their strategies.

4.2 Developing high level strategies: an example

Strategies geared to high level proof automation, either of full proofs or of proof steps at a high conceptual level, almost invariably require use of several of the techniques described in Section 3. To illustrate how some of the techniques described in Sections 2 and 3 can be applied to developing an automatic strategy for proving lemmas belonging to a particular class, we will show how the defined rule adt_unique_strat from TAME was developed.[8] Although adt_unique_strat was developed for TAME, it is useful in any context in which the DATATYPE construct is used: it allows the user to supply a one-step proof for any lemma that asserts that

[8] A little history: development of TAME strategies began with an early version of PVS, in which the PVS step DECOMPOSE-EQUALITY was not a standard proof rule. With this rule, one can write a much simpler version of adt_unique_strat. The example in this section nevertheless serves to illustrate a general approach to creating a specialized high level strategy.

if two elements of the same DATATYPE with the same constructor are equal, then the arguments to which the constructor is applied to obtain these elements must be pairwise equal. Figure 13 shows an example DATATYPE and its "uniqueness properties" taken from the TAME specification of the basic TESLA multicast stream authentication protocol [8, 2]. Such a lemma is a corollary of the fact that the elements of any PVS DATATYPE form a free algebra, that is, a term algebra with no nontrivial equalities between terms. Unfortunately, the automatic PVS proof procedures such as ASSERT, SIMPLIFY, and GRIND do not automatically "know" this information. Moreover, as can be seen from the proof of Receive_unique in Figure 14, one does not really

```
actions: DATATYPE
  BEGIN
    nu(timeof:(fintime?)): nu?
    SSend (Si:nat, Sc,Sk1,Sk2:Key, Sm:Message): SSend?
    ASend (Ai:nat, Ac:Commit, Ak1,Ak2:Key, Am:Message): ASend?
    Receive (RSentPacket:SentPacket): Receive?
  END actions

nu_unique: LEMMA FORALL (t1, t2: (fintime?)):
    nu(t1) = nu(t2) => t1 = t2;

Send_unique: LEMMA FORALL (i1,i2:nat, c1,c2,k11,k12,k21,k22: Key,
                            m1,m2:Message):
    SSend(i1,c1,k11,k21,m1) = SSend(i2,c2,k12,k22,m2)
    => i1=i2 & c1=c2 & k11=k12 & k21=k22 & m1=m2;

ASend_unique: LEMMA FORALL(i1,i2:nat, c1,c2:Commit,
                            k11,k12,k21,k22: Key, m1,m2:Message):
    ASend(i1,c1,k11,k21,m1) = ASend(i2,c2,k12,k22,m2)
    => i1=i2 & c1=c2 & k11=k12 & k21=k22 & m1=m2;

Receive_unique: LEMMA FORALL (sp1, sp2: SentPacket):
    Receive(sp1) = Receive(sp2) => sp1 = sp2;
```

Fig. 13. Example of a PVS DATATYPE declaration, and its "uniqueness lemmas".

want to make an excursion in a PVS proof to establish this property.

The first step in developing adt_unique_strat is to prove several uniqueness lemmas in PVS and look for patterns. Figure 14 shows the pattern to follow in establishing a uniqueness lemma for a constructor with one

```
(""
 (SKOLEM!)
 (FLATTEN)
 (CASE "sp1!1 = RSentPacket(Receive(sp1!1))")
 (("1" (CASE "sp2!1 = RSentPacket(Receive(sp2!1))")
   (("1"
     (APPLY (THEN (REPLACE -1 +) (REPLACE -2 +) (HIDE -1 -2)))
     (REPLACE -1)
     (PROPAX))
    ("2" (ASSERT))))
  ("2" (ASSERT))))
```

Fig. 14. Proof of a uniqueness lemma for a DATATYPE constructor with one parameter.

parameter: One can see that, after skolemizing and flattening the formula in the lemma, one does two case splits, each based on an equality of an individual skolem constant to an application of the single datatype accessor function RSentPacket for Receive actions to an application of the Receive constructor to the same

```
(""
 (SKOLEM 1
  ("i_1" "i_2" "c_1" "c_2" "k1_1" "k1_2" "k2_1" "k2_2" "m_1" "m_2"))
 (FLATTEN)
 (SPLIT)
 (("1" (CASE "i_1 = Si(SSend(i_1,c_1,k1_1,k2_1,m_1))")
   (("1" (CASE "i_2 = Si(SSend(i_2,c_2,k1_2,k2_2,m_2))")
     (("1"
       (APPLY (THEN (REPLACE -1 +) (REPLACE -2 +) (HIDE -1 -2)))
       (REPLACE -1)
       (PROPAX))
      ("2" (ASSERT))))
    ("2" (ASSERT))))
  ("2" (CASE "c_1 = Sc(SSend(i_1,c_1,k1_1,k2_1,m_1))")
   (("1" (CASE "c_2 = Sc(SSend(i_2,c_2,k1_2,k2_2,m_2))")
     (("1"
       (APPLY (THEN (REPLACE -1 +) (REPLACE -2 +) (HIDE -1 -2)))
       (REPLACE -1)
       (PROPAX))
      ("2" (ASSERT))))
    ("2" (ASSERT))))
  ("3" (CASE "k1_1 = Sk1(SSend(i_1,c_1,k1_1,k2_1,m_1))")
   (("1" (CASE "k1_2 = Sk1(SSend(i_2,c_2,k1_2,k2_2,m_2))")
     (("1"
       (APPLY (THEN (REPLACE -1 +) (REPLACE -2 +) (HIDE -1 -2)))
       (REPLACE -1)
       (PROPAX))
      ("2" (ASSERT))))
    ("2" (ASSERT))))
  ("4" (CASE "k2_1 = Sk2(SSend(i_1,c_1,k1_1,k2_1,m_1))")
   (("1" (CASE "k2_2 = Sk2(SSend(i_2,c_2,k1_2,k2_2,m_2))")
     (("1"
       (APPLY (THEN (REPLACE -1 +) (REPLACE -2 +) (HIDE -1 -2)))
       (REPLACE -1)
       (PROPAX))
      ("2" (ASSERT))))
    ("2" (ASSERT))))
  ("5" (CASE "m_1 = Sm(SSend(i_1, c_1,k1_1,k2_1,m_1))")
   (("1" (CASE "m_2 = Sm(SSend(i_2,c_2,k1_2,k2_2,m_2))")
     (("1"
       (APPLY (THEN (REPLACE -1 +) (REPLACE -2 +) (HIDE -1 -2)))
       (REPLACE -1)
       (PROPAX))
      ("2" (ASSERT))))
    ("2" (ASSERT))))))
```

Fig. 15. Proof of a uniqueness lemma for a DATATYPE constructor with five parameters.

skolem constant. The technique used in this proof can be adapted to handle the case of a constructor with more arguments. Figure 15 shows a proof of the uniqueness lemma for the constructor SSend:

The proof of this lemma also begins with skolemization and flattening, but this is followed by a SPLIT command. By executing the proof, one can see that the SPLIT splits the proof into subcases, one for each accessor function of SSend, and therefore, calling (SPLIT) at the third step in the shorter proof would have no effect. In each subcase of the longer proof, the pattern in the shorter proof reappears. Moreover, this pattern is now more detailed: the two individual skolem constants correspond to the variables retrieved by the accessor function, and the constructor SSend is applied not just to these skolem constants, but to the two sets of skolem constants corresponding to the variables in the SSend expressions in the hypothesis of the lemma.

We now have enough information to design a strategy. We can begin by defining a Lisp function that returns a command that follows the pattern of the subcases. Figure 16 shows the definition of such a function: mk_adt_unique_case, which takes as arguments the accessor function name, the two skolem constant names, and the two instantiated constructor expressions used in the pattern. We will expect to begin our strategy as the proof in Figure 15 begins: with a skolemization step, a (FLATTEN), and a (SPLIT). Following the

```
(defun mk_adt_unique_case (acc skconst-1 skconst-2
                               sk-expr-1 sk-expr-2)
  (let ((firstcase
          (format nil "~a~a~a~a~a~a"
           skconst-1 " = " acc "(" sk-expr-1 ")"))
        (secondcase
          (format nil "~a~a~a~a~a~a"
           skconst-2 " = " acc "(" sk-expr-2 ")")))
  '(spread (case ,firstcase)
           ((spread (case ,secondcase)
                    ((then (replace -1 +)
                           (replace -2 +)
                           (hide -1 -2)
                           (replace -1))
                     (assert)))
            (assert)))))
```

Fig. 16. A Lisp function that computes a command to prove a uniqueness lemma case.

(SPLIT) command, we then plan to use SPREAD to apply an appropriate subcase command to each of the subgoals.

To apply SPREAD, we we need a list of appropriate subcase commands, so we next define a Lisp function collect_adt_unique_cases that returns such a list, as follows. From the proof of the lemma SSend_unique in Figure 15, we see that there is a uniqueness case for every accessor function. Moreover, the two instantiated constructor expressions are the same for each uniqueness case, and the two skolem constants in each uniqueness case appear in these two expressions in the position corresponding to the accessor function. The function collect_adt_unique_cases, whose definition is shown in Figure 17, expects as arguments 1) the list of accessors for a DATATYPE constructor, 2) a list of skolem constant names for the quantified variables in the uniqueness lemma for the constructor, which *by convention* are arranged in the lemma formulation so that the first two correspond to the first accessor, the second two correspond to the second accessor, and so on, and 3) and 4) two constructor expressions in which the skolem constants are correctly matched with their corresponding accessor positions.

```
(defun collect_adt_unique_cases (acclist skconstlist
                                  sk-expr-1 sk-expr-2)
  (cond ((null acclist) nil)
        (t (cons
            (mk_adt_unique_case
              (car acclist)
              (car skconstlist) (cadr skconstlist)
              sk-expr-1 sk-expr-2)
            (collect_adt_unique_cases
              (cdr acclist) (cddr skconstlist)
              sk-expr-1 sk-expr-2)))))
```

Fig. 17. A Lisp function that computes a list of uniqueness-case commands.

Note that to work correctly when it is applied, collect_adt_unique_cases must be given the appropriate arguments. Appropriate arguments can be computed from the formula in the lemma being proved. To compute the constructor expression instances corresponding the list of skolem constants, we need to know the names of the skolem constants. A convenient way to do this is to compute special skolem constant names from the list of bound variables in the lemma. Once the prover is invoked on the lemma, this can be done by using the Lisp function get_binding_names (see Figure 18) to probe the proof state for the names of

```
(defun get_binding_names (sform)
   (mapcar 'id (bindings (formula sform))))

(defun mk_adt_unique_skolem_names (varlis)
   (mapcar #'(lambda (varname)
                (concatenate 'string (string varname) "_uniq"))
           varlis))

(defun get_sk_constructor_exprs (sform)
   (exprs (argument (car (exprs (argument (formula sform)))))))
```

Fig. 18. Three auxiliary functions used in datatype_unique_strat.

the bound variables, and then applying the Lisp function mk_adt_unique_skolem_names to transform this list into a list of skolem names for the bound variables. The two constructor expressions are found by again probing the proof state, this time using the function get_sk_constructor_exprs.

Finally, we can define the proof rule adt_unique_strat, using the defstep macro, as shown in Figure 19. Note that both adt_unique_strat and its auxiliary rule adt_unique_strat_continue begin with a probe of the proof state *ps* to retrieve a value sform representing the current proof goal. The expected proof goal for adt_unique_strat corresponds to a uniqueness lemma. The initial call to (ASSERT) in adt_unique_strat assures that PVS has filled in all the fields in the CLOS structure for this goal, rather than lazily leaving them unbound. Both proof steps use the technique of first computing and then applying a command.

```
(defstep adt_unique_strat ()
   (then
     (assert)
     (let ((sform (car (s-forms (current-goal *ps*))))
           (bind-names (get_binding_names sform))
           (uniq-sk-names
             (mk_adt_unique_skolem_names bind-names))
           (cmd
             '(then (skolem 1 ,uniq-sk-names)
                    (adt_unique_strat_continue ,uniq-sk-names))))
     cmd))
   "" "")

(defstep adt_unique_strat_continue (sk-name-list)
   (let ((sform (car (s-forms (current-goal *ps*))))
         (sk-constr-exprs (get_sk_constructor_exprs sform))
         (sk-constr-expr-1 (car sk-constr-exprs))
         (sk-constr-expr-2 (cadr sk-constr-exprs))
         (constr-name (id (operator sk-constr-expr-1)))
         (all-constrs
           (constructors
             (adt (adt-type (operator sk-constr-expr-1))))))
   (let ((constr-form (car
           (select #'(lambda (x) (eq (id x) constr-name))
                   all-constrs)))
         (accessors (mapcar 'id (acc-decls constr-form)))
         (cases
           (collect_adt_unique_cases accessors sk-name-list
                                     sk-constr-expr-1
                                     sk-constr-expr-2))
         (cmd '(then (flatten) (spread (split) ,cases))))
   cmd))
   "" "")
```

Fig. 19. Defining a new proof rule adt_unique_strat.

The effect of the part of adt_unique_strat up to the point where it calls adt_unique_strat_continue is to skolemize the formula in the lemma using the skolem constants computed by mk_adt_unique_skolem_names. Thus, the value sform computed at the beginning of adt_unique_strat_continue corresponds to the skolemized version of the uniqueness lemma. Moreover, adt_unique_strat_continue is passed the list of skolem names as an argument so that it need not be recomputed. The step adt_unique_strat_continue proceeds by first computing the arguments it needs to pass to the function collect_adt_unique_cases, and uses the result of applying this function to the arguments in its computation of a proof command in the form of a strategy, which it then applies.

5 Discussion

Chapter 5 of the PVS Prover Guide [10, 11] contains much information useful to users who wish to write their own strategies. This information includes a description of global variables used in the prover, the CLOS slots in a proof state, methods for retrieving formulae and recognizing the class of an expression, several useful PVS functions including args1, args2, and gather-fnums, and the macros defstep, defhelper, and defstrat for defining new rules and strategies.

Several things could provide additional help for writing user strategies in PVS. One is simply easily accessible documentation of additional useful PVS functions and macros. Documentation of the helper functions used in the standard PVS strategies would eliminate duplication of effort on the part of PVS users who write their own strategies.

Currently, the CLOS structure must be probed to determine how to retrieve many details of the information on the proof state. Explicit documentation of this structure could allow this "probing" to be done off-line.

Strategies that explicitly reference the CLOS structure used for the internal representation of the PVS proof state must rely on the stability of this internal representation. An extra layer of "retrieval" functions whose names and effects would remain the same despite any changes in the internal representation of the proof state is one possibility for reducing the sensitivity of user strategies to any changes in the PVS implementation.

Even without these extra aids, however, it is possible for users to develop sophisticated strategies to serve their special needs—and to share with others.

Acknowledgement

Many thanks to N. Shankar for clarifying the semantics of TRY and providing several examples of its use. We also thank both N. Shankar and S. Owre for helpful comments.

References

1. Myla Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):139–181, 2000. Published Feb., 2001.
2. Myla Archer. Proving correctness of the basic TESLA multicast stream authentication protocol with TAME. In *Informal Proceedings of the Workshop on Issues in the Theory of Security (WITS'02)*, Portland, OR, Jan. 14–15 2002.
3. Myla Archer, Constance Heitmeyer, and Elvinia Riccobene. Proving invariants of I/O automata with TAME. *Automated Software Engineering*, 9(3):201–232, 2002.
4. M. Devillers, D. Griffioen, J. Romijn, and F. Vaandrager. Verification of a leader election protocol—formal methods applied to ieee 1394. *Formal Methods in System Design*, 16(3):307–320, June 2000.
5. B. Di Vito. A PVS prover strategy package for common manipulations. Technical Memorandum NASA/TM-2002-211647, NASA Langley Research Center, Hampton, VA, April 2002.
6. C. Muñoz and M. Mayero. Real automation in the field. Technical Report Interim ICASE Report No. 39, NASA/CR-2001-211271, ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681-2199, USA, December 2001.

7. S. Owre and N. Shankar. *PVS API Reference.* SRI Computer Science Laboratory, Menlo Park, California, USA, September 2003.
8. Adrian Perrig, Ran Canetti, J. D. Tygar, and Dawn Song. Efficient authentication and signing of multicast streams over lossy channels. In *Proc. of IEEE Security and Privacy Symposium (S&P2000)*, pages 56–73, May 2000.
9. John Rushby. Personal communication. 2003.
10. N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. The PVS prover guide. Technical report, Computer Science Lab., SRI Intl., Menlo Park, CA, 1998.
11. N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS Prover Guide, Version 2.4. Technical report, Computer Science Lab., SRI Intl., Menlo Park, CA, November 2001.
12. Guy L. Steele. *COMMON LISP: the Language.* Digital Press, Bedford, MA, 1990. Second edition.

Strategy-Enhanced Interactive Proving and Arithmetic Simplification for PVS

Ben L. Di Vito

NASA Langley Research Center, Hampton VA 23681, USA
`b.l.divito@larc.nasa.gov`
`http://shemesh.larc.nasa.gov/~bld`

Abstract. We describe an approach to strategy-based proving for improved interactive deduction in specialized domains. An experimental package of strategies (tactics) and support functions called Manip has been developed for PVS to reduce the tedium of arithmetic manipulation. Included are strategies aimed at algebraic simplification of real-valued expressions. A general deduction architecture is described in which domain-specific strategies, such as those for algebraic manipulation, are supported by more generic features, such as term-access techniques applicable in arbitrary settings. An extended expression language provides access to subterms within a sequent.

1 Introduction

Recent verification research at NASA Langley has emphasized extensive theorem proving over the domain of reals [4, 5], with PVS [15] serving as the primary proof tool. Efforts in this area have met with some difficulties, prompting a search for improved techniques for interactive proving. Significant productivity gains will be needed to fully realize our formal methods goals.

For arithmetic reasoning, PVS relies on decision procedures augmented by automatic rewriting. When a conjecture fails to yield to these tools, which often happens with nonlinear arithmetic, considerable interactive work may be required to complete the proof. Large productivity variances are the result.

SRI continues to increase the degree of automation in PVS. In particular, decision procedures for real arithmetic are a planned future enhancement. We look forward to these improvements. Nevertheless, there will always be a point where the automation runs out. When that point is reached, tactic-based[1] techniques can be applied to good effect.

In this paper we describe an approach to strategy-based proving for improved interactive deduction in specialized domains. An experimental package of strategies (tactics) and support functions called Manip has been developed for PVS to reduce the tedium of arithmetic manipulation. Included are strategies aimed at algebraic simplification of real-valued expressions. A general deduction architecture is described in which domain-specific strategies, such as those for algebraic manipulation, are supported by more generic features, such as term-access techniques applicable in arbitrary settings. User-defined proof strategies can be seen as a type of "deductive middleware." Our approach is general enough to serve other problem domains in the pursuit of such middleware.

By way of motivation, consider the following lemma for reasoning about trigonometric approximations:

$$0 < a \leq \pi/2 \supset |T_n(a)| > 2\,|T_{n+1}(a)| \qquad (1)$$

where $T_i(a)$ is the ith term in the power series expansion of the sine function:

$$\sin(a) = \sum_{i=1}^{\infty} (-1)^{i-1} a^{2i-1}/(2i-1)!\ .$$

[1] In PVS nomenclature, a *rule* is an atomic prover command while a *strategy* expands into one or more atomic steps. A *defined rule* is defined as a strategy but invoked as an atomic step. For our purposes, we regard the terms "tactic," "strategy" and "defined rule" as roughly synonymous.

Using only built-in rules, an early proof attempt for (1) required 68 steps. A common technique to carry out algebraic manipulation in such proofs is to use the case rule to force a case split on the (usually obvious) equality of two subexpressions, such as:

```
(CASE "a!1 * a!1 * (b!1 * b!1) = (b!1 * a!1) * (b!1 * a!1)")        (2)
```

Although not peculiar to PVS, this need to identify equivalent subexpressions and bring them to the prover's attention via cut-and-paste methods is rather awkward. It leads to a tedious style of proof that tries the patience of most users.

In contrast, by using the Manip package we were able to prove the lemma more naturally in 18 steps, 8 of which are strategies from our package, as shown in Fig. 1. Unlike the case-split technique, none of the steps contains excerpts from the sequent, such as those seen in (2). This proof represents one of the better examples of improvement from the use of our strategies. Although many proofs will experience a less dramatic reduction in complexity, the results have been encouraging thus far.

```
("" (SKOSIMP*)
    (REWRITE "sin_term_next")
    (RECIP-MULT! (! 1 R (-> "abs") 1))       ; strategy
    (APPLY (REPEAT (REWRITE "abs_mult")))
    (PERMUTE-MULT 1 R 3 R)                   ; strategy
    (OP-IDENT 1 L 1*)                        ; strategy
    (CANCEL 1)                               ; strategy
    (("1" (EXPAND "abs")
          (ASSERT)
          (PERMUTE-MULT 1 R 2 R)             ; strategy
          (CROSS-MULT 1)                     ; strategy
          (MULT-INEQ -2 -2)                  ; strategy
          (TYPEPRED "PI")
          (EXPAND "PI_ub")
          (MULT-INEQ -4 -4)                  ; strategy
          (ASSERT))
     ("2" (USE "sin_term_nonzero")
          (GRIND NIL :REWRITES ("abs")))))
```

Fig. 1. Proof steps for lemma (1) using built-in rules plus manipulation strategies

2 Architecture

We have integrated several elements to arrive at a strategy-based deduction architecture for user enhancements to PVS.

1. *Domain-specific proof strategies.* Common reasoning domains, such as nonlinear real arithmetic, provide natural targets for increasing automation. Extracting terms from sequents using suitable access facilities is vital for implementing strategies that do meaningful work.
2. *Extended expression language.* Inputs to existing prover rules are primarily formula numbers and expressions in the PVS language. For greater effectiveness, we provide users with a language for specifying subexpressions by location reference and pattern matching.
3. *Higher-order strategies with substitution.* Strategies that apply other proof rules offer the usual convenience of functional programming. Adding command-line substitutions derived from sequent expressions yields a more powerful way to construct and apply rules dynamically.
4. *Prelude extension libraries.* The PVS *prelude* holds built-in core theories. Strategies use prelude lemmas but often need additional facts. PVS's prelude extension feature adds such theorems in a manner transparent to the user.

5. *User-interface utilities.* To improve command line invocation of proof rules as well as offer various proof maintenance functions, a set of Emacs-based interface enhancements is included.

Note that only elements 1 and 4 are domain specific; the others are quite generic. In this paper we will focus on elements 1–3.

Several benefits accrue from the complementary elements of this architecture.

- User interaction is more natural, less laborious and occurs at a higher level of abstraction.
- Many manipulations apply lemmas from the prelude or its extensions. Strategies enable proving without explicit knowledge of these lemmas.
- The brittleness of proofs (breakage caused by changes in definitions or lemmas) is reduced by avoiding the inclusion of expressions from the current sequent in stored proof steps.
- Proving becomes more approachable for those with mathematical sophistication but little experience using mechanical provers.

We envision some features as being more useful during later stages of proof development, especially when finalizing a proof to make the permanent version more robust. During the early stages, it is easier to work directly with actual expressions. Once the outline of a proof is firm, extended expression features can be introduced to abstract away excessive detail.

3 Domain-Specific Strategies

Systematic strategy development for various domains could improve user productivity considerably. This section proposes a general scheme for structuring and implementing strategies in PVS and briefly sketches a particular set of strategies for manipulating arithmetic expressions.

3.1 Design Considerations

Input to the PVS prover is via Lisp s-expressions. Internally the prover uses CLOS (Common Lisp Object System) classes to represent expressions and other data. PVS provides macros for creating user-defined proof rules, which may include fragments of Lisp code to compute new values for invoking other rules.

We suggest the following guidelines for developing a strategy package.

1. *Introduce domain-relevant arguments.* For arithmetic strategies, a user typically needs to specify values such as the *side* of a relation (`L`, `R`), the *sign* of a term (`+`, `-`), and *term numbers*. Variations on the conventions of existing prover input handle these cases nicely.
2. *Augment term access functions.* Besides the access functions provided by the prover, additional ones may be needed to extract relevant values, e.g., the *i*th term of an additive expression. A modest set of access functions suffices for working with common language elements, such as arithmetic terms.
3. *Use text-based expression construction.* A proper implementation style would be to use object constructors to create new expression values. This requires knowledge of a large interface. Instead, it is adequate for most uses to exploit the objects' print methods and construct the desired expressions in textual form, which can then be supplied as arguments to other proof rules.
4. *Use Lisp-based symbolic construction.* To build final proof rules for invocation, the standard Lisp techniques for s-expression construction, such as backquote expressions, work well.
5. *Incorporate prelude extensions as needed.* When prelude lemmas are inadequate to support the desired deductions, a few judiciously crafted lemmas, custom designed for specific strategies, can be added invisibly.

Applications of items 1–4 are demonstrated in the simple example of Fig. 2. Most strategies are rather more complicated than this example, often requiring the services of auxiliary Lisp functions and intermediate helper strategies.

An example of a prelude extension lemma of the sort described in guideline (5) is the following:

```
div_mult_pos_neg_lt1: LEMMA
    z/n0y < x IFF IF n0y > 0 THEN z < x * n0y ELSE x * n0y < z ENDIF
```

```
(DEFSTEP has-sign (term &optional (sign +) (try-just nil))
  (LET ((term-expr (ee-obj-or-string (car (eval-ext-expr term))))
        (relation (case sign
                    ((+) '>) ((-) '<) ((0) '=)
                    ((0+) '>=) ((0-) '<=) ((+-) '/=) (t '>)))
        (case-step '(CASE ,(format nil "~A ~A 0" term-expr relation)))
        (step-list
          (list '(SKIP) (try-justification 'has-sign try-just))))
    (SPREAD case-step step-list))
  "Try claiming that a TERM has the designated SIGN (relationship to 0).
Symbols for SIGN are (+ - 0 0+ 0- +-), which have meanings positive,
negative, zero, nonnegative, nonpositive, and nonzero.  Proof of the
justification step can be tried or deferred.  Use TRY-JUST to supply
a step for the justification proof or T for the default rule (GRIND)."
  "~%Claiming the selected term has the designated sign")
```

Fig. 2. Sample strategy built using PVS `defstep` macro

This lemma simply combines two existing lemmas in prelude theory `real_props` into a conditional form to allow rewriting for any nonzero divisor. In ordinary settings, rewriting to such a conditional expression is likely to be undesirable. In this case, however, the lemma accommodates rewriting plus follow-up steps such as case splitting.

Following the design guidelines above will lead to strategies that are sound by construction. Prover objects are examined but not modified. Proof steps are obtained by expanding the strategies into rule applications for execution by the prover. New PVS expressions are submitted through the parser and typechecked. There are no mechanisms to enforce these good intentions, however. Coding errors could have unintended consequences, but with proper care there should be no side effects on the proof state.

3.2 Algebraic Manipulation Strategies

Users often want to manipulate expressions in the familiar style of conventional algebra, as one would do on paper. We now present a brief sampling of an arithmetic package to support this goal. Selected strategies are discussed that illustrate typical design choices. Appendix A lists the primary strategies in this family. Full details are available in a technical report [8] and user's manual [9].

- `move-terms` *fnum side* `&optional` (*term-nums* *)

  With `move-terms` a user can move a set of additive terms numbered *term-nums* in relational formula *fnum* from *side* (L or R) to the other side, adding or subtracting individual terms from both sides as needed. *term-nums* can be specified in a manner similar to the way formula numbers are presented to the prover. Either a list or a single number may be provided, as well as the symbol "*" to denote all terms on the chosen side. Example: invoking `(move-terms 3 L (2 4))` moves terms 2 and 4 from the left to the right side of formula 3.

- `cross-mult &optional` (*fnums* *)

  To eliminate divisions, `cross-mult` may be used to explicitly perform "cross multiplication" on one or more relational formulas. For example, $a/b < c/d$ will be transformed to $ad < cb$. The strategy determines which lemmas to apply based on the relational operator and whether negative divisors are involved. Cross multiplication is applied recursively until all outermost division operators are gone.

- `cancel &optional` (*fnums* *) (*sign* `nil`)

  When the top-level operator on both sides of a relation in *fnums* is the same operator drawn from the set $\{+, -, *, /\}$, `cancel` tries to eliminate common terms using a small set of rewrite rules and possible case splitting. Cancellation applies when *fnum* has the form $x \circ y \; R \; x \circ z$ or $y \circ x \; R \; z \circ x$. In the default case, when *sign* is `NIL`, x is assumed to be (non)positive or (non)negative as needed for the appropriate rewrite rules to apply. Otherwise, an explicit *sign* can be supplied to force a case split so the rules will apply. If *sign* is `+` or `-`, x is claimed to be strictly positive or negative. If *sign* is `0+` or `0-`, x is claimed to be nonnegative or nonpositive. If *sign* is `*`, x is assumed to be an arbitrary real and a three-way case split is used. Example: `(cancel 3 0+)` tries to cancel from both sides of formula 3 after first splitting on the assumption that the common term is nonnegative.

- `factor` *fnums* `&optional` (*side* `*`) (*term-nums* `*`) (*id?* `nil`)
  `factor!` *expr-loc* `&optional` (*term-nums* `*`) (*id?* `nil`)

  If the expression on *side* of each formula in *fnums* has multiple additive terms, `factor` may be used to extract common multiplicative factors and rearrange the expression. The additive terms indicated by *term-nums* are regarded as bags of factors to be intersected for common factors. Terms not found in *term-nums* are excluded from this process. In the `!`-variant, the *expr-loc* argument supplies a location reference to identify the target expression so that it may be factored in place. As an example, suppose formula 4 has the form

  ```
  f(x) = 2 * a * b + c * d - 2 * b
  ```

  and the command "`(factor 4 R (1 3))`" is issued. Then the strategy will rearrange formula 4 to:

  ```
  f(x) = 2 * b * (a - 1) + c * d
  ```

We provide several strategies for manipulating products or generating new products. This supports an overall approach of first converting divisions into multiplications where necessary, then using a broad array of tools for reasoning about multiplication. Three examples follow.

- `permute-mult` *fnums* `&optional` (*side* `R`) (*term-nums* `2`) (*end* `L`)

  For *end* = `L`, the action of `permute-mult` is as follows. Let the expression on *side* of a formula in *fnums* be a product of terms, $P = t_1 * \ldots * t_n$. Identify a list of indices I (*term-nums*) drawn from $\{1, \ldots, n\}$. Construct the product $t_{i_1} * \ldots * t_{i_l}$ where $i_k \in I$. Construct the product $t_{j_1} * \ldots * t_{j_m}$ where $j_k \in \{1, \ldots, n\} - I$. Then rewrite the original product P to the new product $t_{i_1} * \ldots * t_{i_l} * t_{j_1} * \ldots * t_{j_m}$. Thus the new product is a permutation of the original set of factors with the selected terms brought to the left. For *end* = `R`, the selected terms are placed on the right. Example: `(permute-mult 3 L (4 2))` rearranges the product on the left side of formula 3 to be `t4 * t2 * t1 * t3`, with the default association rules making it internally represented as `((t4 * t2) * t1) * t3`.

- `mult-eq` *rel-fnum eq-fnum* `&optional` (*sign* `+`)

  Given a relational formula $a\ R\ b$ and an antecedent equality $x = y$, `mult-eq` forms a new antecedent or consequent relating their products, $a*x\ R\ b*y$. If R is an inequality, the *sign* argument can be set to one of the symbols in `{+, -, 0+, 0-}` to indicate the polarity of x and y. Example: `(mult-eq -3 -2 -)` multiplies the sides of formula -3 by the sides of equality -2, which are assumed to be negative.

- `mult-ineq` *fnum1 fnum2* `&optional` (*signs* `(+ +)`)

  Given two relational formulas *fnum1* and *fnum2* having the forms $a\ R_1\ b$ and $x\ R_2\ y$, `mult-ineq` forms a new antecedent relating their products, $a*x\ R_3\ b*y$. If R_2 is an inequality having the opposite direction as R_1, `mult-ineq` proceeds as if it had been $y\ R_2'\ x$ instead, where R_2' is the reverse of R_2. The choice of R_3 is inferred automatically based on R_1, R_2, and the declared signs of the terms. R_3 is chosen to be a strict inequality if either R_1 or R_2 is. If either formula appears as a consequent, its relation is negated before carrying out the multiplication. Not all combinations of term polarities can produce useful results with `mult-ineq`. Therefore, the terms of each formula are required to have the same sign, designated by the symbols `+` and `-` in argument *signs*. Example: `(mult-ineq -3 -2 (- +))` multiplies the sides of inequality formula -3 by the sides of inequality -2, which are assumed to relate negative and positive values, respectively.

Figure 3 illustrates these strategies by displaying several proof steps for lemma (1) (see Fig. 1).

```
sin_terms_decr.1 :

[-1]  0 < a!1
[-2]  a!1 <= PI / 2
  |-------
{1}   1 > 2 *
        ((1 / (4 * (n!1 * n!1)
                + 2 * n!1))
          * --1 * a!1 * a!1)

Rule? (PERMUTE-MULT 1 R 2 R)

Permuting factors in selected
expressions, this simplifies to:
sin_terms_decr.1 :

[-1]  0 < a!1
[-2]  a!1 <= PI / 2
  |-------
{1}   1 > 2 * --1 * a!1 * a!1 *
          (1 / (4 * (n!1 * n!1)
                + 2 * n!1))

Rule? (CROSS-MULT 1)
```

```
Multiplying both sides of selected
 formulas by LHS/RHS divisor(s),
this simplifies to:
sin_terms_decr.1 :

[-1]  0 < a!1
[-2]  a!1 <= PI / 2
  |-------
{1}   1 * (4 * (n!1 * n!1) + 2 * n!1)
       > 2 * (--1 * a!1 * a!1)

Rule? (MULT-INEQ -2 -2)

Multiplying terms from formulas -2
 and -2 to derive a new inequality,
this simplifies to:
sin_terms_decr.1 :

{-1}  a!1 * a!1 <= (PI / 2) * (PI / 2)
[-2]  0 < a!1
[-3]  a!1 <= PI / 2
  |-------
[1]   1 * (4 * (n!1 * n!1) + 2 * n!1)
       > 2 * (--1 * a!1 * a!1)
```

Fig. 3. Proof trace fragment for selected steps from Fig. 1

4 Extended Expression Language

Many prover rules accept PVS expressions as arguments, which take the form of literal strings such as "`2 * PI * a!1`". Strategies in our package may be supplied extended expressions as well as the familiar text string form. This works equally well at the command line and within strategy definitions.

The main extensions provided are location references and textual pattern matching. Location references allow a user to indicate a precise subexpression within a formula by giving a path of indices to follow when descending through the formula's expression tree. Pattern matching allows strings to be found and extracted using a specialized pattern language that is based on, but much less elaborate than, regular expressions.

4.1 Location References

In the location reference form `(! <ext-expr> i1 ... in)`, the starting point `<ext-expr>` must describe the location of a valid PVS expression within the current sequent. Usually this is a simple formula number or one of the formula-list symbols $\{+,-,*\}$. The index values $\{i_j\}$ are used to descend the parse tree to arrive at a subexpression, which becomes the final value of the overall reference. Actually, the final value is a list of expressions, which allows for wild-card indices to traverse multiple paths through the tree. Moreover, the index values may include various other forms and indicators used to control path generation.

Location references may be used as arguments for certain strategies where a mere text string is inadequate. For example, the `factor!` strategy can factor an expression in place using this feature even if the target terms appear in the argument to a function. Thus, location references are reminiscent of array or structure references in procedural programming languages.

An example of a simple location reference is `(! -3 2)`, which evaluates to the right-hand side (argument 2) of formula -3. If this formula is "`x!1 = cos(a!1)`", then the string form of the location reference is "`cos(a!1)`". Adding index values reaches deeper into the formula, e.g., `(! -3 2 1)` evaluates to "`a!1`". Breadth can be achieved as well as depth; `(! -3 *)` evaluates to a list containing "`x!1`" and "`cos(a!1)`".

Index values and directives $\{i_j\}$ may assume one of the following forms:

- An integer i in the range $1, \ldots, k$, where k is the arity of the function at the current point in the expression tree. Paths follow the i^{th} branch or argument, returning the argument as value if i is the last index. The symbols `L` and `R` are synonyms for 1 and 2.
- The index value 0, which returns the function symbol of the current expression. If the function is itself an expression, as $f(x)$ in $f(x)(y)$, indices after the 0 will retrieve components of the expression.
- The *wild-card* symbol `*`, which indicates that this path should be replicated for each argument expression, returning values from all n paths.
- A list `(j1 ... jm)` of integers indicating which argument paths should be included for replication, i.e., a subset of the `*` case.
- One of the *deep wild-card* symbols {`-*`, `*-`, `**`}, which indicates that this path should be replicated as many times as needed to visit all nodes in the current subtree. The values returned are the leaf objects (terminal nodes) for `-*`, the nonterminal nodes for `*-`, and all nodes (subexpressions) for `**`.
- A text string serving as a *guard* to select desired paths from multiple candidates. If the current function symbol matches the string, path elaboration continues. Otherwise, the path is terminated, returning an empty list.
- A list `(s1 ... sk)` of strings that serves as a guard by matching each pattern s_i in the manner of Section 4.2.
- A form `(-> g1 ... gk)` that serves as a *go-to* operator to specify a systematic search down and across the subtree until the first path is found having intermediate points satisfying all the guards $\{g_i\}$ in sequence. The form `(->* g1 ... gk)` returns all eligible paths.

Table 1 illustrates the formulation of location references using this notation.

Note that indexing works for both infix and prefix function applications. For arithmetic expressions, special indexing rules result in some "flattening" of the parse tree during traversal. These conventions are more convenient for arithmetic terms and correspond more closely to our usual algebraic intuition for numbering terms. In particular, additive (multiplicative) terms are counted left to right irrespective of the associative groupings that may be in effect. They are treated as if they were all arguments of a single addition/subtraction (multiplication) operator of arbitrary arity.

In practice, not all of the location reference features are likely to be equally useful. We provide a variety of traversal and search mechanisms to ensure some measure of thoroughness. Some users may choose to limit themselves to simple numeric indexing.

Table 1. Examples of location reference expressions applied to the formulas below

Loc. reference	Expr. strings	Loc. reference	Expr. strings
`(! -2)`	`r!1 = 2 * x!1 + 1`	`(! -1 L • *)`	`x!1, r!1, y!1, r!1`
`(! -2 R)`	`2 * x!1 + 1`	`(! 1 R 1 **)`	`sq(x!1 / 4),`
`(! -2 R 1)`	`2 * x!1`		`x!1 / 4, x!1, 4`
`(! -1 L 2 1)`	`y!1`	`(! - "=")`	`r!1 = 2 * x!1 + 1`
`(! 1 R 1)`	`sq(x!1 / 4)`	`(! -2 * "+")`	`2 * x!1 + 1`
`(! -2 *)`	`r!1, 2 * x!1 + 1`	`(! 1 (-> "sq"))`	`sq(x!1 / 4)`
`(! -1 L 2 *)`	`y!1, r!1`	`(! 1 (-> "sq") 1)`	`x!1 / 4`
`(! -1 L * 1)`	`x!1, y!1`	`(! -1 (->* "*"))`	`x!1 * r!1, y!1 * r!1`

```
{-1}  x!1 * r!1 + y!1 * r!1 > r!1 - 1
[-2]  r!1 = 2 * x!1 + 1
  |-------
[1]   sqrt(r!1) < sqrt(sq(x!1 / 4))
```

4.2 Pattern Matching

Each pattern p_j in `(? <ext-expr> p1 ... pn)` is expressed as a text string using a specialized pattern language. Unlike location references, pattern matches usually produce only a text string and lack a corresponding CLOS object for a PVS expression. The patterns $p_1, \ldots, p_n$ are applied in order to the textual representation of each member of the base expression list. In each case, matching stops after the first successful match among the $\{p_j\}$ is obtained. All resulting output strings are collected and concatenated into a single list of output strings.

A pattern string may denote either a *simple* or a *rich* pattern. Simple patterns are easier to express and are expected to suffice for many everyday applications. When more precision is required, rich patterns offer more expressive power.

Simple patterns allow matching against literal characters, whitespace fields, and arbitrary substrings. Pattern strings comprise a mixture of literal characters and meta-strings for designating text fields. Meta-strings denote either whitespace or non-whitespace fields. A whitespace field is indicated by a space character in the pattern. A non-whitespace field is a meta-string consisting of the percent (`%`) character followed by a digit character (0–9), which matches zero or more arbitrary characters in the target string.

Both capturing and non-capturing fields are provided. A capturing field causes the matching substring to be returned as an output. The meta-string `%0` denotes a noncapturing field, while those with nonzero digits are capturing fields. If a nonzero digit d is the first occurrence of d in the pattern, a new capturing field is thereby indicated. Otherwise, it is a reference to a previously captured field whose contents must be matched. Table 2 illustrates the formulation of simple patterns using this notation.

Rich patterns follow the same basic approach as simple patterns, but add features for multiple matching types and multiple text-field types. The match types include full and partial string matching as well as top-down and bottom-up expression matching.

Table 2. Examples of simple pattern matching applied to the formulas below

Pattern	Matching string(s)	Captured fields
`(? 1 "%1(r!1)")`	`sqrt(r!1)`	`sqrt`
`(? 1 "sqrt(sq(%1))")`	`sqrt(sq(x!1 / 4))`	`x!1 / 4`
`(? -2 "r!1 = %1")`	`r!1 = 2 * x!1 + 1`	`2 * x!1 + 1`
`(? 1 "%1(%0) <")`	`sqrt(r!1) <`	`sqrt`
`(? -1 "> %1 - %0")`	`> r!1 - 1`	`r!1`
`(? 1 "%1(%0) < %1(%2)")`	All of formula 1	`sqrt, sq(x!1 / 4)`

```
{-1}  x!1 * r!1 + y!1 * r!1 > r!1 - 1
[-2]  r!1 = 2 * x!1 + 1
  |-------
[1]   sqrt(r!1) < sqrt(sq(x!1 / 4))
```

5 Higher-Order Strategies with Substitution

Extended expressions allow us to capture subexpressions from the current sequent. Next we add a parameter substitution technique to formulate prover commands. To complete the suite, we add higher-order strategies that substitute strings and formula numbers into parameterized commands. These features are intended primarily for command line use. In LCF-family provers, ML scripting can achieve similar effects.

5.1 Parameter Substitution

A parameterized command is regarded as a template expression (actually, a Lisp *form*) in which embedded text strings and special symbols serve as substitutable parameters. The outcome of evaluating extended expressions is used to carry out textual and symbolic substitutions. Each descriptor computed during evaluation contains a text string and, optionally, a formula number and CLOS object. Descriptors are the source of substitution data while the parameterized command is its target.

Within this framework, we allow two classes of substitutable data: literal text strings and Lisp symbols. The top-level s-expression is traversed down to its leaves. Wherever a string or symbol is encountered, a substitution is attempted. The final command thus produced will be invoked as a prover command in the manner defined for the chosen higher-order strategy. (In Lisp programming terms, this process can be imagined as evaluating a backquote expression with specialized implicit unquoting. It also has some similarities to substitution in Unix shell languages as well as the scripting language Tcl.)

Parametric variables for substitution are allowed as follows. Within literal text strings, the substrings `%1`, ..., `%9` serve as implicit text variables. The substring `%1` will be replaced by the string component of the first expression descriptor. The other `%`-variables will be replaced in order by the corresponding strings of the remaining descriptors.

Certain reserved symbols beginning with the `$` character serve as symbolic parameters. Such symbols are not embedded within strings as are the `%`-variables; they appear as stand-alone symbols within the list structure of the parameterized command. The symbols `$1`, `$2`, etc., represent the first, second, etc., expression descriptors from the list of available descriptors.

Variants of these symbols exist to retrieve the text string, formula number, and CLOS object components of a descriptor. These are needed to supply arguments for built-in prover commands, which are not cognizant of extended expressions. The symbols `$1s`, `$1n` and `$1j` serve this purpose. Aggregations may be obtained using the symbol `$*` and it variants. Table 3 summarizes the special symbols usable in substitutions.

Table 3. Special symbols for command substitution

Symbol	Value
`$1, $2, ...`	nth expression descriptor
`$*`	List of all expression descriptors
`$1s, $2s, ...`	nth expression string
`$*s`	List of all expression strings
`$1n, $2n, ...`	Formula number for nth expression
`$+n`	List of formula numbers (no duplicates)
`$*n`	List of all formula numbers (includes duplicates)
`$1j, $2j, ...`	CLOS object for nth expression
`$*j`	List of all CLOS objects

5.2 Invocation Strategies

Next we describe a set of general-purpose, higher-order strategies. They are not specialized for arithmetic. Some offer generic capabilities useful in implementing other strategies for specific purposes. For each of these strategies, multiple expression specifications may be supplied as arguments. In such cases, each specification gives rise to an arbitrary number of descriptors. All descriptor lists are then concatenated to build a single list before substitutions are performed. Table 4 lists the strategies provided; several are discussed below.

`invoke` *command* `&rest` *expr-specs*

This strategy is used to invoke *command* after applying substitutions extracted by evaluating the expression specifications *expr-specs*.

As an example, suppose formula 3 is

```
f(x!1 + y!1) <= f(a!1 * (z!1 + 1))
```

Then the command

Table 4. Summary of higher-order strategies

Syntax	Function
`(invoke command &rest expr-specs)`	Invoke command by instantiating from expressions and patterns
`(for-each command &rest expr-specs)`	Instantiate and invoke separately for each expression
`(for-each-rev command &rest expr-specs)`	Invoke in reverse order
`(show-subst command &rest expr-specs)`	Show but don't invoke the instantiated command
`(claim cond &opt (try-just nil) &rest expr-specs)`	Claims condition on terms
`(name-extract name &rest expr-specs)`	Extract & name expr, then replace

```
(invoke (case "%1 <= %2") (? 3 "f(%1) <= f(%2)"))
```

would apply pattern matching to formula 3 to create bindings %1 = "x!1 + y!1" and %2 = "a!1 * (z!1 + 1)", which would result in the prover command

```
(case "x!1 + y!1 <= a!1 * (z!1 + 1)")
```

being invoked. An alternative way to achieve the same effect using location referencing is the following:

```
(invoke (case "%1 <= %2") (! 3 * 1))
```

As another example, suppose we wish to hide most of the formulas in the current sequent, retaining only those that mention the sqrt function. We search for all formulas containing a reference to sqrt using a simple pattern, then collect all the formula numbers and use them to invoke the hide-all-but rule:

```
(invoke (hide-all-but ($+n)) (? * "sqrt"))
```

for-each *command* &rest *expr-specs*

This strategy is used to invoke *command* repeatedly, with a different substitution for each expression generated by *expr-specs*. The effect is equivalent to applying (invoke command e_i) *n* times.

As an example, suppose we wish to expand every function in the consequent formulas having the string "cos" as part of its name. The following command carries this out, assuming there is only one instance per formula.

```
(for-each (expand "%1") (! + *- ("cos") 0))
```

for-each-rev *command* &rest *expr-specs*

This strategy is identical to for-each except that the expressions are taken in reverse order.

Imagine we wish to find all antecedent equalities and use them for replacement, hiding each one as we go. This needs to be done in reverse order because formula numbers will change after each replacement.

```
(for-each-rev (replace $1n :hide? t) (! - "="))
```

claim *cond* &optional (*try-just* nil) &rest *expr-specs*

The claim strategy is basically the same as the primitive rule case, except that the condition is derived using the parameterization technique. The condition presented in *cond* is instantiated by the terms found in *expr-specs*. Argument try-just allows the user to try proving the justification step (the second case resulting from the case split).

For example, to claim that a numerical expression lies between two others, we could use something like

```
(claim "%1 <= %2 & %2 <= %3" nil "a/b" "x+y" "c/d")
```

to generate a case split on the formula "`a/b <= x+y & x+y <= c/d`".

Invocation strategies are useful as building blocks for more specialized strategies that users might need for particular circumstances. Extended expressions can support an alternative to the more code-intensive strategy-writing style that requires accessing the data structures (CLOS objects) representing PVS expressions. This alternative can make lightweight strategy writing more accessible to users without a deep background in Lisp programming.

6 Related Work

Tactic-based proving was pioneered by Milner and advanced by many others, beginning with the work on Edinburgh LCF [11]. The introduction of ML and its use for accessing subterms was also introduced in LCF. Constable and his students developed the Nuprl system [6], which included heavy reliance on tactic-based proof techniques. Tactic-based proving also has been used extensively in more recent interactive provers such as HOL [10], Isabelle [16] and Coq [12]. Although much of this has been devoted to low-level automation, there also have been higher level tactics developed.

In the case of PVS, strategy development has not been as much a focus as tactic development has been for provers in the LCF family. Partly this is due to greater use of decision procedures in PVS as well as an increasing emphasis on rewrite rules. For example, Shankar [17] sketches an approach to the use of rewrite libraries for arithmetic simplification. While these methods are certainly helpful, we believe they need to be augmented by proof interaction of the sort we advocate.

Several researchers have developed PVS strategy packages for specialized types of proving. Examples include a mechanization of the TRIO temporal logic [1], a proof assistant for the Duration Calculus [18], and the verification of simple properties for state-based requirements models [7]. A notable example is Archer's account of the TAME effort [2], which has a good discussion on developing PVS strategies for timed automata models and using them to promote "human-style" theorem proving.

In the area of arithmetic strategy packages for PVS, a semi-decision procedure for the field of real numbers [13], which had been developed originally for Coq, was recently ported to PVS. This package is called Field; it achieves simplification by eliminating divisions and rearranging multiplicative terms extensively. Field has been designed to use some Manip strategies for working with multiplication. César Muñoz continues to enhance Field and maintains an active line of development.

Our work on Manip emphasizes applied interactive proving, features for extracting terms from the working sequent, and flexible mechanisms for exploiting such terms. Many PVS strategy approaches stress control issues, giving less attention to the equally important data issues. Only by placing nontrivial term-access facilities at the user interface can the full potential of interactive strategies be realized.

In a typical control-oriented approach, a strategy might have several plausible sets of rules to apply in speculative fashion. If a given try fails to produce results, bracktracking is performed and an alternative is attempted. By placing more emphasis on data or proof state, the strategy can determine which alternative to select based on attributes of the current state. Allowing users to indicate relevant terms from the sequent sharpens the focus even further during interactive proving.

Currently under study are term access features that allow selection by mouse gestures. "Proof by pointing" techniques [3] are examples of applicable methods that can improve usability in this area. Once selected, a term can be matched with an extended expression for locating it. This can be done without burdening the user to derive the extended expression.

7 Conclusion

The Manip arithmetic package has been used experimentally at NASA Langley and made available to the PVS user community. Along with Field [13], it is now being used to prove new lemmas as they are introduced in Langley's PVS libraries [14]. Proofs for the real analysis and vectors libraries, in particular, have made regular use of Manip strategies. As of May 2003, a total of 325 Manip strategy instances were counted in the proofs distributed as part of the Langley libraries. Further evaluation is needed to gauge effectiveness and suggest new strategies.

Tactic-based theorem proving still holds substantial promise for automating domain-specific reasoning. In the case of PVS, much effort has gone into developing decision procedures and rewrite rule capabilities. While these are undoubtedly valuable, there is still ample room for other advances, particularly those that can leverage the accumulated knowledge of experienced users of deduction systems. Such users are well poised to introduce the wide variety of deductive middleware needed by the formal methods and computational logic communities. Our tools and techniques aim to further this goal.

Future activities will focus on refining the techniques and introducing new strategy packages for additional domains. One domain of interest is reasoning about sets, especially finite sets. We expect that ideas from the arithmetic strategies can be readily adapted.

Acknowledgments

The need for this package and many initial ideas on its operation were inspired by Ricky Butler of NASA Langley. Additional ideas and useful suggestions have come from César Muñoz of NIA, and John Rushby and Sam Owre of SRI.

References

1. A. Alborghetti, A. Gargantini, and A. Morzenti. Providing automated support to deductive analysis of time critical systems. In *Proc. of the 6th European Software Engineering Conf. (ESEC/FSE'97)*, volume 1301 of *LNCS*, pages 221–226, 1997.
2. Myla Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):139–181, 2000.
3. Y. Bertot, G. Kahn, and L. Théry. Proof by pointing. In *Proc. of Theoretical Aspects of Computer Software (TACS'94)*, volume 789 of *LNCS*, 1994.
4. R.W. Butler, V. Carreño, G. Dowek, and C. Muñoz. Formal verification of conflict detection algorithms. In *Proceedings of the 11th Working Conference on Correct Hardware Design and Verification Methods CHARME 2001*, volume 2144 of *LNCS*, pages 403–417, Livingston, Scotland, UK, 2001.
5. Víctor Carreño and César Muñoz. Aircraft trajectory modeling and alerting algorithm verification. In *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, pages 90–105, 2000.
6. R. L. Constable and et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.
7. Ben L. Di Vito. High-automation proofs for properties of requirements models. *Software Tools for Technology Transfer*, 3(1):20–31, September 2000.
8. Ben L. Di Vito. A PVS prover strategy package for common manipulations. NASA Technical Memorandum NASA/TM-2002-211647, April 2002.
9. Ben L. Di Vito. *Manip User's Manual, Version 1.1*, February 2003. Complete package available through NASA Langley PVS library page [14].
10. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.
11. M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.* Springer LNCS 78, 1979.
12. INRIA. *The Coq Proof Assistant Reference Manual, Version 7.1*, October 2001.
13. C. Muñoz and M. Mayero. Real automation in the field. NASA/CR-2001-211271 Interim ICASE Report No. 39, December 2001.
14. NASA Langley PVS library collection. Theories and proofs available at `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS2-library/pvslib.html`.
15. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
16. L. C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer LNCS 828, 1994.
17. N. Shankar. Arithmetic simplification in PVS. Final Report for SRI Project 6464, Task 15; NASA Langley contract number NAS1-20334., December 2000.
18. J. Skakkebaek and N. Shankar. Towards a duration calculus proof assistant in PVS. In *Third Intern. School and Symp. on Formal Techniques in Real Time and Fault Tolerant Systems*, volume 863 of *LNCS*, 1994.

A Algebraic Manipulation Strategies

The following list summarizes the set of manipulation strategies. A few variants have been omitted in the interest of brevity.

Syntax	Function
`(swap lhs operator rhs &opt (infix? T))`	$x \circ y \Longrightarrow y \circ x$
`(group term1 operator term2 term3 &opt (side L) (infix? T))`	L: $x \circ (y \circ z) \Longrightarrow (x \circ y) \circ z$ R: $(x \circ y) \circ z \Longrightarrow x \circ (y \circ z)$
`(swap-group term1 operator term2 term3 &opt (side L) (infix? T))`	L: $x \circ (y \circ z) \Longrightarrow y \circ (x \circ z)$ R: $(x \circ y) \circ z \Longrightarrow (x \circ z) \circ y$
`(swap-rel &rest fnums)`	Swap sides and reverse relations
`(equate lhs rhs &opt (try-just nil))`	$\ldots lhs \ldots \Longrightarrow \ldots rhs \ldots$
`(has-sign term &opt (sign +) (try-just nil))`	Claims term has sign indicated
`(mult-by fnums term &opt (sign +))`	Multiply both sides by term
`(div-by fnums term &opt (sign +))`	Divide both sides by term
`(split-ineq fnum &opt (replace? nil))`	Split $\leq$ ($\geq$) into $<$ ($>$) and $=$ cases
`(flip-ineq fnums &opt (hide? T))`	Negate and move inequalities
`(move-terms fnum side &opt (term-nums *))`	Move additive terms to other side
`(isolate fnum side term-num)`	Move all but one term
`(isolate-replace fnum side term-num &opt (targets *))`	Isolate then replace with equation
`(cancel &opt (fnums *) (sign nil))`	Cancel terms from both sides
`(cancel-terms &opt (fnums *) (end L) (sign nil) (try-just nil))`	Cancel speculatively & defer proof
`(op-ident fnum &opt (side L) (operation *1))`	Apply operator identity to rewrite expression
`(cross-mult &opt (fnums *))`	Multiply both sides by denom.
`(cross-add &opt (fnums *))`	Add subtrahend to both sides
`(factor fnums &opt (side *) (term-nums *) (id? nil))`	Extract common multiplicative factors from additive terms given
`(transform-both fnum transform &opt (swap nil) (try-just nil))`	Apply transform to both sides of formula
`(permute-mult fnums &opt (side R) (term-nums 2) (end L))`	Rearrange factors in a product
`(name-mult name fnum side &opt (term-nums *))`	Select factors, assign name to their product, then replace
`(recip-mult fnums side)`	$x/d \Longrightarrow x * (1/d)$
`(isolate-mult fnum &opt (side L) (term-num 1) (sign +))`	Select a factor and divide both both sides to isolate factor
`(mult-eq rel-fnum eq-fnum &opt (sign +))`	Multiply sides of relation by sides of equality
`(mult-ineq fnum1 fnum2 &opt (signs (+ +)))`	Multiply sides of inequality by sides of another inequality
`(mult-cases fnum &opt (abs? nil) (mult-op *1))`	Generate case analyses for products
`(mult-extract name fnum &opt (side *) (term-nums *))`	Extract selected terms, name replace them, then simplify

First-Order Proof Tactics in Higher-Order Logic Theorem Provers

Joe Hurd*

Computer Laboratory
University of Cambridge,
joe.hurd@cl.cam.ac.uk

Abstract. In this paper we evaluate the effectiveness of first-order proof procedures when used as tactics for proving subgoals in a higher-order logic interactive theorem prover. We first motivate why such first-order proof tactics are useful, and then describe the core integrating technology: an 'LCF-style' logical kernel for clausal first-order logic. This allows the choice of different logical mappings between higher-order logic and first-order logic to be used depending on the subgoal, and also enables several different first-order proof procedures to cooperate on constructing the proof. This work was carried out using the HOL4 theorem prover; we comment on the ease of transferring the technology to other higher-order logic theorem provers.

1 Introduction

Performing interactive proof in the HOL theorem prover [12] involves reducing goals to simpler subgoals. It turns out that many of these subgoals can be efficiently 'finished off' by an automatic first-order prover. To fill this niche, Harrison implemented a version of the MESON procedure [13] with the ability to translate proofs to higher-order logic. The original MESON procedure, due to Loveland [17], is a sound and complete calculus for finding proofs in full first-order logic. This was integrated as a HOL tactic in 1996, and has since become a standard workhorse of interactive proof. Today, building all the theories in the most recent distribution of HOL4[1] relies on MESON to prove 1726 subgoals; the HOL formalization of probability theory, including an example verification of the Miller-Rabin primality test, contributes another 1953 subgoals to this total.

The primary goal of this paper is to evaluate the effectiveness of different first-order proof calculi for use as HOL tactics supporting interactive proof. We compare the performance of several first-order calculi on three different problem sets: the TPTP first-order problem set; and two problem sets derived from HOL subgoals proved by MESON. The TPTP (Thousands of Problems for Theorem Provers) problem set is a collection of first-order problems designed to test the limits of current automatic first-order provers [24]. This experiment allows us to directly compare the performance of the first-order proof procedures in the different environments of fully automatic proof of deep theorems and supporting a user engaged in driving an interactive theorem prover. In this paper we show that performance in these two environments is correlated. Therefore, if a new first-order prover is developed that can prove more TPTP problems than the existing state of the art, we can expect the same prover to prove more HOL subgoals, thus improving the user experience.

The most obvious difference between the fully-automatic and interactive environments is the real-time nature of interactive proof. Whether the cost of proof search is incurred each time the theory is loaded, compiled, or even just once when the theory is created, the user usually requires any HOL tactic to respond (almost) immediately. By contrast, fully automatic provers are generally judged on the number of theorems that they can prove within a much more relaxed time-limit. To simulate this environmental difference in our experiments, for the TPTP problem set we allow the provers 60 seconds per problem, and for the HOL problem sets we allow only 5 seconds per problem.

A limit of 5 seconds per problem suggests that the performance of a prover may be rather sensitive to the characteristics of initial proof search. Following this reasoning, it seems plausible that a combination of different proof procedures may perform better than any individual, on the grounds that any problems

* This work was supported by EPSRC project GR/R27105/01

[1] HOL4 is available at http://hol.sf.net/.

that are 'shallow' for one of the procedures may be quickly solved within the time limit. We therefore implement a proof procedure that combines resolution, model elimination, and the Delta preprocessor. All three procedures may be run in parallel (using time slicing), and they cooperate by sharing unit clauses.[2] It turns out that not only does this combination procedure significantly outperform each individual procedure on the HOL problem sets, but also on the TPTP problem set.

The device that allows the provers to share unit clauses is a small 'LCF-style' kernel for clausal first-order logic. As well as providing a convenient mechanism for detecting unit clauses derived by model elimination, it also provides a convenient place to install the proof recording in the event that it is necessary to translate them to HOL. In particular, it is the only place that needs to worry about keeping track of proofs, and this enabled broader experimentation with the first-order provers.

Since HOL4 is written in Standard ML, this is a convenient implementation language for our experiment, though in the past similar experiments have been performed by making calls to external C provers [14]. Therefore, this paper also provides a view of implementing first-order proof procedures in a functional programming language, and some interesting aspects of this are brought out in discussion.

The secondary goal of this paper is to serve as a 'HOW-TO guide' for would-be implementors of first-order proof tactics in higher-order theorem provers. We will present all the steps necessary to prove higher-order subgoals using automatic first-order provers: the initial conversion from higher-order subgoal to first-order clauses; the first-order proof search; and the final translation of the first-order refutation to a higher-order logic theorem.

The main contributions of this paper are as follows:

- A relative performace comparison of different first-order proof calculi in the two environments of proving deep first-order theorems (the TPTP problem set) and aiding the user engaged in interactive proof (the two problem sets extracted from HOL subgoals).
- A combination resolution and model elimination procedure that performs significantly better than either individually, in both the TPTP and the HOL environments.
- A detailed description of how to implement tactics for proving higher-order logic subgoals using first-order proof procedures.

The paper is structured as follows: in Section 2 we point out the interesting features of the mapping between higher-order and first-order logic; Section 3 examines the syntactic differences between the problem sets and presents our evaluation methodology; in Section 4 we describe the ML implementation of the first-order provers and their subsequent optimization; Section 5 presents the results of running different combinations of provers on the problem sets; Section 6 comments on how this technology might be ported to other higher-order logic theorem provers; and finally in Sections 7 and 8 we conclude and take a look at related work.

2 The HOL Interface to First-Order Logic

This is the high-level view of how we prove the HOL goal g using a first-order prover:

1. We first convert the negation of g to conjunctive normal form; this results in a HOL theorem of the form
$$\vdash \neg g \iff \exists \boldsymbol{a}.\ (\forall \boldsymbol{v_1}.\ c_1) \wedge \cdots \wedge (\forall \boldsymbol{v_n}.\ c_n) \tag{1}$$
where each c_i is a HOL term having the form of a disjunction of literals, and may contain variables from the vectors $\boldsymbol{a}$ and $\boldsymbol{v_i}$.
2. Next, we map each HOL term c_i to first-order logic, producing the clause set
$$C = \{C_1, \ldots, C_n\}$$
3. The first-order prover runs on C, and finds a refutation ρ.

[2] Unit clauses are clauses with only one literal, and are used to simplify other clauses.

4. By proof translation, the refutation ρ is lifted to a HOL proof of the theorem
$$\{(\forall \mathbf{v_1}.\ c_1), \ldots, (\forall \mathbf{v_n}.\ c_n)\} \vdash \bot \tag{2}$$
5. Finally, some HOL primitive inferences use theorems (1) and (2) to derive
$$\vdash g \tag{3}$$

In the following subsections we examine the translation of formulas and proofs between higher-order logic and first-order logic, which plays a role in steps 2 and 4 of the above process. Much of this information appears in a previously published system description [15]; it is reproduced here because it is an essential part of our framework for creating first-order proof tactics.

Before getting into the details, we first give an extended example of the whole process with a typical HOL subgoal that we prove using a first-order proof tactic. Consider the subgoal
$$\forall x, y, z.\ \mathsf{divides}\ x\ y \Rightarrow \mathsf{divides}\ x\ (z * y) \tag{4}$$
where the predicate divides is defined as
$$\vdash \forall x, y, z.\ \mathsf{divides}\ x\ y \iff \exists z.\ y = z * x \tag{5}$$
To prove the subgoal, we also need the following theorems about multiplication:
$$\vdash \forall x, y.\ x * y = y * x \tag{6}$$
$$\vdash \forall x, y, z.\ (x * y) * z = x * (y * z) \tag{7}$$
The user invokes the first-order proof tactic on the subgoal (4), passing as arguments the definition (5) and theorems (6) and (7). Initially, the first-order proof tactic uses the arguments to set up the equivalent subgoal g:
$$(5) \land (6) \land (7) \Rightarrow (4)$$
The next step is to negate g and convert to conjunctive normal form. This conversion is completely standard—negation normal form followed by pushing out quantifiers and Skolemization—and we refer the interested reader to a textbook such as Chang and Lee [6] for more details. In our example, this results in the theorem
$$\begin{array}{l} \vdash \neg g \iff \\ \quad \exists a, b, c, d. \\ \quad\quad (\forall x, y.\ x * y = y * x)\ \land \\ \quad\quad (\forall x, y, z.\ (x * y) * z = x * (y * z))\ \land \\ \quad\quad (\forall x, y, z.\ \neg(y = z * x)\ \lor\ \mathsf{divides}\ x\ y)\ \land \\ \quad\quad (\forall x, y.\ \neg\mathsf{divides}\ x\ y\ \lor\ y = d\ x\ y * x)\ \land \\ \quad\quad \mathsf{divides}\ a\ b\ \land \\ \quad\quad \neg\mathsf{divides}\ a\ (c * b) \end{array}$$
We map each line of this formula to a first-order clause. The existential variables a, b, c, d are mapped to first-order function symbols, and the universal variables x, y, z are mapped to first-order variables. The first-order prover runs, finds a refutation, and this is translated to a HOL theorem from which the first-order tactic derives $\vdash g$, thus proving the initial goal.

2.1 Mapping HOL Terms to First-Order Logic

Seemingly the hardest problem with mapping HOL terms to first-order logic—dealing with λ-abstractions—can be smoothly dealt with as part of the conversion to CNF. Any λ-abstraction at or beneath the literal level is rewritten to combinatory form, using the set of combinators $\{S, K, I, C, \circ\}$. Using this set of combinators prevents the exponential blow-up that is encountered when only $\{S, K, I\}$ are used [25].[3]

The mapping that we use makes explicit function application, so that the HOL term $m + n$ maps to the first-order term $@(@(+, m), n)$. Since in HOL there is no distinction between terms and formulas, we model this in first-order logic by defining a special relation called B (short for Boolean) that converts a first-order term to a first-order formula. For example, the HOL boolean term $m \leq n$ is mapped to the first-order formula $B(@(@(\leq, m), n))$. The only exception to this rule is equality: the HOL term $x = y$ is mapped to the first-order logic formula $=(x, y)$.

As described thus far, this mapping is used to generate the **uHOL** first-order problem set from HOL subgoals sent to MESON. uHOL stands for untyped HOL, because no type information is included in this representation. However, we also experimented with including higher-order logic types in the first-order mapping of a HOL term. Using this idea, the HOL term $m + n$ would map to the first-order term

$$@(@(+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}, m : \mathbb{N}) : \mathbb{N} \to \mathbb{N}, n) : \mathbb{N}$$

where ':' is a binary function symbol (written infix for readability), and higher-order logic types are encoded as first-order terms.[4] This mapping is used to produce the **HOL** problem set from HOL subgoals. As might be expected, this produces much larger first-order clauses than omitting the types, and this results in first-order deduction steps taking longer to perform. However, we cannot conclude that including types is definitely harmful: the extra information may pay for itself by cutting down the search space. This hypothesis is examined in Section 5.

2.2 Translating First-Order Refutations to HOL

When the first-order prover has found a refutation of a set of clauses, the HOL tactic must translate the refutation to a HOL theorem, thus ensuring that no soundness bugs in the first-order prover are propagated into HOL. At first sight it may appear that the necessity of translating first-order refutations to higher-order logic proofs imposes a burden that hampers free experimentation with the first-order provers. However, by applying the logical kernel idea of the LCF project [11], we can make the proof translation invisible to the developer of first-order proof procedures, leaving him free to experiment with new calculi. We have implemented this automatic proof translation for both the mapping with type information and the one without, and they have been successfully used to prove many HOL subgoals.

$$\frac{}{A_1 \vee \cdots \vee A_n}\ \textsf{AXIOM}\ [A_1, \ldots, A_n] \qquad\qquad \frac{}{L \vee \neg L}\ \textsf{ASSUME}\ L$$

$$\frac{A_1 \vee \cdots \vee A_n}{A_1[\sigma] \vee \cdots \vee A_n[\sigma]}\ \textsf{INST}\ \sigma \qquad\qquad \frac{A_1 \vee \cdots \vee A_n}{A_{i_1} \vee \cdots \vee A_{i_m}}\ \textsf{FACTOR}$$

$$\frac{A_1 \vee \cdots \vee L \vee \cdots \vee A_m \qquad B_1 \vee \cdots \vee \neg L \vee \cdots \vee B_n}{A_1 \vee \cdots \vee A_m \vee B_1 \vee \cdots \vee B_n}\ \textsf{RESOLVE}\ L$$

Fig. 1. The Primitive Rules of Inference of Clausal First-Order Logic.

This is achieved by defining a logical kernel of ML functions that execute a primitive set of deduction rules on first-order clauses. For our purposes, we only need the five rules in Figure 1.

The AXIOM rule is used to create a new axiom of the logical system; it takes as argument the list of literals in the axiom clause. The ASSUME rule takes a literal L and returns the theorem $L \vee \neg L$.[5] The INST

[3] In principle we could use more combinators to guarantee an even more compact translation, but HOL goals are normally small enough that this extra complication is not worth the effort.

[4] Encoding type variables as first-order logic variables allows polymorphic types to be dealt with in a straightforward manner.

[5] This rule is used to keep track of reductions in the model elimination procedure.

rule takes a substitution σ and a theorem A, and applies the substitution to every literal in A.[6] The FACTOR rule takes a theorem and removes duplicate literals in the clause: note that no variable instantiation takes place here, two literals must be identical for one to be removed. Finally, the RESOLVE rule takes a literal L and two theorems A, B, and creates a theorem containing every literal except L from A and every literal except $\neg L$ from B. Again, no variable instantiation takes place here: only literals identical to L in A (or $\neg L$ in B) are removed.

These five primitive rules define a (refutation) complete proof system for clausal first-order logic. To see this, recall that a complete proof system results from FACTOR and RESOLVE rules that perform unification [6]. However, we can simulate these rules by first instantiating appropriately using the INST rule, and then applying our identical-match versions FACTOR and RESOLVE.

```
signature Kernel =
sig
  type formula = Term.formula
  type subst   = Term.subst

  (* An ABSTRACT type for theorems *)
  eqtype thm

  (* Destruction of theorems is fine *)
  val dest_thm : thm -> formula list

  (* But creation is only allowed by these primitive rules *)
  val AXIOM   : formula list -> thm
  val ASSUME  : formula -> thm
  val INST    : subst -> thm -> thm
  val FACTOR  : thm -> thm
  val RESOLVE : formula -> thm -> thm -> thm
end
```

Fig. 2. The ML Signature of a Logical Kernel Implementing Clausal First-Order Logic

The ML type system can be used to ensure that these primitive rules of inference represent the only way to create elements of an abstract `thm` type.[7] In Figure 2 we show the signature of an ML `Kernel` module that implements the logical kernel. We insist that the programmer of a first-order provers derive refutations by creating an empty clause of type `thm`. The only way to do this is to use the primitive rules of inference in the `Kernel` module: this is both easy and efficient for all the standard first-order proof procedures.

At this point it is simple to translate first-order refutations to HOL proofs. We add proof logs into the representation of theorems in the `Kernel`, so that each theorem remembers the primitive rule and theorems that were used to create it. When we complete a refutation, we therefore have a chain of proof steps starting at the empty clause and leading back to axioms. In addition, for each primitive rule of inference in `Kernel`, we create a higher-order logic version that works on HOL terms, substitutions and theorems. The final ingredient needed to translate a proof is a HOL theorem corresponding to each of the first-order axioms. These theorems are the HOL clauses in the CNF representation of the original (negated) goal, which we mapped to first-order logic and axiomatized.

To summarize: by requiring the programmer of a first-order proof procedure to derive refutations using a logical kernel, lifting these refutations to HOL proofs can be done completely automatically.

[6] In some presentations of logic, this uniform instantiation of variables in a theorem is called specialization.

[7] Indeed, the ability to define an abstract theorem type was the original reason that the ML type system was created.

2.3 The Scope of a First-Order Prover for HOL

Using the mapping in Section 2.1, we can use a first-order prover to prove some higher-order HOL goals, such as the classic derivation of an identity function from combinator theory:

$$\vdash (\forall x, y.\ \mathsf{K}\ x\ y = x) \land (\forall f, g, x.\ \mathsf{S}\ f\ g\ x = (f\ x)\ (g\ x)) \Rightarrow \exists f.\ \forall x.\ f\ x = x$$

Similarly, the framework for translating refutations in Section 2.2 is general enough to translate any first-order theorem to HOL. Therefore, we can use a first-order prover to solve for HOL terms satisfying a set of HOL formulas, just as Prolog does for Horn formulas.[8]

However, our method of embedding higher-order in first-order logic is not without danger. The whole reason for adding types to higher-order logic is to avoid the Russell paradox, and so if we choose to remove them in our translation we must beware of unsoundness. Defining a 'Russell combinator' R as

$$R = (\lambda x.\ \neg(x\ x)) = \mathsf{S}\ (\mathsf{K}\ (\neg))\ (\mathsf{S}\ \mathsf{I}\ \mathsf{I})$$

we find that we can use the reduction rules for S and K to prove

$$R\ R = \neg(R\ R)$$

and thus derive a contradiction.

Of course, a first-order refutation that is unsound in this way cannot be successfully translated to a HOL proof. It is therefore trivial to discover any problems that occur due to the lack of type information in the first-order representation. When unsoundness is discovered, the subgoal is simply tried again with the type information included. Fortunately, this phenomenon occurs in less than 1% of all HOL subgoals.

3 Problem Sets and Evaluation Methodology

In the next section we describe the ML implementation of our combination of first-order provers. To evaluate and compare different procedures, we use the following three problem sets:

TPTP This consists of all the problems classified as 'unsatisfiable' in version 2.4.1 of TPTP.[9]

uHOL This problem set consists of all subgoals proved by MESON when building: the standard theories included with version Kananaskis-0 of the HOL4 theorem prover; the HOL formalization of probability theory; and the example verification of the Miller-Rabin primality test.[10] The HOL subgoals are mapped to first-order logic without type information (uHOL = untyped HOL).

HOL The same problem set as uHOL, but the HOL subgoals are mapped to first-order logic with type information included.

Table 1. Profiles of the Problem Sets.

Set	N	C	L	S	L/C	S/C	S/L
TPTP	2752	31.0	65.0	229.0	2.07	8.17	4.00
uHOL	3679	11.0	19.0	146.0	1.78	12.86	7.14
HOL	3679	11.0	19.0	701.0	1.78	63.71	35.30

Table 1 profiles the three problem sets. For each problem set, we show the number of problems (**N**), and the median of several statistics for each problem: number of clauses (**C**), number of literals (**L**), number

[8] The model elimination procedure has the capability to solve for terms in this way.

[9] The TPTP problem set is available at `http://www.cs.miami.edu/~tptp/`.

[10] Available at `http://www.cl.cam.ac.uk/~jeh1004/research/problems/`.

of symbols[11] (**S**), mean literals per clause (**L/C**), mean symbols per clause (**S/C**), and mean symbols per literal (**S/L**).

Comparing the TPTP and HOL problem sets, it can be seen that the average TPTP problem has more clauses, while the average HOL problem has more symbols per literal. However, by looking at the uHOL row, it is apparent that most of the symbols in the HOL problems come from type information. One similarity between all three problem sets is the average number of literals per clause: around two.

As previously mentioned, we allow 60 seconds per TPTP problem and 5 seconds per HOL problem, to simulate the difference in requirements between fully automatic and interactive proof. All experiments were run on Athlon 1.4GHz processors with at least 512Mb of memory, using version 2.00 of Moscow ML[12] on RedHat Linux 7.1.

So that we can use statistical methods to compare the first-order provers, we randomly split each problem set into 10 equally sized sections. By counting the number of problems in each section that any two provers solve within the time limit, we can use the *t*-test to compute the statistical significance that one prover is better than the other [9]. Here is an example results table where we compare two hypothetical provers, **foo** and **bar**:

	foo	**bar**
foo	$*$	$\overset{+95}{99.5\%}$
bar	$\overset{+7}{—}$	$*$

Since we do not compare provers with themselves, the diagonal entries are marked with $*$. The 99.5% in the upper right entry means that **foo** is statistically better than **bar** with 99.5% confidence. The +95 above this means that, over the whole problem set, **foo** proved 95 problems that **bar** could not. The lower left entry in the table means that **bar** is *not* significantly better than **foo**, but it did prove 7 problems that **foo** could not.

4 Implementing the First-Order Provers

In Sections 2 and 3 we presented a mechanism for mapping HOL subgoals to first-order problems and a way to evaluate first-order provers. In this section we will describe an ML implementation of a collection of first-order proof procedures, using our evaluation method to select optimum parameters and justify optimizations.

Since the literature contains such an abundance of strategies and techniques for first-order proof, it was necessary to select just a few for the purpose of our present experiment. In particular, we do not treat equality at all, and add equality axioms as part of our mapping to first-order logic. However, in the future there is nothing to stop us including more sophisticated methods for handling equality, say by adding a `PARAMODULATION` primitive inference rule.

4.1 Model Elimination Procedure

The first proof procedure that we implement is the model elimination procedure of Loveland [17]; our prover is essentially a ground-up reimplementation of Harrison's MESON [13], incorporating some optimizations of Astrachan, Loveland and Stickel [2, 3].

Our strategy is to first produce a naive implemention, and then incrementally optimize it. The starting point is a version of model elimination called **m-0** that treats every input clause as an initial clause. A clause must contain at least one negative literal for **m-1** to treat it as initial, and initial clauses in **m-n** must contain all negative literals.

Building upon **m-n**, we add ancestor pruning to get **m-a**, and then ancestor cutting to get **m-x**. Ancestor cutting means that if the negation of an ancestor exactly matches the current goal, we do a reduction on that ancestor and disallow backtracking. Incorporating Harrison's divide-and-conquer search strategy brings

[11] By symbols we mean variables, functions, relations and logical connectives.

[12] Moscow ML is available at `http://www.dina.dk/~sestoft/mosml.html`.

us to **m-d**, and trying to match the goal from the clause set before trying unification is called **m-s**.[13] The optimization in **m-c** is slightly dubious, incorporating a limited form of caching to stop us attempting the same goal twice from a given point in the search. The overhead of this pays off on the TPTP problem set with a time limit of 60 seconds, but not on the HOL problem sets where the limit is 5 seconds.

Finally, **m-u** incorporates unit lemmaizing, where the use of a unit lemma contributes size 1 to the proof. Since we use iterative deepening to search for proofs in order of size, the penalty of using a unit lemma is an important factor in the optimization. However, we cannot make the penalty depend on the proof size of the unit lemma, since later we will import unit lemmas from a resolution prover where proof sizes do not play any part.

Table 2. Comparing Model Elimination Optimizations on the TPTP Problem Set.

	m-u	**m-c**	**m-s**	**m-d**	**m-x**	**m-a**	**m-n**	**m-1**	**m-0**
m-u	*	+89 99.5%	+91 99.5%	+99 99.5%	+180 99.5%	+186 99.5%	+198 99.5%	+345 99.5%	+371 99.5%
m-c	+13	*	+2 80.0%	+14 97.5%	+103 99.5%	+109 99.5%	+121 99.5%	+269 99.5%	+295 99.5%
m-s	+13	+0	*	+13 95.0%	+101 99.5%	+107 99.5%	+119 99.5%	+267 99.5%	+293 99.5%
m-d	+9	+0	+1	*	+90 99.5%	+96 99.5%	+108 99.5%	+255 99.5%	+281 99.5%
m-x	+12	+11	+11	+12	*	+7 95.0%	+25 99.5%	+178 99.5%	+204 99.5%
m-a	+12	+11	+11	+12	+1	*	+19 97.5%	+174 99.5%	+200 99.5%
m-n	+5	+4	+4	+5	+0	+0	*	+158 99.5%	+184 99.5%
m-1	+0	+0	+0	+0	+1	+3	+6	*	+29 99.5%
m-0	+0	+0	+0	+0	+1	+3	+6	+3	*

Table 2 shows the result of a pairwise comparison between each step in the evolution of our model elimination prover, from the humble **m-0** to the hi-tech **m-u** which proves 371 more TPTP problems. Similar results occur on the HOL problem sets.

4.2 Resolution Procedure

The second proof procedure that we implemented is the resolution procedure of Robinson [21]. Our version uses the given clause algorithm, and we implement term nets to improve the speed of unification and subsumption checking. Additionally, unit clauses are used whenever possible to simplify clauses. In contrast with the incremental sequence of optimizations that we used for model elimination, our resolution procedure has several independent parameters that control the search strategy.

The first parameter controls how much subsumption checking is done. By default, as clauses are taken from the unused list they are checked to see if they are subsumed by a clause in the used list. If so, they are dropped and the next clause is chosen from the unused list. We also implement a higher level of subsumption, indicated by an extra **s** in the prover name. Here, when we lift a clause from the unused list, we immediately see if it is subsumed by another clause in the unused list. If so, we use that clause instead.

The second parameter is a number n, and controls the order that we pick clauses from the unused list. For every clause picked from the unused list in FIFO order, we pick n clauses with the smallest symbol count.[14] The number n is one of **1**, **2**, **3**, **4** or **5**, and is part of the prover name. This is called the ratio strategy, originally used in the Otter theorem prover [26].

[13] The fact that we are implementing this in ML might help to explain why this is such an effective optimization. If matching succeeds then there is no need to update the substitution context, and this results in less allocation and reduced garbage collection times. These reductions range between 20% and 80% on TPTP problems.

[14] We efficiently implement this alternation in ML by storing unused clauses as both queues and (leftist) heaps. Okasaki [18] implements functional versions of these and many more data structures.

The final parameter is Robinson's positive refinement [20], which we indicate with a final '+' in the prover name.

We found that the best prover for all three problem sets is **r3**$^+$: the default level of subsumption; picking 3 smallest clauses for every clause at the head of the queue; and using positive resolution. On the TPTP problem set, this parameter setting is better than any other with confidence at least 95%.

4.3 Delta-style Procedure

The third and final proof procedure that we implemented is based on the Delta preprocessor of Schumann [23]. Put simply, for every n-ary relation R present in the problem, we generate the 'Delta goals' $R(X_1, \ldots, X_n)$ and $\neg R(Y_1, \ldots, Y_n)$ (with fresh variables X_i and Y_i). We then use the model elimination procedure with iterative deepening to search for solutions to the Delta goals. Every unit clause that is derived during this process is shared with the other proof procedures.

The Delta procedure takes the same optimization parameters as model elimination, and so it is not necessary to separately optimize this procedure. In any case, since it is not designed to directly solve the goal, but rather to help the other procedures by finding useful unit clauses, it only makes sense to use it when unit lemmaizing is switched on.

5 Combining the First-Order Provers

As already mentioned, when we run different proof procedures together they can cooperate by sharing unit clauses. Whenever a unit clause is derived, it is inserted into a global store that is available to every proof procedure. The way that the individual proof procedures make use of unit clauses was described in the previous sections.

Each proof procedure runs for a time-slice,[15] and a scheduler decides which proof procedure to run based on the cost of the execution time it has already consumed. For model elimination and resolution, the cost of execution time is simply the number of seconds, but for the Delta procedure it was empirically found to be better to use the square of the number of seconds.

For example, if each proof procedure has consumed 1/3 second of CPU time, the model elimination and resolution cost is 1/3, while the Delta cost is $(1/3)^2 = 1/9$. Therefore, Delta will be scheduled as the cheapest procedure. If each proof procedure has consumed 2 seconds, the model elimination and resolution cost is 2, while the Delta cost is $2^2 = 4$, and so one of model elimination and resolution will be scheduled to run for a time-slice.

Table 3. Comparing Combinations of Provers on the TPTP Problem Set.

	mrd	**mr**	**md**	**m**	**rd**	**r**
mrd	*	+22 99.5%	+160 99.5%	+200 99.5%	+291 99.5%	+322 99.5%
mr	+9 —	*	+161 99.5%	+189 99.5%	+283 99.5%	+307 99.5%
md	+22 —	+36 —	*	+56 99.5%	+274 99.5%	+298 99.5%
m	+13 —	+15 —	+7 —	*	+243 99.5%	+264 99.5%
rd	+25 —	+30 —	+146 —	+164 —	*	+42 99.5%
r	+25 —	+23 —	+139 —	+154 —	+11 —	*

Tables 3, 4, and 5 show the result of running different proof procedure combinations on the TPTP, uHOL and HOL problem sets, respectively.

[15] In our experiments we set each time slice to be 1/3 second long.

Table 4. Comparing Combinations of Provers on the uHOL Problem Set.

	mrd	**mr**	**md**	**m**	**rd**	**r**
mrd	*	+16 70.0%	+111 99.5%	+187 99.5%	+164 99.5%	+148 99.5%
mr	+13 —	*	+119 99.5%	+182 99.5%	+156 99.5%	+137 99.5%
md	+11 —	+22 —	*	+91 99.5%	+152 97.5%	+133 90.0%
m	+14 —	+12 —	+18 —	*	+133 —	+113 —
rd	+21 —	+16 —	+109 —	+163 90.0%	*	+33 —
r	+22 —	+14 —	+107 —	+160 99.0%	+50 90.0%	*

Table 5. Comparing Combinations of Provers on the HOL Problem Set.

	mrd	**mr**	**md**	**m**	**rd**	**r**
mrd	*	+9 —	+171 99.5%	+246 99.5%	+174 99.5%	+127 99.5%
mr	+22 95.0%	*	+184 99.5%	+258 99.5%	+184 99.5%	+131 99.5%
md	+25 —	+25 —	*	+102 99.5%	+168 75.0%	+124 —
m	+23 —	+22 —	+25 —	*	+146 —	+105 —
rd	+14 —	+11 —	+154 —	+209 99.5%	*	+20 —
r	+32 —	+23 —	+175 99.5%	+233 99.5%	+85 99.5%	*

In every case, the combined model elimination and resolution procedure performed significantly better than either individually, with the highest level of confidence (99.5%). For the TPTP problem set, we can use some arithmetic to see that there must exist at least 166 problems that were proved by the combined procedure but were not proved by either acting alone. This is compelling evidence that the combined procedure does more than simply harvest the problems that are 'shallow' for one of model elimination and resolution. Rather, the sharing of unit clauses creates a whole new procedure that is better than either.

Comparing the combinations of proof procedures across the three problem sets, as we move from TPTP through uHOL to HOL we find resolution becoming better relative to model elimination. Similarly, the Delta procedure helps the combined procedure less as we move from TPTP to HOL. This latter effect is probably due to the cost functions we chose, which favours Delta in the first 3 seconds of CPU time. When the total limit is 5 seconds, this represents a serious bias.

Because the proof procedures share unit clauses, some rather counter-intuitive effects can arise from combining proof procedures. For example, there is a class of 7 TPTP problems that model elimination can prove, but model elimination and Delta together fail to prove within 60 seconds. One of these is GRP128_4_003, which model elimination acting alone proves in about 12 seconds! The only explanation is that Delta finds some 'helpful' unit clauses that lead model elimination into an unprofitable area of the search space. However, these kind of events are rare: the other 6 problems in this class take model elimination acting alone more than 45 seconds to prove.

Finally, we can compare the performance of provers on the different versions of each problem in the uHOL and HOL problem sets. The best prover in this domain is the combination of model elimination and resolution, and there were 142 problems that it could prove in uHOL but not in HOL, and 13 problems that it could prove in HOL but not in uHOL. Therefore, this confirms our expectations that the much smaller versions of the problems in uHOL can be proved more efficiently, though there are a few examples where the types cut down the search space enough to make the difference between finding a proof within the time limit and not.

6 Transferring the Technology to Other Higher-Order Logic Theorem Provers

The results of the previous section show that we can create combination first-order proof tactics that are more powerful than individual proof procedures, and the LCF-style logical kernel allows us to easily translate first-order refutations to higher-order logic theorems. The existing implementation in HOL4 has been found to be a useful proof tool when used in proofs by the HOL developers.

The general architecture is easily ported to other higher-order logic theorem provers such as PVS or Isabelle. Indeed, the first-order proof procedures complete with logical kernel are a standalone library in Standard ML, so could be imported without any change at all.[16] The only part that needs to be created afresh for each higher-order logic theorem prover is a rule of inference that translates first-order refutations to higher-order logic theorems. This will differ slightly according to the particular details of the higher-order logic.

There is one difference between HOL and PVS that may be significant here. When translating a first-order refutation to higher-order logic, it is often necessary to translate a first-order term to higher-order logic. In HOL this is straightforward, but in PVS some auxiliary theorems may be necessary to establish the TCCs of the translated term. Additionally, there may be no type information in the first-order term, in which case we generate a HOL term and infer its principal type. In PVS this may not be possible in general, which may jeopardize the possibility of an untyped mapping to first-order logic. More investigation is necessary to establish whether the required information can be reconstructed in some way.

Finally, for some interactive theorem proving applications, it may be appropriate to simply trust that the subgoal is valid if the first-order prover detects unsatisfiability of the corresponding clauses. There is already strong pressure on the implementors of first-order provers to make sure their system is sound; at the annual CADE automatic system competition, provers face disqualification if they fail any soundness test. Abandoning the idea of translating refutations gives an additional benefit: many first-order provers perform much better if they are not required to keep track of the refutation as they search.

7 Conclusions

In this paper we described a framework for implementing first-order proof tactics in higher-order logic theorem provers, which uses an LCF-style logical kernel to create a modular interface between the two logics. The architecture we presented is not specific to a particular higher-order theorem prover, and we have sketched out how it could be ported to a new theorem prover, notwithstanding the potential problem with type reconstruction in PVS. We have implemented a version in HOL4, with two working mappings: one that preserves type information from higher-order logic, and one that reconstructs it while translating the first-order refutation.

We also implemented a combination of first-order proof procedures in Standard ML, and compared their performance on three different problem sets. Based on our experiments, we tentatively conclude that good performance from a first-order prover in one domain suggests that it will also perform well in other domains. Optimizations we made to improve performance on the TPTP problem set usually also improved performance on the HOL problem sets, though there was a significant shift from model elimination in the TPTP domain towards resolution in the HOL domain. This was extremely surprising, since the HOL problems are self-selected for the MESON prover in HOL. During interactive proof in HOL, if MESON cannot prove a subgoal within a reasonable time, then a user can perform a manual inference step and then try again. Further investigation is needed to establish why resolution seems to do better on HOL subgoals.

We found the LCF-style kernel for clausal first-order logic to be more than just a convenient interface to a proof translator. Reducing the steps of proof procedures to primitive inferences clarified their behaviour, and also helped catch bugs early. Also, assuming the (52 line) ML `Kernel` module is correctly implemented and the programmer is careful about asserting axioms, loss of soundness arising from 'prover optimizations' can be completely avoided.

Finally, on all three problem sets the combination of model elimination and resolution was found to perform significantly better than either individually. This supports the hypothesis that first-order search

[16] Available at `http://www.cl.cam.ac.uk/~jeh1004/research/metis/`.

spaces have a structure that rewards the use of a variety of search methods, despite the extra redundancy that is entailed; a view neatly summarized by Astrachan and Loveland [2]:

> "Unlike chess, theorems are a very diverse lot and different proof methods may excel in different areas."

8 Related Work

In addition to MESON in HOL, there are many other examples of automatic first-order provers being used to prove problems in an interactive theorem prover: (in chronological order) FAUST in HOL [16]; SEDUCT in LAMBDA [5]; 3TAP in KIV [1]; Paulson's `blast` in Isabelle [19]; Gandalf in HOL [14]; and Bliksem in Coq [4]. Various mappings are used from the theorem prover subgoals into problems of first-order logic, defining the scope of what can be automatically proved. Using the architecture presented in this paper for translating first-order refutations would allow different first-order provers to be 'plugged-in' to the theorem prover. Moreover, if first-order provers emitted proofs in a standardized 'LCF-style' logical kernel for clausal first-order logic, then this would further simplify their integration into interactive theorem provers.

As part of the ILF Mathematical Library Project, Dahn and Wernhard [8] extracted 97 first-order problems from the article *Boolean Properties of Sets* in the MIZAR Mathematical Library. Later, Dahn [7] added the ability to represent MIZAR type information, and extracted 47 problems from the article *Relations Defined on Sets*. However, there has been no published study of the comparitive effectiveness of first-order provers on this problem set.

Several projects have aimed to create combination first-order provers that are better than the individual components. For example, the TECHS system [10] uses automatic referees to decide which clauses to exchange between provers, as opposed to our system that simply shares unit clauses. Further investigation is needed to decide the best way of combining proof procedures in our application.

Finally, we note that Robinson [22] proposed a version of higher-order logic in terms of combinators (though it is typeless and therefore unsound due to the 'Russell combinator' we defined in Section 2.3).[17] However, the motivation behind combinators instead of the λ-calculus is that proof automation can be simplified, and this also motivates our combinator mapping from HOL subgoals to first-order logic.

Acknowledgements

This project greatly benefitted from discussions with Mike Gordon, Michael Norrish and Konrad Slind. Also, the CAML code in John Harrison's forthcoming book provided a useful starting point for my own ML implementation of first-order provers. Thanks are also due to Judita Preiss for technical support.

References

1. Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Wolfram Menzel, Wolfgang Reif, Gerhard Schellhorn, and Peter H. Schmitt. Integration of automated and interactive theorem proving. In W. Bibel and P. Schmitt, editors, *Automated Deduction: A Basis for Applications*, volume II, chapter 4, pages 97–116. Kluwer, 1998.
2. O. L. Astrachan and Donald W. Loveland. The use of lemmas in the model elimination procedure. *Journal of Automated Reasoning*, 19(1):117–141, August 1997.
3. Owen L. Astrachan and Mark E. Stickel. Caching and lemmaizing in model elimination theorem provers. In Deepak Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction (CADE-11)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 224–238, Saratoga Springs, NY, USA, June 1992. Springer.
4. Marc Bezem, Dimitri Hendriks, and Hans de Nivelle. Automated proof construction in type theory using resolution. In David A. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*, volume 1831 of *Lecture Notes in Computer Science*, pages 148–163, Pittsburgh, PA, USA, June 2000. Springer.

[17] Thanks to John Harrison for drawing my attention to this.

5. H. Busch. First-order automation for higher-order-logic theorem proving. In Tom Melham and Juanito Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications, 7th International Workshop*, volume 859 of *Lecture Notes in Computer Science*, Valletta, Malta, September 1994. Springer.
6. Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York, 1973.
7. Ingo Dahn. Interpretation of a Mizar-like logic in first-order logic. In Ricardo Caferra and Gernot Salzer, editors, *International Workshop on First-Order Theorem Proving (FTP '98)*, Technical Report E1852-GS-981, pages 116–126, Vienna, Austria, November 1998. Technische Universität Wien.
8. Ingo Dahn and Christoph Wernhard. First order proof problems extracted from an article in the MIZAR mathematical library. In Maria Paola Bonacina and Ulrich Furbach, editors, *International Workshop on First-Order Theorem Proving (FTP '97)*, number 97-50 in RISC-Linz Report Series, Schloss Hagenberg, Austria, October 1997. Johannes Kepler Universität Linz.
9. Morris DeGroot. *Probability and Statistics.* Addison-Wesley, 2nd edition, 1989.
10. Jörg Denzinger and Dirk Fuchs. Knowledge-based cooperation between theorem provers by TECHS. SEKI-Report SR-97-11, University of Kaiserslautern, 1997.
11. M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science.* Springer, 1979.
12. M. J. C. Gordon and T. F. Melham. *Introduction to HOL (A theorem-proving environment for higher order logic).* Cambridge University Press, 1993.
13. John Harrison. Optimizing proof search in model elimination. In Michael A. McRobbie and John K. Slaney, editors, *13th International Conference on Automated Deduction (CADE-13)*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 313–327, New Brunswick, NJ, USA, July 1996. Springer.
14. Joe Hurd. Integrating Gandalf and HOL. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*, pages 311–321, Nice, France, September 1999. Springer.
15. Joe Hurd. An LCF-style interface between HOL and first-order logic. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 134–138, Copenhagen, Denmark, July 2002. Springer.
16. R. Kumar, T. Kropf, and K. Schneider. Integrating a first-order automatic prover in the HOL environment. In Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications (HOL '91), August 1991*, pages 170–176, Davis, CA, USA, 1992. IEEE Computer Society Press.
17. Donald W. Loveland. Mechanical theorem proving by model elimination. *Journal of the ACM*, 15(2):236–251, April 1968.
18. Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, Cambridge, UK, 1998.
19. L. C. Paulson. A generic tableau prover and its integration with Isabelle. *Journal of Universal Computer Science*, 5(3), March 1999.
20. J. A. Robinson. Automatic deduction with hyper-resolution. *International Journal of Computer Mathematics*, 1:227–234, 1965.
21. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–49, January 1965.
22. J. A. Robinson. A note on mechanizing higher order logic. *Machine Intelligence*, 5:121–135, 1970.
23. Johann Ph. Schumann. DELTA — A bottom-up processor for top-down theorem provers (system abstract). In Alan Bundy, editor, *12th International Conference on Automated Deduction (CADE-12)*, volume 814 of *Lecture Notes in Artificial Intelligence*, Nancy, France, June 1994. Springer.
24. Christian B. Suttner and Geoff Sutcliffe. The TPTP problem library — v2.1.0. Technical Report JCU-CS-97/8, Department of Computer Science, James Cook University, December 1997.
25. D. A. Turner. A new implementation technique for applicative languages. *Software – Practice and Experience*, 9(1):31–49, January 1979.
26. Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications.* McGraw-Hill, New York, 2nd edition, 1992.

Coq Tacticals and PVS Strategies: A Small Step Semantics *

Florent Kirchner

École Normale Supérieure de Cachan, France
florent.kirchner@inria.fr ; fkirchne@nianet.org

Abstract. The need for a small step semantics and more generally for a thorough documentation and understanding of Coq's tacticals and PVS's strategies arise with their growing use and the progressive uncovering of their subtleties. The purpose of the following study is to provide a simple and clear formal framework to describe their detailed semantics, and highlight their differences and similarities.

1 Introduction

Procedural proof languages are used to prove propositions with the assistance of a proof engine: the user wields the language to give the theorem prover instructions or *tactics* on the way to proceed throughout the proof. The instruction set roughly corresponds to the elementary steps of the formal logic inherent to the prover; a *proof script* is a collection of such instructions. The need for a way to express the proof scripts in a more sophisticated and factorized way emerges as soon as proofs get more complicated, resulting in very large proof scripts of elementary steps. This makes any proof reading or maintenance operation tedious if not impossible. Both Coq [1] and PVS [11], derived from the LCF theorem prover, introduce proof combinators in their proof language to powerfully compose elementary proof tactics: *tacticals* in Coq, *strategies* in PVS[1]. Though other provers such as Isabelle and NuPrl also implement tacticals, they have not been included in this work but a similar reasonning could probably apply. The following sections expose the semantics of the tacticals of Coq and PVS, using a small steps semantics and some appropriate structures and notations.

2 Conventions and Structures

Coq and PVS, as most procedural theorem provers, usually implement a goal oriented proof style. That is, given a proof goal and an elementary logical rule, the prover applies the logical rule backwards to the goal, yielding a set of potentially simpler subgoals. For example, given the proof goal $\Gamma \vdash 0 \leq X \wedge X \leq 1$, the Coq instruction *Intro* (`(split)` in PVS) generates the subgoals $\Gamma \vdash 0 \leq X$ and $\Gamma \vdash X \leq 1$. This corresponds to the application of the logical rule:

$$\frac{A \vdash B \qquad A \vdash C}{A \vdash B \wedge C}\wedge\text{-intro} .$$

In turn, some new rules are applied to the new subgoals, and the process stops when all the subgoals are refined enough to be trivially proven true. This repetition creates an arborescent structure of subgoals, which is called here the *proof context.* Goals, i.e., sets of formulas of the form $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$, are commonly named *sequents.*

2.1 The Proof Context

The proof context is considered here as a collection of sequents organized in a tree of sequents, its leaves representing the sequents that are currently to be proven. A leaf, when modified by some command, becomes the parent of the sequents created by this command: the nodes of the tree of sequents are the "old" sequents.

* This work was supported by INRIA FUTURS and the National Institute of Aerospace (under NASA Cooperative Agreement NCC-1-02043).

[1] Henceforth, when refering to the combinators in general, the name *tactical* will be used.

Thus, the tree of sequents keeps track of the proof progression. Incidentally, one has to consider the number of features that are related to the proof context (state of the proof, proven branches, goal numbering, etc.). Hence the semantics is made much clearer by blending a simplified object-oriented structure with the tree representation. This way, the proof context, the sequents, and the formulas are considered as non mutable objects including *attributes*, which correspond to their features, and functions or *methods* that read or modify these attributes and eventually return a new object. For instance, one of the attributes of the proof context object is the tree of sequent objects. Furthermore, a sequent object has a set of formula objects as attribute.

Let us now define some notations. A sequent is represented as $\Gamma \vdash \Delta$, where Γ is the *antecedent* and Δ is the *consequent*, each being a list of formulas[2]. Latin letters A, B, etc. represent individual formulas. We write $O.\,m(\bar{x})$ for the invocation of the method m of object O with the list of parameters $\bar{x}$. The objects here are non mutable, meaning that methods modifying an object return a new object. Thus, a method call $O.\,m(\bar{x})$ is a synonym for the function call $m(\bar{x}, O)$, and the objects could also be seen as records. The letter τ denotes a proof context object; we distinguish a few particular proof contexts:

- $\top$ is a proof context that is completely proven.
- $\bot_n$ stands for a failed proof context. The integer n codes for an "error level", i.e., an indicator of the propagation range of the error. Errors are raised by tacticals and tactics, when they are called in an inappropriate situation (i.e., when none of the reduction rules of our semantics apply[3]).
- And $\varnothing$ is the empty proof context, i.e., a proof context object hosting an empty tree.

The equality test between a context and an empty, proven or failed context is the only equality test between contexts we authorize in our semantics.

The description of the attributes and methods of τ is as follows.

- Attributes:
  - $\tau.$ seq_tree: the tree of sequents.
  - $\tau.$ active: pointer to the active subtree of sequents, i.e., the subtree on which the next command will take effect. In case it is a leaf, then $\tau.$ active represents a sequent $\Gamma \vdash \Delta$, and we will write: $\tau.\ \ \Gamma \vdash \Delta$ to refer to such a proof context.
  - $\tau.$ progress: this is a flag raised when the tree of sequents has gone through changes. Basically, when a tactic successfully applies, it raises the progress flag ; it is reseted by a specific, "Break", command.
- Methods:
  - $\tau.$ addLeaves($\Gamma_1 \vdash \Delta_1, \dots, \Gamma_n \vdash \Delta_n$): this method applies when the *active* attribute points to a leaf: it adds n leaves to the tree. In the new tree, the new sequents $\Gamma_i \vdash \Delta_i$, $i \in \{1, \dots, n\}$, will be leaves, and the former active leaf of the old tree will become their common parent node.
  - $\tau.$ lowerPointer(i): moves the active pointer down (towards the root) in the tree, $i \geq 0$ being the depth of the move.
  - $\tau.$ raisePointerToLeaf(): moves the pointer up to the first (i.e., innermost leftmost) unproven leaf of the tree.
  - $\tau.$ pointNextSibling(): moves the pointer to the closest unproven leaf, sibling of the active sequent. If there is no such sibling, the pointer is set to a default empty value, which is represented by the method returning the empty proof context $\varnothing$.
  - $\tau.$ setProgress(b): sets the corresponding flag to b.
  - $\tau.$ hasProgressed(): returns the value of the progress flag.
  - $\tau.$ setLeafProven(): the active leaves, that is, the leaves of the active subtree, are labeled as proven. If there are no unproven sequents left, the proof is finished (i.e., $\tau.$ setLeafProven() $= \top$).
  - $\tau.$ isActiveTreeProven(): returns true if all the leaves in the active subtree are labeled as proven, false otherwise.

[2] The semantics presented in this paper does not distinguish between sequents with permuted formulas. This limitation is not problematic since we focus on tacticals, which do not require formula-level knowledge. But it should be addressed if a detailed semantics of the tactics, in addition to the semantics of tacticals, was to be considered.

[3] The error system is a bit more complicated than this, especially in Coq. But this simplification is a valid, understandable approximation of the provers' behaviour.

The sequent and formula objects are illustrated in Fig. 1, which also provides some type information. The figure uses the UML formalism where a class notation is a rectangle divided into three parts: class name, attributes, and methods. The diamond end arrow represents an aggregation, that is, a relation "is part of". The types presented here are basic and purely informative.

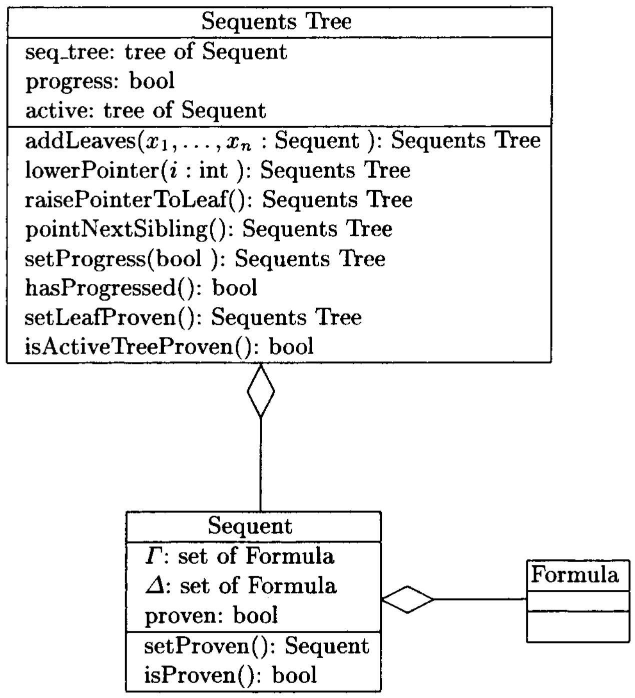
Sequents Tree
seq_tree: tree of Sequent
progress: bool
active: tree of Sequent
addLeaves($x_1, \ldots, x_n$: Sequent): Sequents Tree
lowerPointer(i : int): Sequents Tree
raisePointerToLeaf(): Sequents Tree
pointNextSibling(): Sequents Tree
setProgress(bool): Sequents Tree
hasProgressed(): bool
setLeafProven(): Sequents Tree
isActiveTreeProven(): bool
Sequent
Γ: set of Formula
Δ: set of Formula
proven: bool
setProven(): Sequent
isProven(): bool
Formula

Fig. 1. Proof context objects

2.2 The Proof Script

Given a set of tactics and of tacticals, a proof script is built by combining tactics with tacticals. For instance, in Coq, with the *Intro* and *Assumption* tactics, and the tactical ";", one can build the proof script *Intro ; Assumption*. Such a proof script applies to a proof context τ. We use p, p' to designate tactics and e, e' to denote proof scripts.

The distinction between tactics and tacticals within the proof language is somewhat fuzzy, as they both modify the proof context object. Here we consider that the tactics are the elements of the proof language that attempt to modify the tree of sequents, by adding leaves to it. For example, in PVS, the `(split)` tactic applied to the sequent $A \vdash B \wedge C$ behaves as the $\wedge$-intro logical rule, adding two leaves $A \vdash B$ and $A \vdash C$ to the sequent tree. Thus the sequent tree

$$\frac{A \vdash B \wedge C}{\vdots}$$

is transformed into the sequent tree

$$\cfrac{\cfrac{A \vdash B \qquad A \vdash C}{A \vdash B \wedge C}}{\vdots} \;.$$

The tacticals represent the proof language's control structures. In our semantics, they do not modify the tree of sequents directly but rather reduce into simpler proof scripts, and possibly modify some other attributes of the proof context. For instance in PVS, assuming a non-failed non-proven context τ, the proof script `(if` *nil* `(fail) (split))`, formed of the tactical `if` and the two tactics `(split)` and `(fail)`, evaluates in the `(split)` tactic:

$$\texttt{(if } \mathit{nil} \texttt{ (fail) (split))} \;/\, \tau \xrightarrow{\epsilon} \texttt{(split)} \;/\, \tau \ .$$

The actual modification of the proof context is performed by `(split)`.

In these examples the difference between tactics and tacticals appears quite clear, but we also note that the definition of a tactical implies the manipulation of tactics. Because of this dependency, the presentation of the semantics of the tacticals needs to be parameterized by the computation rule for tactics.

3 The Semantic Framework

The notion of *small step* or *reduction* semantics was introduced by Plotkin [9] in 1981. It consists in a set of rewriting rules specifying the elementary steps of the computation, within a context. The idea behind the present formalism is to use the reduction semantics of the imperative part of Objective ML, popularized by Wright and Felleisen [12], as an inspiration to deal with the interactions between the proof language and the proof context.

As exposed in the previous section, the reduction rules for the tacticals are dependent on the way tactics are applied to proof contexts. The semantics of the tacticals is parametrized by that of the tactics. Hence a formal definition of a tactic application is needed before any semantic rules are given. Since tactics, when evaluated, modify the tree of sequents, we consider them as expressions which modify the proof context. A tactic p applied to a proof context τ returns another proof context τ':

$$p\%\tau = \tau' \ .$$

The exact instanciations of this functional definition are of course system specific, and will be exposed in sections 4 and 5.

Tacticals are combinators, therefore their evaluation within a proof script should return either a simpler proof script or a tactic. We denote this returned expression by e'. The reduction of tacticals can also modify the proof context τ, thus a reduction rule in our semantics will look like:

$$e \;/\, \tau \xrightarrow{\epsilon} e' \;/\, \tau' \ ,$$

where ϵ denotes a head reduction (i.e., reduction of the head redex). These rules are conditionnal rewriting rules, with the tactics' computation function as a possible parameter. For example, the Coq tactical ";" applies its first argument to the current goal and then its second argument to all the subgoals generated. If the first argument proves the current goal or fails, applying another proof script to that failed or proven proof context does not make any sense, and the second argument is neglected :

$$\begin{aligned} v_1 \ ; e_2 \;/\, \tau \xrightarrow{\epsilon} e_2 \;/\, (v_1\%\tau) \quad & \text{if } \forall n \ (v_1\%\tau) \neq \bot_n \\ & \text{and } \neg(v_1\%\tau).\text{isActiveTreeProven}() \ , \\ v_1 \ ; e_2 \;/\, \tau \xrightarrow{\epsilon} v_1 \;/\, \tau \quad & \text{if } \exists n \ (v_1\%\tau) = \bot_n \\ & \text{or } (v_1\%\tau).\text{isActiveTreeProven}() \ . \end{aligned}$$

The context rule

$$\frac{e \;/\, \tau \xrightarrow{\epsilon} e' \;/\, \tau'}{E[e] \;/\, \tau \longrightarrow E[e'] \;/\, \tau'}$$

allows processing a proof script on which no head reduction applies. The definitions of the detailed reduction rules as well as that of the grammar of the context E depend highly on the language, and will be presented in the later prover-specific sections.

Finally, the values of our semantics consist, for each language, in the set of its components we do not wish to reduce. Thus they will be defined as the subset of each languages that are tactics, augmented, in the case of Coq, by the recursively defined functional and recursive operations (see section 4.2).

Note that this definition of the reduction semantics of the tacticals produces, when all tacticals have been reduced, something like $v \;/\; \tau$ as a final result. This is unsatisfying since we would like to see this final tactic v applied to τ (as in $v\%\tau$). Hence the use, for each langage, of a "Break" command that does this final evaluation.

4 Coq

In Coq the tactical commands are defined as an independent language, called $\mathcal{L}_{tac}$[4]. Delahaye [4] gives the definition of this language and an informal big step semantics[5].

4.1 Syntax

Let us define the syntax of a Coq proof script:

$$e \quad ::= \quad expr. \qquad\qquad \text{all expressions must end with "." .}$$

And

$$
\begin{array}{rcll}
expr & ::= & x & \text{identifiers} \\
 & | & p & \text{tactic} \\
 & | & k & \text{integer } (\mathcal{L}_{tac}\text{-specific}) \\
 & | & t & \text{Coq term} \\
 & | & \text{Fun } x \rightarrow e & \\
 & | & \text{Rec } x_1\ x_2 \rightarrow e & \\
 & | & (e_1\ e_2) & \\
 & | & \text{Let } x_1 = e_1 \text{ And } \ldots \text{And } x_n = e_n \text{ In } e & \\
 & | & \text{Match } t \text{ With } ([t_i] \rightarrow e_i)_{i=1}^{n} & \\
 & | & \text{Match Context With } ([hp_i \vdash p_i] \rightarrow e_i)_{i=1}^{n} & \\
 & | & e_1 \text{ Orelse } e_2 & \\
 & | & \text{Do } k\ e & \\
 & | & \text{Repeat } e & \\
 & | & \text{Try } e & \\
 & | & \text{Progress } e & \\
 & | & \text{First } [e_1 | \ldots | e_n] & \\
 & | & \text{Solve } [e_1 | \ldots | e_n] & \\
 & | & \text{Tactic Definition } x\ e & \\
 & | & \text{Meta Definition } x\ e & \\
 & | & \text{Recursive Tactic Definition } x\ e & \\
 & | & \text{Recursive Meta Definition } x\ e & \\
 & | & e_1\ ;\ e_2 & \\
 & | & e_0\ ;\ [e_1 | \ldots | e_n] \quad . &
\end{array}
$$

4.2 Semantics

The values of the semantics are defined as:

$$
\begin{array}{rcl}
v & ::= & p \\
 & | & \text{Fun } x \rightarrow e \\
 & | & \text{Rec } x_1\ x_2 \rightarrow e \quad .
\end{array}
$$

[4] $\mathcal{L}_{tac}$ also includes some commands that correspond to our definition of tactics, which we will see later; and some miscellaneous features that will not be presented in this paper.

[5] Whereas a small step semantics is defined by a set of reduction rules that apply within a reduction context, a big step semantics directly links an expression with its normal form.

The reduction rules for the tacticals follow.

Applications These simply correspond to the β-reduction rules of the λ-calculus.

$$(\text{Fun } x \twoheadrightarrow e)(v) \;/\; \tau \xrightarrow{\epsilon} e[x \hookleftarrow v] \;/\; \tau \;.$$
$$(\text{Rec f } x \twoheadrightarrow e)(v) \;/\; \tau \xrightarrow{\epsilon} e[x \hookleftarrow v][\text{f} \hookleftarrow (\text{Rec f } x \twoheadrightarrow e)] \;/\; \tau \;.$$

Local variable binding The x_i are bound to the values v_i in the expression e. The bindings are not mutually dependent.

$$\text{Let } x_1 = v_1 \text{ And} \ldots \text{And } x_n = v_n \text{ In } e \;/\; \tau \xrightarrow{\epsilon} e[x_1 \hookleftarrow v_1, \ldots, x_n \hookleftarrow v_n] \;/\; \tau \;.$$

Term matching This tactical matches a Coq term with a series of patterns, and return the appropriate expression, properly instanciated.
Let $\oplus$ be the binary operator defined as:

$$\begin{aligned} \sigma_1 e_1 \oplus \sigma_2 e_2 \;/\; \tau &\longrightarrow v_1 \;/\; \tau && \text{if the substitution } \sigma_1 \text{ is defined} \\ & && \text{and } \sigma_1 e_1 \;/\; \tau \text{ evaluates in } v_1; \\ &\longrightarrow v_2 \;/\; \tau && \text{else, if } \sigma_2 \text{ is defined} \\ & && \text{and } \sigma_2 e_2 \;/\; \tau \text{ evaluates in } v_2; \\ &\longrightarrow \text{Idtac} \;/\; \tau && \text{otherwise.} \end{aligned}$$

For all $i \in \{1, \ldots, n\}$, $\sigma_{p_i \hookleftarrow t}$ is the substitution resulting from the matching of t by p_i (undefined if p_i does not match t ; matching by _ always succeds and yields the empty substitution).
The reduction rule then is:

$$\text{Match } t \text{ With } ([p_i] \twoheadrightarrow e_i)_{i=1}^{n} \;/\; \tau \xrightarrow{\epsilon} \bigoplus_{i=1}^{n} \sigma_{p_i \hookleftarrow t}\, e_i \;/\; \tau \;.$$

Context matching This tactical matches the current goal with a series of patterns, and returns the appropriate expression, properly instanciated. The order of the patterns is not significant ; since Coq uses constructive logic, the consequent Δ is limited to a single formula B.
The original Coq rule allows for multiple antecedent patterns, which is a simple nesting of the presented form:

$$\begin{aligned} &\text{Match Context With } ([hp_i \vdash p_i] \twoheadrightarrow e_i)_{i=1}^{n} \;/\; \tau.(\ldots A_j \ldots \vdash B) \xrightarrow{\epsilon} \\ &\quad \bigoplus_{i=1}^{n} \sigma_{hp_i \hookleftarrow A_j} \sigma'_{p_i \hookleftarrow B}\, e_i \;/\; \tau \;. \end{aligned}$$

If this does not succeed then the context progression rule is used instead:

$$\begin{aligned} &\text{Match Context With } ([hp_i \vdash p_i] \twoheadrightarrow e_i)_{i=1}^{n} \;/\; \tau.(\ldots A_j \ldots \vdash B) \xrightarrow{\epsilon} \\ &\text{Match Context With } ([hp_i \vdash p_i] \twoheadrightarrow e_i)_{i=1}^{n} \;/\; \tau.(\ldots A_{j-1} \ldots \vdash B) \;. \end{aligned}$$

Break The break command '.' triggers the evaluation of the tactics and then resets some parameters in the proof context before the application of the next proof script:

$$v. \;/\; \tau \xrightarrow{\epsilon} (v\%\tau).\,\text{raisePointerToLeaf}().\,\text{setProgress}(\text{false}) \;.$$

Sequence The sequential application of two tactics: v_2 is applied to all the subgoals generated by v_1. This is the basic example of the use of conditional rules in conjunction with the % relation.

$$\begin{aligned} v_1 \;;\; e_2 \;/\; \tau \xrightarrow{\epsilon} e_2 \;/\; (v_1\%\tau) \quad & \text{if } \forall n \geq 0\ (v_1\%\tau) \neq \bot_n \\ & \text{and } \neg(v_1\%\tau).\,\text{isActiveTreeProven}() \;, \\ v_1 \;;\; e_2 \;/\; \tau \xrightarrow{\epsilon} v_1 \;/\; \tau \quad & \text{if } \exists n \geq 0\ (v_1\%\tau) = \bot_n \\ & \text{or } (v_1\%\tau).\,\text{isActiveTreeProven}() \;. \end{aligned}$$

N-ary sequence First applies v_0 and then each of the v_i to one of the subgoals generated. The definition of this command uses an additional operator, $\overline{;_\mathcal{T}}$, to allow potential backtracking.

$$\begin{array}{ll}
v_0; [e_1| \dots |e_n] \;/\; \tau \xrightarrow{\epsilon} \overline{;_\mathcal{T}} e_1, \dots, e_n \;/\; (v_0\%\tau).\,\mathrm{raisePointerToLeaf}() & \\
\qquad \text{if } \forall n \geq 0 \; (v_0\%\tau) \neq \bot_n & \\
\qquad \text{and } \neg(v_0\%\tau).\,\mathrm{isActiveTreeProven}() & \\
v_0; [e_1| \dots |e_n] \;/\; \tau \xrightarrow{\epsilon} v_0 \;/\; \tau & \\
\qquad \text{if } \exists n \geq 0 \; (v_0\%\tau) = \bot_n & \\
\qquad \text{or } (v_0\%\tau).\,\mathrm{isActiveTreeProven}() \;, &
\end{array}$$

and

$$\begin{array}{ll}
\overline{;_\mathcal{T}} v_1, e_2 \dots, e_n \;/\; \tau' \xrightarrow{\epsilon} \overline{;_\mathcal{T}} e_2, \dots, e_n \;/\; (v_1\%\tau').\,\mathrm{pointNextSibling}() & \\
\qquad \text{if } \forall n \geq 0 \; (v_1\%\tau') \neq \bot_n \;, & \\
\overline{;_\mathcal{T}} v_1, e_2 \dots, e_n \;/\; \tau' \xrightarrow{\epsilon} (\mathrm{Fail}\ 0) \;/\; \tau & \text{if } \exists n \geq 0 \; (v_1\%\tau') = \bot_n \;, \\
\overline{;_\mathcal{T}} v_n \;/\; \tau' \xrightarrow{\epsilon} (\mathrm{Fail}\ 0) \;/\; \tau & \text{if } \tau' = \varnothing \\
& \text{or } (v_n\%\tau').\,\mathrm{pointNextSibling}() \neq \varnothing \;, \\
\overline{;_\mathcal{T}} v_n \;/\; \tau' \xrightarrow{\epsilon} \mathrm{Idtac} \;/\; (v_n\%\tau').\,\mathrm{lowerPointer}(1) & \text{if } \tau' \neq \varnothing \\
& \text{and } (v_n\%\tau').\,\mathrm{pointNextSibling}() = \varnothing \;.
\end{array}$$

Branching This tactical tests whether the application of v_1 fails or does not progress, in which case it applies v_2.

$$\begin{array}{ll}
v_1 \text{ Orelse } e_2 \;/\; \tau \xrightarrow{\epsilon} e_2 \;/\; \tau & \text{if } (v_1\%\tau) = \bot_n \\
& \text{or } \neg(v_1\%\tau).\,\mathrm{hasProgressed}() \;, \\
v_1 \text{ Orelse } e_2 \;/\; \tau \xrightarrow{\epsilon} v_1 \;/\; \tau & \text{if } (v_1\%\tau) = \neq \bot_n \\
& \text{and } (v_1\%\tau).\,\mathrm{hasProgressed}() \;.
\end{array}$$

Progression The progression test. Fails if its argument does not make any change to the current proof context.

$$\begin{array}{ll}
\text{Progress } v \;/\; \tau \xrightarrow{\epsilon} v \;/\; \tau & \text{if } (v\%\tau).\,\mathrm{hasProgressed}() \;, \\
\text{Progress } v \;/\; \tau \xrightarrow{\epsilon} (\mathrm{Fail}\ 0) \;/\; \tau & \text{if } \neg(v\%\tau).\,\mathrm{hasProgressed}() \;.
\end{array}$$

Iteration Here k is a primitive integer, only used in $\mathcal{L}_{tac}$. This tactical repeats v, k times, along all the branches of the sequent subtree. Here again we introduce an additional operator $\overline{\mathrm{Do}_e}$.

$$\mathrm{Do}\ k\ e \;/\; \tau \xrightarrow{\epsilon} (\overline{\mathrm{Do}_e}\ k\ e) \;/\; \tau \;,$$

with

$$\begin{array}{ll}
\overline{\mathrm{Do}}\ 0\ e \;/\; \tau \xrightarrow{\epsilon} \mathrm{Idtac} \;/\; \tau & \\
(\overline{\mathrm{Do}_e}\ k\ v) \;/\; \tau \xrightarrow{\epsilon} (\overline{\mathrm{Do}_e}\ (k-1)\ e) \;/\; (v\%\tau) & \\
& \text{if } \forall n \geq 0 \; (v\%\tau) \neq \bot_n \\
& \text{and } \neg(v\%\tau).\,\mathrm{isActiveTreeProven}() \\
(\overline{\mathrm{Do}_e}\ k\ v) \;/\; \tau \xrightarrow{\epsilon} v \;/\; \tau & \text{if } \exists n \geq 0 \; (v\%\tau) = \bot_n \\
& \text{or } (v\%\tau).\,\mathrm{isActiveTreeProven}() \;.
\end{array}$$

Indefinite iteration This is the indefinite version of the previous iteration. It stops when all the applications of v fail. As for the previous finite iteration, notice the additional operator $\overline{\text{Repeat}_e}$.

$$\text{Repeat } e \;/\; \tau \xrightarrow{\epsilon} \overline{\text{Repeat}_e}\; e \;/\; \tau \ ,$$

with

$$\begin{array}{ll} \overline{\text{Repeat}_e}\; v \;/\; \tau \xrightarrow{\epsilon} \text{Idtac} \;/\; \tau & \text{if } \exists n \geq 0\ (v\%\tau) = \bot_n \\ \overline{\text{Repeat}_e}\; v \;/\; \tau \xrightarrow{\epsilon} v \;/\; \tau & \text{if } (v\%\tau).\text{isActiveTreeProven}() \\ \overline{\text{Repeat}_e}\; v \;/\; \tau \xrightarrow{\epsilon} \overline{\text{Repeat}_e}\; e \;/\; (v\%\tau) & \\ & \text{if } \forall n \geq 0\ (v\%\tau) \neq \bot_n \\ & \text{and } \neg(v\%\tau).\text{isActiveTreeProven}() \ , \end{array}$$

Catch The Try tactical catches errors of level 0, and decreases the level of other errors by 1.

$$\begin{array}{ll} \text{Try } v \;/\; \tau \xrightarrow{\epsilon} \text{Idtac} \;/\; \tau & \text{if } (v\%\tau) = \bot_0 \\ \text{Try } v \;/\; \tau \xrightarrow{\epsilon} [\text{Fail } (n-1)] \;/\; \tau & \text{if } \exists n > 0\ (v\%\tau) = \bot_0 \\ \text{Try } v \;/\; \tau \xrightarrow{\epsilon} v \;/\; \tau & \text{if } \forall n \geq 0\ (v\%\tau) \neq \bot_0 \ . \end{array}$$

First tactic to succeed Applies the first tactic that does not fail. It fails if all of its arguments fail.

$$\begin{array}{ll} \text{First } [\,] \;/\; \tau \xrightarrow{\epsilon} (\text{Fail } 0) \;/\; \tau & \\ \text{First } [v_1|e_2|\ldots|v_n] \;/\; \tau \xrightarrow{\epsilon} v_1 \;/\; \tau & \text{if } \forall n \geq 0\ (v_1\%\tau) \neq \bot_n \\ \text{First } [v_1|e_2|\ldots|e_n] \;/\; \tau \xrightarrow{\epsilon} \text{First } [e_2|\ldots|e_n] \;/\; \tau & \text{if } \exists n \geq 0\ (v_1\%\tau) = \bot_n \ . \end{array}$$

First tactic to solve Applies the first tactic that solves the current goal. It fails if none of its arguments qualify.

$$\begin{array}{ll} \text{Solve } [\,] \;/\; \tau \xrightarrow{\epsilon} \text{Fail } 0 \;/\; \tau & \\ \text{Solve } [v_1|e_2|\ldots|e_n] \;/\; \tau \xrightarrow{\epsilon} v_1 \;/\; \tau & \text{if } (v_1\%\tau).\text{isActiveTreeProven}() \\ \text{Solve } [v_1|e_2|\ldots|v_n] \;/\; \tau \xrightarrow{\epsilon} \text{Solve } [e_2|\ldots|e_n] \;/\; \tau & \\ & \text{if } \neg(v_1\%\tau).\text{isActiveTreeProven}() \ . \end{array}$$

4.3 Toplevel Definitions

The semantics of the user-defined tactics and tacticals requires an extension of the meta-notation. Let $\mathcal{M}$ be a memory state object with its two trivial methods newTactical(name, description) and getTactical(name).

$$\begin{array}{l} \mathcal{M}.\text{newTactical}(x, e) \longrightarrow \mathcal{M}\{x \leftarrow\!\shortmid e\} \ , \\ \qquad\qquad\qquad\qquad \text{if } x \notin \text{Dom}(\mathcal{M}). \\ \mathcal{M}.\text{getTactical}(x) \longrightarrow \mathcal{M}(x) \ . \end{array}$$

The declaration of new commands simply writes:

$$\begin{array}{l} \text{(Recursive) Tactic Definition } x := v \;/\; \tau \xrightarrow{\epsilon} \mathcal{M}.\text{newTactical}(x, v) \;/\; \tau \ , \\ \text{(Recursive) Meta Definition } x := t \;/\; \tau \xrightarrow{\epsilon} \mathcal{M}.\text{newTactical}(x, t) \;/\; \tau \ , \end{array}$$

where the "Recursive" tag is optional.

Thus when evaluating an expression on which none of the previous reduction rules apply, the following will be tried:

$$x \;/\; \tau \xrightarrow{\epsilon} \mathcal{M}.\text{getTactical}(x) \;/\; \tau \ .$$

4.4 Context

The evaluation context is defined as:

$$
\begin{aligned}
E ::= \ & [\,] \\
& \mid E. \\
& \mid E\ e \mid v\ E \\
& \mid \text{Let } x = E \text{ In } e \\
& \mid E \text{ Orelse } e \mid v \text{ Orelse } E \\
& \mid E; e \mid v; E \\
& \mid \overline{;_T} E \mid \overline{;_T} E, e_2, \ldots, e_n \\
& \mid \overline{\text{Do}_e}\ n\ E \\
& \mid \overline{\text{Repeat}_e}\ E \\
& \mid \text{Try } E \\
& \mid \text{Progress } E \\
& \mid \text{Match } E \text{ With } (p_i \longrightarrow e_i)_{i=1}^{n} \\
& \mid \text{First}[E|e_2|\ldots|e_n] \\
& \mid \text{Solve}[E|e_2|\ldots|e_n] \\
& \mid \text{Tactic Definition } x := E \mid \text{Meta Definition } x := E \\
& \mid \text{Recursive Tactic Definition } x := E \\
& \mid \text{Recursive Meta Definition } x := E \ .
\end{aligned}
$$

4.5 Tactics

The goal of this section is not to give the semantics for all the tactics but rather to demonstrate on a few specific examples how the application of simple tactics to a proof context can be expressed.
In general tactics apply to a sequent tree, but will be exposed here only the case where τ.active designates a leaf. When the pointer designates a subtree, the tactic is simultaneously applied to all the unproven leaves of this subtree.
The following equations define partial functions, they are extended to complete functions by taking the failed proof context $\bot_0$ as a return value for any undefined point.

$$\text{Intro}\%\tau.\ \ \Gamma \vdash (x : A)B \ = \tau.\text{addLeafs}\ (\Gamma, (x : A) \vdash B).\text{setProgess(true)}\ .$$

$$\text{Clear } x\%\tau.\ \ \Gamma, (x : A) \vdash B \ = \tau.\text{addLeafs}\ (\Gamma \vdash B).\text{setProgess(true)}\ ,$$
$$\text{with } \forall (x_i : A_i) \in \Gamma \cdot x \notin A_i .$$

$$\text{Assumption}\%\tau.\ \ \Gamma, (x : A) \vdash A' \ = \tau.\text{setLeafProven}().\text{setProgess(true)}\ ,$$
$$\text{with } A \text{ and } A' \text{ unifiable.}$$

$$\text{Cut } A\%\tau.\ \ \Gamma \vdash B \ = \tau.\text{addLeafs}\ (\Gamma \vdash (x : A) \cdot B, \Gamma \vdash A).\text{setProgess(true)}\ .$$

The identity was introduced in [4] as a tactical, but it behaves as a tactic:

$$\text{Idtac}\%\tau = \tau\ .$$

The same holds for the error command:

$$(\text{Fail } n)\%\tau = \bot_n\ .$$

5 PVS

PVS tactics and strategies are thoroughly described in [8] and [6], but as far as we know, there is no published small-step semantics of the strategy language.

5.1 Syntax

Here is the syntax of the subset of PVS's tactics that will be considered: not all of PVS's strategies are exposed here; those that appear are believed to be the most significant ones, the others being either special cases or slight variants of the aforementionned.

Contrary to Coq, there is no symbol in PVS to mark the end of the proof command. This problem is dealt with by using a special symbol (¶):

$$e \quad ::= \quad \mathit{expr}\ \text{¶} \qquad\qquad \text{all expressions must end with "¶"} \ .$$

And

$$\begin{array}{lll}
\mathit{expr} ::= & x & \text{identifier} \\
\mid & p & \text{tactic} \\
\mid & t & \text{Lisp term} \\
\mid & \texttt{(if}\ t\ e_1\ e_2\texttt{)} & \\
\mid & \texttt{(let ((}x_1\ t_1\texttt{)}\ldots\texttt{(}x_n\ t_n\texttt{))}\ e\texttt{)} & \\
\mid & \texttt{(try}\ e_1\ e_2\ e_3\texttt{)} & \\
\mid & \texttt{(repeat}\ e\texttt{)} & \\
\mid & \texttt{(repeat*}\ e\texttt{)} & \\
\mid & \texttt{(spread}\ e_0\ \texttt{(}e_1\ldots e_n\texttt{))} & \\
\mid & \texttt{(branch}\ e_0\ \texttt{(}e_1\ldots e_n\texttt{))} & \\
\mid & \texttt{(try-branch}\ e_0\ \texttt{(}e_1\ldots e_n\texttt{)}\ e_{n+1}\texttt{)} \ . &
\end{array}$$

5.2 Semantics

There are no abstraction strategies in PVS therefore the values are defined as the tactics:

$$v \quad ::= \quad p \ .$$

The reduction rules for the tacticals follow.

Break ¶ triggers the evaluation of the tactics and does the final proof context parameter reset:

$$v\ \text{¶}\ /\ \tau \xrightarrow{\epsilon} (v\%\tau).\mathrm{raisePointerToLeaf}().\mathrm{setProgress}(\mathrm{false}) \ .$$

Lisp conditional A lisp argument t is evaluated to determine whether the first or the second tactic argument is applied.

$$\begin{array}{ll}
\texttt{(if}\ t\ e_1\ e_2\texttt{)}\ /\ \tau \xrightarrow{\epsilon} e_2\ /\ \tau & \text{if } t = \texttt{nil} \\
\texttt{(if}\ t\ e_1\ e_2\texttt{)}\ /\ \tau \xrightarrow{\epsilon} e_1\ /\ \tau & \text{if } t \neq \texttt{nil} \ .
\end{array}$$

Lisp variable binding The local variable binding strategy. The symbols x_i are bound to the lisp expressions t_i in the latter bindings and in e.

$$\begin{array}{l}
\texttt{(let ((}x_1\ t_1\texttt{)}\ldots\texttt{(}x_n\ t_n\texttt{))}\ e\texttt{)}\ /\ \tau \xrightarrow{\epsilon} \\
\qquad e[x_1 \hookleftarrow t_1, \ldots, x_n \hookleftarrow t_n]\ /\ \tau \ .
\end{array}$$

Backtracking This strategy combines a branching facility triggered by the progress condition, with an error catching functionnality. It applies v_1 to the current goal, it this shows a progress then it applies v_2, else it applies v_3. Moreover, if v_2 fails then this strategy returns `(skip)`. This final backtracking feature calls for the use of an additional operator $\overline{\text{try}_{\mathcal{T}}}$.
Remark that the sequencial tactical `then` is simply defined as $(\texttt{then}\ v_1\ v_2) = (\texttt{try}\ v_1\ v_2\ v_2)$.

$$\begin{array}{ll}
(\texttt{try}\ v_1\ e_2\ e_3)\ /\ \tau \xrightarrow{\epsilon} (\overline{\text{try}_{\mathcal{T}}}\ e_2)\ /\ (v_1\%\tau) & \text{if } (v_1\%\tau).\text{hasProgressed}() \\
& \text{and } \forall n \geq 0\ (v_1\%\tau) \neq \bot_n \\
& \text{and } \neg(v_1\%\tau).\text{isActiveTreeProven}() \\
(\texttt{try}\ v_1\ e_2\ e_3)\ /\ \tau \xrightarrow{\epsilon} (\texttt{fail})\ /\ \tau & \text{if } \exists n \geq 0\ (v_1\%\tau) = \bot_n \\
(\texttt{try}\ v_1\ e_2\ e_3)\ /\ \tau \xrightarrow{\epsilon} e_3\ /\ \tau & \text{if } \neg(v_1\%\tau).\text{hasProgressed}() \\
(\texttt{try}\ v_1\ e_2\ e_3)\ /\ \tau \xrightarrow{\epsilon} v_1\ /\ \tau & \text{if } (v_1\%\tau).\text{isActiveTreeProven}(),
\end{array}$$

with

$$\begin{array}{ll}
(\overline{\text{try}_{\mathcal{T}}}\ v)\ /\ \tau' \xrightarrow{\epsilon} v\ /\ \tau' & \text{if } \forall n \geq 0\ (v\%\tau') \neq \bot_n \\
(\overline{\text{try}_{\mathcal{T}}}\ v)\ /\ \tau' \xrightarrow{\epsilon} (\texttt{skip})\ /\ \tau & \text{if } \exists n \geq 0\ (v\%\tau') = \bot_n\ .
\end{array}$$

Indefinite iteration The tactic argument is applied to the current goal, if it generates any subgoals then it is recursively applied to the first of these subgoals. The repetition stops when an application of the tactic has no effect.

$$(\texttt{repeat}\ e)\ /\ \tau \xrightarrow{\epsilon} \overline{\text{repeat}_e}\ e\ /\ \tau\ ,$$

with

$$\begin{array}{ll}
\overline{\text{repeat}_e}\ v\ /\ \tau \xrightarrow{\epsilon} \text{Idtac}\ /\ \tau & \text{if } \exists n \geq 0\ (v\%\tau) = \bot_n \\
\overline{\text{repeat}_e}\ v\ /\ \tau \xrightarrow{\epsilon} v\ /\ \tau & \text{if } (v\%\tau).\text{isActiveTreeProven}() \\
\overline{\text{repeat}_e}\ v\ /\ \tau \xrightarrow{\epsilon} \overline{\text{repeat}_e}\ e\ /\ (v\%\tau).\text{raisePointerToLeaf}() & \\
& \text{if } \forall n \geq 0\ (v\%\tau) \neq \bot_n \\
& \text{and } \neg(v\%\tau).\text{isActiveTreeProven}()\ ,
\end{array}$$

Like `repeat`, `repeat*` repeats v, but on all the previously generated subgoals.

$$(\texttt{repeat*}\ e)\ /\ \tau \xrightarrow{\epsilon} \overline{\text{repeat*}_e}\ e\ /\ \tau\ ,$$

with

$$\begin{array}{ll}
\overline{\text{repeat*}_e}\ v\ /\ \tau \xrightarrow{\epsilon} (\texttt{skip})\ /\ \tau & \text{if } \exists n \geq 0\ (v\%\tau) = \bot_n \\
\overline{\text{repeat*}_e}\ v\ /\ \tau \xrightarrow{\epsilon} v\ /\ \tau & \text{if } (v\%\tau).\text{isActiveTreeProven}() \\
\overline{\text{repeat*}_e}\ v\ /\ \tau \xrightarrow{\epsilon} \overline{\text{repeat*}_e}\ e\ /\ (v\%\tau) & \\
& \text{if } \forall n \geq 0\ (v\%\tau) \neq \bot_n \\
& \text{and } \neg(v\%\tau).\text{isActiveTreeProven}()\ ,
\end{array}$$

N-ary sequence The N-ary sequence in PVS is similar to that of Coq, but here the number of generated subgoals need not be exactly n.

$$\begin{array}{c}
(\texttt{spread}\ v_0\ (e_1 \dots e_n))\ /\ \tau \xrightarrow{\epsilon} \\
\overline{\text{spread}_{\mathcal{T}}^{v_0,e_1,\dots,e_n}} e_1, \dots, e_n\ /\ (v_0\%\tau).\text{raisePointerToLeaf}()\ ,
\end{array}$$

and, with l representing the list $v_0, e_1, \dots, e_n$:

$$\begin{array}{l}
\qquad \overline{\text{spread}_{\mathcal{T}}^{l}} v_1, e_2 \dots, e_n\ /\ \tau' \xrightarrow{\epsilon} \\
\overline{\text{spread}_{\mathcal{T}}^{l}} e_2, \dots, e_n\ /\ (v_1\%\tau').\text{pointNextSibling}() \\
\qquad\qquad\qquad\qquad \text{if } \forall n \geq 0\ (v_1\%\tau') \neq \bot_n\ ,
\end{array}$$

and

$$\overline{\text{spread}^{l}_{T}} v_1, e_2 \ldots, e_n \;/\; \tau\text{'} \xrightarrow{\epsilon} \texttt{(fail)} \;/\; \tau \quad \text{if } \exists n \geq 0\ (v_1\%\tau\text{'}) = \bot_n \ ,$$

and

$$\begin{aligned}&\overline{\text{spread}^{v_0,e_1,\ldots,e_n}_{T}} v_n \;/\; \tau\text{'} \xrightarrow{\epsilon} \\ \overline{\text{spread}^{v_0,e_1,\ldots,e_{n-1}}_{T}}\, & v_0, e_1, \ldots, e_{n-1} \;/\; \tau \quad \text{if } \tau\text{'}= \varnothing \ ,\end{aligned}$$

and

$$\begin{aligned}\overline{\text{spread}^{l}_{T}} v_n \;/\; \tau\text{'} \xrightarrow{\epsilon} \texttt{(skip)} \;/\; &(v_n\%\tau\text{'}).\,\text{lowerPointer}(1)\\ &\text{if } \tau\text{'} \neq \varnothing\\ &\text{and } (v_n\%\tau\text{'}).\,\text{pointNextSibling}() = \varnothing \ ,\end{aligned}$$

and

$$\begin{aligned}&\overline{\text{spread}^{v_0,e_1,\ldots,e_n}_{T}} v_n \;/\; \tau\text{'} \xrightarrow{\epsilon}\\ &\overline{\text{spread}^{v_0,e_1,\ldots,e_n,(skip)}_{T}} v_0, e_1, \ldots, e_n, \texttt{(skip)} \;/\; \tau\\ &\qquad\qquad \text{if } (v_n\%\tau\text{'}).\,\text{pointNextSibling}() \neq \varnothing \ ,\end{aligned}$$

The `(branch ...)` method behaves likewise, but repeats the last element of the list on all the remaining siblings when necessary:

$$\begin{aligned}&\texttt{(branch } v_0 \texttt{ (}e_1 \ldots e_n\texttt{))} \;/\; \tau \xrightarrow{\epsilon}\\ &\overline{\text{branch}^{v_0,e_1,\ldots,e_n}_{T}} e_1, \ldots, e_n \;/\; (v_0\%\tau).\,\text{raisePointerToLeaf}() \ .\end{aligned}$$

The reduction rules are the same for $\overline{\text{branch}^{v_0,e_1,\ldots,e_n}_{T}}$ as for $\overline{\text{spread}^{v_0,e_1,\ldots,e_n}_{T}}$, but for the last rule:

$$\begin{aligned}&\overline{\text{branch}^{v_0,e_1,\ldots,e_n}_{T}} v_n \;/\; \tau\text{'} \xrightarrow{\epsilon}\\ &\overline{\text{branch}^{v_0,e_1,\ldots,e_n,e_n}_{T}} v_0, e_1, \ldots, e_n, e_n \;/\; \tau\\ &\qquad\qquad \text{if } (v_n\%\tau\text{'}).\,\text{pointNextSibling}() \neq \varnothing \ ,\end{aligned}$$

N-ary backtracking A combination of the `try` and the `branch` strategies, `try-branch` applies v_1 to the current goal, and in case it generated subgoals it applies each of the v_i' to one of the subgoals. Else it applies v_2. As for `try`, this strategy catches any failure that would arise from the application of any of the v_i'.

$$\begin{aligned}&\texttt{(try-branch } v_0 \ (e_1 \ldots e_n)\ e'\texttt{)} \;/\; \tau \xrightarrow{\epsilon}\\ &(\overline{\text{try-branch}^{v_0,e_1,\ldots,e_n}_{T}}\ e_1 \ldots e_n) \;/\; (v_0\%\tau)\\ &\qquad\qquad \text{if } (v_0\%\tau).\,\text{hasProgressed}()\\ &\qquad\qquad \text{and } \forall n \geq 0\ (v_0\%\tau) \neq \bot_n \ .\end{aligned}$$

$$\begin{aligned}&\texttt{(try-branch } v_0 \ (e_1 \ldots e_n)\ e'\texttt{)} \;/\; \tau \xrightarrow{\epsilon} \texttt{(fail)} \;/\; \tau \quad \text{if } \exists n \geq 0\ (v_0\%\tau) = \bot_n\\ &\texttt{(try-branch } v_0 \ (e_1 \ldots e_n)\ e'\texttt{)} \;/\; \tau \xrightarrow{\epsilon} e' \;/\; \tau \quad \text{if } \neg(v_0\%\tau).\,\text{hasProgressed}(),\end{aligned}$$

with

$$\begin{aligned}&(\overline{\text{try-branch}^{l}_{T}}\ v_1 e_2 \ldots e_n) \;/\; \tau\text{'} \xrightarrow{\epsilon}\\ &(\overline{\text{try-branch}^{l}_{T}}\ e_2 \ldots e_n) \;/\; (v_1\%\tau\text{'}).\,\text{pointNextSibling}()\\ &\qquad\qquad \text{if } \forall n \geq 0\ (v_1\%\tau\text{'}) \neq \bot_n \ ,\end{aligned}$$

and

$$\overline{(\text{try-branch}^{l}_{\mathcal{T}}}\ v_1e_2\dots e_n)\ /\ \tau\text{'} \xrightarrow{\epsilon} (\texttt{skip})\ /\ \tau$$
$$\text{if } \exists n \geq 0\ (v_1\%\tau\text{'}) = \bot_n\ ,$$

and

$$\overline{\text{try-branch}^{v_0,e_1,\dots,e_n}_{\mathcal{T}}}\,v_n\ /\ \tau\text{'} \xrightarrow{\epsilon}$$
$$\overline{\text{try-branch}^{v_0,e_1,\dots,e_{n-1}}_{\mathcal{T}}}\,e_1,\dots,e_{n-1}\ /\ (v_0\%\tau)$$
$$\text{if } \tau\text{'} = \varnothing \text{ or } (v_n\%\tau\text{'}).\text{pointNextSibling}() \neq \varnothing\ ,$$

and

$$\overline{\text{try-branch}^{l}_{\mathcal{T}}}v_n\ /\ \tau\text{'} \xrightarrow{\epsilon} (\texttt{skip})\ /\ (v_n\%\tau\text{'}).\text{lowerPointer}(1)$$
$$\text{if } \tau\text{'} \neq \varnothing \text{ and } (v_n\%\tau\text{'}).\text{pointNextSibling}() = \varnothing\ ,$$

and

$$\overline{\text{try-branch}^{v_0,e_1,\dots,e_n}_{\mathcal{T}}}\,v_n\ /\ \tau\text{'} \xrightarrow{\epsilon}$$
$$\overline{\text{try-branch}^{v_0,e_1,\dots,e_n,e_n}_{\mathcal{T}}}\,e_1,\dots,e_n,e_n\ /\ (v_0\%\tau)$$
$$\text{if } (v_n\%\tau\text{'}).\text{pointNextSibling}() \neq \varnothing\ ,$$

5.3 User-defined strategies

As for Coq, the meta-notation needs to be enriched to cope with the user definitions. Let $\mathcal{M}$ be a memory state object storing the new strategies, and its methods setStrategy(name, description) and getStrategy(name). Unlike Coq though, PVS uses a specific file, `pvs-strategies`, to load user definitions, and does not allow for toplevel declarations. Moreover, these definitions split into two categories, rules i.e. atomic commands or *blackbox*, and strategies i.e. non-atomic commands or *glassbox*.
PVS calls the setStrategy at launch to initialize the memory state, and only allows readings during runtime:

$$\mathcal{M}.\text{getStrategy}(x) \longrightarrow \mathcal{M}(x)\ ,$$

where $\mathcal{M}(x) = (\mathit{Box}\ e)$, *Box* is one of the two tags `Glass` or `Black`, and e is a proof script. The tags are not part of the real PVS syntax: they are introduced here to describe a phenomenon that is actually hidden in the implementation.
When evaluating a tactic on which none of the previous reduction rules apply, the following will be tried:

$$x\ /\ \tau \xrightarrow{\epsilon} \mathcal{M}.\text{getStrategy}(x)\ /\ \tau\ .$$

Finally this calls for a definition of the semantics of the `Glass` and `Black` commands:

$$\begin{array}{lll}(\texttt{Black}\ v)\ /\ \tau \xrightarrow{\epsilon} (\texttt{skip})\ /\ \tau & \text{if } \exists n \geq 0\ (v\%\tau) = \bot_n \\ (\texttt{Black}\ v)\ /\ \tau \xrightarrow{\epsilon} v\ /\ \tau & \text{if } \forall n \geq 0\ (v\%\tau) \neq \bot_n\ ,\end{array}$$

$$(\texttt{Glass}\ v)\ /\ \tau \xrightarrow{\epsilon} v\ /\ \tau\ .$$

5.4 Context

$$\begin{aligned}
E ::= &\ [\,] \\
&\mid E\ \P \\
&\mid \texttt{(try}\ E\ e_2\ e_3\texttt{)} \\
&\mid \overline{\mathrm{try}_{\mathcal{T}}}E \\
&\mid \texttt{(spread}\ E\ (e'_1 \ldots e'_n)\texttt{)} \\
&\mid \overline{\mathrm{spread}_{\mathcal{T}}^{v_0,e_1,\ldots,e_n}}E\ e_i \ldots e_n \\
&\mid \texttt{(branch}\ E\ (e'_1 \ldots e'_n)\texttt{)} \\
&\mid \overline{\mathrm{branch}_{\mathcal{T}}^{v_0,e_1,\ldots,e_n}}E\ e_i \ldots e_n \\
&\mid \texttt{(try-branch}\ E\ (e'_1 \ldots e'_n)\ e_2\texttt{)} \\
&\mid \overline{\mathrm{try\text{-}branch}_{\mathcal{T}}^{v_0,e_1,\ldots,e_n}}E\ e_i \ldots e_n \\
&\mid \texttt{(Glass}\ E\texttt{)} \\
&\mid \texttt{(Black}\ E\texttt{)}\ .
\end{aligned}$$

5.5 Tactics

The same conventions will be used as for Coq's tactics. Note that PVS does not use the error level: $\bot_0$ is the only error possible.

$$\texttt{(flatten)}\%\tau.\ \ \Gamma \vdash A \supset B\ = \tau.\mathrm{addLeaves}\ (\Gamma, A \vdash B).\mathrm{setProgess(true)}\ .$$

$$\texttt{(flatten)}\%\tau.\ \ \Gamma \vdash A \vee B\ = \tau.\mathrm{addLeaves}\ (\Gamma \vdash A, B).\mathrm{setProgess(true)}\ .$$

$$\begin{gathered}\texttt{(flatten)}\%\tau.\ \ \Gamma, A \wedge B \vdash C\ = \\ \tau.\mathrm{addLeaves}\ (\Gamma, A, B \vdash C).\mathrm{setProgess(true)}\ .\end{gathered}$$

$$\texttt{(propax)}\%\tau.\ \ \Gamma, A \vdash B\ = \tau.\mathrm{leafProven()}.\mathrm{setProgess(true)}$$
if A and B are syntaxically equal.

$$\texttt{(beta)}\%\tau.\ \ \Gamma \vdash (\lambda x : t)(u)\ = \tau.\mathrm{addLeaves}\ (\Gamma \vdash t[x \leftarrow\!\!\!\dashv u]).\mathrm{setProgess(true)}\ .$$

$$\texttt{(skip)}\%\tau = \tau\ .$$

$$\texttt{(fail)}\%\tau = \bot_0\ .$$

$$\begin{gathered}\texttt{(skolem * (``a''))}\%\tau.\ \ \Gamma, (\exists x : A) \vdash B\ = \\ \tau.\mathrm{addLeaves}\ (\Gamma, A[x \leftarrow\!\!\!\dashv a] \vdash B).\mathrm{setProgess(true)}\ .\end{gathered}$$

$$\begin{gathered}\texttt{(skolem * (``a''))}\%\tau.\ \ \Gamma \vdash (\forall x : A)\ = \\ \tau.\mathrm{addLeaves}\ (\Gamma \vdash A[x \leftarrow\!\!\!\dashv a]).\mathrm{setProgess(true)}\ .\end{gathered}$$

6 Conclusion and Future Work

We have presented a small step semantics for the core of both Coq and PVS's tacticals, as well as for some simple tactics. This semantics seems correct with respect to the formal definition of both languages, provided for Coq by Delahaye's definition of $\mathcal{L}_{tac}$ [4], and for PVS by the Prover Guide [11]. A proof of correctness of our semantics in regard with these definitions is currently under way. Future work will also try to incorporate more advanced tactics to the system, although this will certainly prove more difficult, entailing the use of global proof environments and variables, α-equivalence classes, and most likely the integration of PVS-like automatic conversion methods. It might also be interesting to express tacticals from other languages (such as Isabelle or NuPrl) in this framework, and the idea of a correlation between proof tacticals and rewriting strategies might be worth studying. Nevertheless the formal basis of the semantics is easily and conservably extendable, and should allow for an efficient and – hopefully – not too complicated continuation.

Finally, beyond its informative features, this work sets the very basis for an unified representation of PVS's strategies and Coq's tacticals, which would allow for proof portability, double-checking, prover-relevancy modularization, i.e., an overall improved flexibility and interoperability.

References

[1] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version 7.4. http://coq.inria.fr/doc/main.html, 2003.

[2] David Carlisle, Scott Pakin, and Alexander Holt. *The Great, Big List of LaTeX Symbols*, February 2001.

[3] H. Cirstea, C. Kirchner, and L. Liquori. Rewrite Strategies in the Rewriting Calculus. In *WRLA '02*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science B.V., 2003.

[4] David Delahaye. *Conception de langages pour décrire les preuves et les automatisations dans les outils d'aide à la preuve*. Thèse de doctorat, Université Paris 6, 2001.

[5] Catherine Dubois. Proving ML Type Soundness Within Coq. In Mark Aagaard and John Harrison, editors, *TPHOLs*, volume 1869 of *Lecture Notes in Computer Science*, pages 126–144. Springer, 2000.

[6] César Muñoz. Strategies in PVS. Lecture notes, 2002. National Institute of Aerospace.

[7] Tobias Oetiker. *The Not So Short Introduction to LaTeX2e*, January 1999.

[8] Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report CR-1999-209321, Computer Science Laboratory, SRI International, Menlo Park, CA, May 1999.

[9] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[10] François Pottier. Typage et Programmation. Lecture notes, 2002. DEA PSPL.

[11] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[12] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.

Rippling in PVS

A. A. Adams[1] and L. A. Dennis[2]

[1] The University of Reading
A.A.Adams@Rdg.ac.uk
http://www.rdg.ac.uk/~sis00aaa/
[2] The University of Nottingham
lad@cs.nott.ac.uk
http://cs.nott.ac.uk/~lad/

Abstract. Rippling is a method of controlling rewriting of the terms in an induction step of an inductive proof, to ensure that a position is reached whereby the induction hypothesis can be applied. Rippling was developed primarily by the Mathematical Reasoning Group at the University of Edinburgh. The primary implementations are in the two proof planning systems Clam and λClam. An implementation is also available in HOL. In this paper we explain how we plan to implement rippling as a tactic for automatic generation of proofs requiring induction in PVS. Rippling has mostly been used as part of a larger project for developing high-level proof strategies, but has rarely been applied to "real-world" examples. Once we have this implementation we intend to assess the utility of this as a tactic by running rippling on the large number of inductive proofs developed by Gottliebsen as part of the PVS Real Analysis library [8]. By comparing the performance of the automation offered by rippling on these proofs with the original proof, which were proved by a combination of hand-generation of proofs and by existing PVS strategies, we hope to assess the utility of rippling as a technique for real-world applications.

1 Introduction

When faced with a lemma (or sub-goal) requiring an inductive proof, the experience PVS user will invoke one of the PVS strategies such as `induct-and-rewrite` or `induct-and-simplify`. While in many cases these may work, there are also often cases where more user-control is necessary, requiring a user to control the rewriting stages of the inductive parts of the proof manually. The Real Analysis library developed by Gottliebsen [8] contains numerous such inductive proofs (mostly convergence arguments for power series) which do not fall to the existing automation of PVS.

There has been a great deal of work on automating inductive proof in different provers, but the approach we concentrate on is *rippling*. This approach is detailed below, but it is primarily a method for controlling the use of rewrite rules to ensure that they are used in the correct order, and in the correct orientation, to allow cross-fertilisation, i.e. the use of the induction hypothesis. In addition to control of the rewriting steps necessary to prove the induction step, it is possible to choose the wrong induction scheme to apply in the first place. We propose to implement rippling and induction scheme choice in PVS to produce tactics that will complement the existing induction tactics in PVS. We plan to test our hypothesis that induction choice and rippling can prove goals on which existing tactics fail, by applying the resulting system to the PVS Real Analysis library's induction proofs.

2 Rippling

Rippling is a rewriting technique which relies upon a difference reduction heuristic rather than an ordering on terms for termination. This makes it attractive in an induction setting where it has been used extensively (see [6] for a full discussion) in which the object is to reduce the differences between the induction conclusion and the induction hypothesis although it has also been used in other settings such as equational rewriting [11].

Rippling was first introduced in [5]. We intend to use the theory as presented by Smaill & Green [13] who proposed a version that naturally copes with higher-order features. Rippling steps apply rewrite rules to a target term (called the *erasure*) which is associated with a *skeleton* and an *embedding* that relates the skeleton to the erasure (e.g. rippling rewrites an induction conclusion which has an induction hypothesis embedded in it so the induction hypothesis is the skeleton and the conclusion is the erasure). After rewriting, a new embedding of the skeleton into the rewritten term is calculated. There is a measure on embeddings and any rewriting step must reduce this *embedding measure* (written as $<_\mu$).

Rippling is terminating. Rippling either moves differences outwards in the term structure so that they can be cancelled away or inwards so that the differences surround a universally quantified variable (or *sink*). If it is possible to move differences inwards in this way the embedding is said to be *sinkable*. The measure on embeddings allows differences that are being moved outwards to be moved inwards but not vice versa. There is not space here to give full details of the embedding measure – details of the first-order measure can be found in [1] and details of the measure for embeddings can be found in [7][3].

Embeddings treat the syntax of terms as a variant of Higher-Order Abstract Syntax or λ-tree syntax [12] in which abstraction and application are explicitly represented as nodes in the term tree.

The position of a node in a term tree is determined by a list of integers indicating the path from the root of the tree to that node (for implementation reasons this list is usually read from right to left). The operator of an application term is labelled as the 1st branch and then each member of the ensuing tuple from 2 to n in order[4]. λ-abstraction nodes are ignored when calculating term positions. So the position of g in the term $g(\lambda x.f(a,x,b))$ is [1], the position of x is [3,2] and the position of f is [1,2].

Embeddings are also described by a tree data structure. Embedding trees describe how the skeleton term tree embeds in the erasure term tree. The nodes in an embedding tree can be viewed as labels on the nodes in the term tree of the skeleton (excluding λ-abstraction nodes). These labels contain addresses. The addresses are the addresses of nodes in the term tree of the erasure into which the skeleton is to be embedded. A node in an embedding tree will appear at a function application node in the skeleton term tree and indicates that this node is matched to the function application term in the erasure term tree at the indicated address. Similarly the leaves of an embedding tree are attached to the leaves of the skeleton term tree.

Example 1 *Consider embedding the term $\lambda x.\ f(x)$ into the term $\lambda y.\ \lambda x.\ (f(y) + x)$. We do this as in figure 1. The two terms are shown as trees with branches represented by solid lines. The address of each node is given (λ-abstraction nodes do not carry addresses). The embedding appears between them as an embedding tree with dashed lines — the address label of the nodes is also shown. The dotted arrows illustrate how the embedding tree links the two terms.*

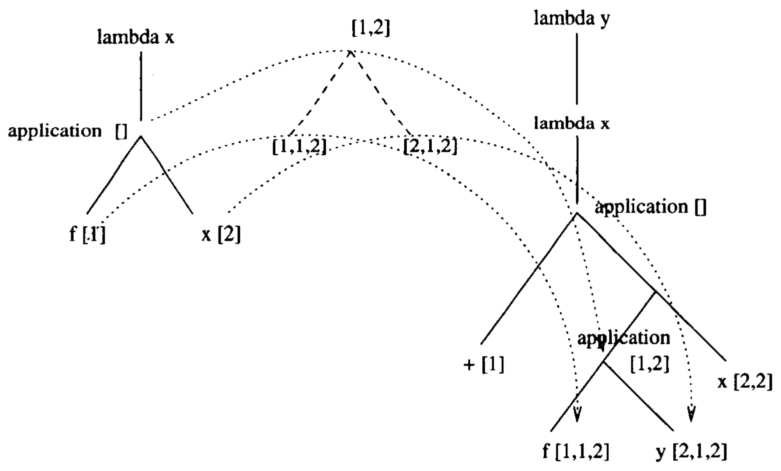

Fig. 1. An Embedding

[3] Draft version available from second author.

[4] This assumes an uncurried syntax is being used — it is easy to restrict rippling to curried syntax if desired.

In proofs by induction, using a constructor formulation, the measure is used as follows. Given an induction hypothesis $\Phi(x)$ and goal $\Phi(s(x))$, there is an embedding, e, between $\Phi(x)$ and $\Phi(s(x))$. Recursion equations and lemmas are available as rewrite rules to be used on the goal. Typically, these rewrite rules are not confluent, and include potentially looping rules, such as associativity. The following heuristic can be used: use rewrites which give a new goal G' such that there is an embedding, e', between $\Phi(x)$ and G', and such that $e' <_{\mathcal{M}} e$, *i.e.* the embedding is smaller in the measure.

The rippling heuristic imposes three conditions on such rewriting:

1. *soundness* — the rewriting should correspond to logically valid inference;
2. *skeleton invariance* — the appropriate skeleton embeds in the goal, before and after rewriting;
3. *measure decreasing* — the embedding in the rewritten formula is smaller in the order.

First order rippling calculi use goals decorated with annotations in the form of boxes, underlines and arrows. We intend to use these conventions within this paper in order to indicate to the reader how the current embedding relates the skeleton to the erasure but do not intend to include such decorations in our implementation. If one expression can be embedded into another then this is indicated in the notation by placing boxes (or *wave fronts*) round the additional parts of the larger expression. The embedded expression can be identified as those parts of the expression that are outside the boxes or those parts within boxes which are underlined (called *wave holes*). As an example the result of difference matching[5] $s(x)$[6] and $s(x+y)$ would be $s(\boxed{\underline{x}+y})$. Here the first expression, $s(x)$, appears outside the wave fronts or in the wave holes of the annotated expression. If the annotations of $s(\boxed{\underline{x}+y})$ are stripped away then the second expression, $s(x+y)$, is left. $s(x)$ is the skeleton and $s(x+y)$ is the erasure.

The main advantages of rippling are that it allows an equation to be treated as a rewrite in both directions without loss of termination and provides useful information for automatically patching failed proof attempts.

Example 2 *Consider the associativity of disjunction:*

$$A \vee (B \vee C) \equiv (A \vee B) \vee C,$$

which can be used as a rewrite in either direction, depending upon the skeleton to be preserved, and the direction in which the wave-front is to be moved. For example, the term $A \vee \boxed{(\underline{B} \vee C)}$ *can be rewritten to* $\boxed{\underline{(A \vee B)} \vee C}$ *using the equation as a rewrite rule from left to right while it is also possible (perhaps later in the same proof) to rewrite the term* $\boxed{(A \vee \underline{B})} \vee C$ *to* $\boxed{A \vee (\underline{B \vee C})}$ *using the rule from right to left.*

In the first case the skeleton is $A \vee B$ (which embeds into the goal both before and after rewriting), and in the second it is $B \vee C$. Thus one is able to use rewrites in both *directions during rippling so long as they are in contexts where the skeletons differ.*

During rippling the direction in which a difference is moving is indicated using an arrow on the wave fronts. If a difference is moving outward in the term structure the arrow is upwards and if the difference is moving inward then the arrow is downwards. We indicate variables which are universally quantified in the skeleton (i.e. sinks) with the symbol $\lfloor\rfloor$. So if for instance the induction hypothesis were $\forall y. n + y = y + n$ then the conclusion might be annotated as $\boxed{s(\underline{n})}^{\uparrow} + \lfloor y \rfloor = \lfloor y \rfloor + \boxed{s(\underline{n})}^{\uparrow}$ [7]. An additional heuristic often used with rippling is to insist that all rewrites involving inward moving differences are *sinkable* (i.e. there is a sink in any subterm involving an inward wave front).

[5] The process by which annotations or embeddings are generated.

[6] The semantics of the symbols here is unimportant to the description but s can be viewed as the successor function on natural numbers and $+$ as addition on natural numbers. This is consistent with the use of these symbols elsewhere in this paper.

[7] Note that in the conclusion the universal quantifier has been removed by skolemisation.

2.1 Coloured Rippling

In [10, 14] notions of “colouring” annotated terms are used for situations in which a number of different skeletons are in play and so differences are manipulated with respect to more than one term. Colouring can be used to indicate to which skeleton a wave hole relates, or whether it relates to both. To achieve this embeddings based versions of rippling extend their annotations so that one erasure is related to several skeleton/embedding pairs.

2.2 An Example of Rippling

Consider the theorem

$$\forall l, m : \mathit{list}(\tau).\ \mathit{rev}(l) <> m = \mathit{qrev}(l, m) \tag{1}$$

The step case goal is annotated by difference matching as

$$\begin{gathered} \mathit{rev}(l) <> M = \mathit{qrev}(l, M) \Rightarrow \\ \mathit{rev}(\boxed{h :: \underline{l}}^{\uparrow}) <> M' = \mathit{qrev}(\boxed{h :: \underline{l}}^{\uparrow}, M') \end{gathered} \tag{2}$$

The following equations are available for use as rewrite rules:

$$\mathit{rev}(X :: Y) \equiv \mathit{rev}(Y) <> X :: \mathit{nil} \tag{3}$$

$$(U <> V) <> W \equiv U <> (V <> W) \tag{4}$$

$$\mathit{qrev}(H :: T, M) \equiv \mathit{qrev}(T, H :: M) \tag{5}$$

Using these the conclusion can be rippled as follows:

$$\boxed{\underline{\mathit{rev}(t)} <> h :: \mathit{nil}}^{\uparrow} <> \lfloor M' \rfloor = \mathit{qrev}(\boxed{h :: \underline{t}}^{\uparrow}, \lfloor M' \rfloor) \tag{6}$$

$$\boxed{\underline{\mathit{rev}(t)} <> h :: \mathit{nil}}^{\uparrow} <> \lfloor M' \rfloor = \mathit{qrev}(t, \boxed{h :: \underline{\lfloor M' \rfloor}}^{\downarrow}) \tag{7}$$

$$\mathit{rev}(t) <> (\boxed{h :: \mathit{nil} <> \underline{\lfloor M' \rfloor}}^{\downarrow}) = \mathit{qrev}(t, \boxed{h :: \underline{\lfloor M' \rfloor}}^{\downarrow}) \tag{8}$$

$h :: \mathit{nil} <> \lfloor M' \rfloor$ can be simplified to $h :: \lfloor M' \rfloor$ making the conclusion

$$\mathit{rev}(t) <> (\boxed{h :: \underline{\lfloor M' \rfloor}}^{\downarrow}) = \mathit{qrev}(t, \boxed{h :: \underline{\lfloor M' \rfloor}}^{\downarrow}) \tag{9}$$

$\lfloor M' \rfloor$ is a sink and the goal can be proved by appeal to the induction hypothesis.

The last simplification step is not a rippling step since $h :: \mathit{nil}$ and h are both unannotated subterms of the annotated terms. At this point rippling is said to be *blocked*, that is no further rippling can occur, but fertilisation isn't possible. There are a number of normalisation techniques which can perform the necessary simplification to unblock rippling. In PVS it will be possible to use existing tactics to complete this last step.

2.3 Implementing Rippling in PVS

In order to implement rippling in PVS we will need to extend the goal representation to allow annotation with embeddings. Once this is done we intend to provide three basic tactics `SET_UP_RIPPLE` which will perform difference matching to annotate a goal with respect to one or more of its hypotheses, `RIPPLE(rule)` which will attempt to rewrite the goal using `rule` — we hope to be able to use existing rewriting tactics within PVS for this and simply add additional checks for the rippling conditions on top of this and lastly a tactic `POST_RIPPLE` which will remove annotations.

Using these basic tactics it will be possible to create more sophisticated tactics which (for instance) choose a rule from PVS' rewrite rule set rather than relying on one provided by the user and which can in one step annotate a goal, ripple as much as possible and then remove annotations thus automating the full rewriting process in the step case of an induction.

We anticipate that taken together these tactics will give the power of fully automated rippling as well as the flexibility of an interactive setting so that a user can intervene and guide the process more carefully if desired.

Extending the PVS goal syntax to allow rippling annotations might prove problematic. However, an alternative approach is to extract a copy of the current goal and annotate it. This annotated goal is then passed as an extra argument to help control the rippling tactics. The annotated goal term can then be recalculated at each step and as a "reality-check" the annotations can be stripped and compared with the actual goal produced by applying a rewrite rule.

3 Ripple Analysis

In addition to controlling the rewriting during the step case rippling has a second major application in proof by induction which is in automating the choice of the induction scheme and variable to use. This automation process is called *ripple analysis* [6] and is based on a rational reconstruction of the recursion analysis of Boyer and Moore [3].

In essence ripple analysis exploits any recursive definitions (or possibly lemmas) for the functions involved in the current goal to create candidate choices for induction schemes. Implementations of ripple analysis examine candidate induction schemes and variable choices and try to see whether a ripple step can be applied to any new structure introduced by the scheme.

Example 3 *Consider the proof of the theorem*

$$\forall a, b, c.\, a + (b + c) = (a + b) + c \tag{10}$$

There are three choices of induction variables here a, b or c. Assume the following equivalence is available as a rewrite rule

$$s(X) + Y \equiv s(X + Y) \tag{11}$$

Consider the annotated conclusion of the the step case if a is chosen as induction variable and the induction scheme

$$\frac{P(0) \qquad P(x) \vdash P(s(x))}{\forall x.\, P(x)} \tag{12}$$

is used:

$$\forall b, c.\, \boxed{s(\underline{a})}^{\uparrow} + (\lfloor b \rfloor + \lfloor c \rfloor) = (\boxed{s(\underline{a})}^{\uparrow} + \lfloor b \rfloor) + \lfloor c \rfloor \tag{13}$$

In this case both occurrences of new structure (the two wave fronts) can be moved by a ripple step using (11). However if b were chosen as the induction variable then the annotated step case conclusion would be

$$\forall a, c.\, \lfloor a \rfloor + (\boxed{s(\underline{b})}^{\uparrow} + \lfloor c \rfloor) = (\lfloor a \rfloor + \boxed{s(\underline{b})}^{\uparrow}) + \lfloor c \rfloor \tag{14}$$

In this case while the wave front on the LHS of the equality can be moved by a ripple step that on the right can not. The wave front on the right is said to be flawed.

Traditionally ripple analysis works with a database of possible schemes and applies them to all candidate induction variables and combinations of induction variables (this allows induction to be applied simultaneously to both a and b (for instance) should there be a rewrite rule which requires a successor constructor in both places — obviously additional base cases need to be generated in such cases). Each candidate scheme/variable choice pair is scored according to how many flawed wave fronts it contains, in general this heuristic score also takes into account the "complexity" of the suggestions (i.e. preferring schemes introducing minimal structure on one variable to schemes introducing more complex new structure or using more than one induction variable) and the suggestion with the highest score is chosen as the candidate for the induction.

Recently Gow [9] has investigated the dynamic generation of induction schemes as an alternative to the use of a fixed database. This was the method used in the original Boyer-Moore system [2], but with rather unsatisfactory results, leading to the removal of this aspect from their later implementations [4]

3.1 Implementing Ripple Analysis in PVS

We intend to implement a new induction tactic in PVS which will automatically chose an induction scheme and induction variable. This tactic will use the rippling machinery we will already have implemented to provide the RIPPLE tactic and will use choose the most appropriate scheme from a defined (but user extensible) set of schemes. There are a number of schemes included in the PVS prelude, such as subrange induction:

```
  k, m: VAR subrange(i, j)
  p: VAR pred[subrange(i, j)]

subrange_induction: LEMMA
    (p(i) AND (FORALL k: k < j AND p(k) IMPLIES p(k + 1)))
        IMPLIES (FORALL k: p(k))
```

Various libraries also implement new induction schemes for new types when introduced.

This tactic can then be combined with our rippling tactics and application of (induction) hypotheses to provide a tactic which completely automates the step case of induction proofs and generates base case goals for the attention of the user.

4 Conclusion

This paper has presented the basics of rippling, which has been shown theoretically (and in practice with other theorem proving systems) to be a useful way of controlling induction proofs. The authors believe it will prove useful as an addition to the strategies currently available in PVS. It is our intention to implement the system as described in this paper and to test our hypothesis of utility on the Real Analysis library. Should these experiments show that rippling is useful we will work with the PVS developers to release a the code for general use.

References

1. David Basin and Toby Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.
2. R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, 1979.
3. R. S. Boyer and J S. Moore. *A Computational Logic.* ACM monograph series. Academic Press, New York, 1979.
4. R. S. Boyer and J. S. Moore. *A Computational Logic Handbook.* Academic Press, 1988.
5. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.
6. Alan Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning, Volume 1.* Elsevier, 2001.
7. L. A Dennis, I. Green, and A. Smaill. Embeddings as a higher-order representation of annotations for rippling. In prep.
8. H. Gottliebsen. Transcendental Functions and Continuity Checking in PVS. pages 198–215. Springer-Verlag LNAI 1869, 2000.
9. J. Gow. *The Dynamic Creation of Induction Rules using Proof Planning.* PhD thesis, Division of informatics.
10. D. Hutter. Guiding inductive proofs. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 147–161. Springer-Verlag, 1990.
11. D. Hutter. Using rippling for equational reasoning. In S. Hölldoblrt, editor, *Proceedings 20th German Annual Conference on Artificial Intelligence KI-96*, number 1137 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1996.

12. D. Miller. Abstract syntax for variable binders: An overview. In et. al. J. Lloyd, editor, *Computation Logic (CL'2000)*, volume 1861 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2000.
13. Alan Smaill and Ian Green. Higher-order annotated terms for proof search. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1275 of *Lecture Notes in Computer Science*, pages 399–414, Turku, Finland, 1996. Springer-Verlag. Also available as DAI Research Paper 799.
14. Tetsuya Yoshida, Alan Bundy, Ian Green, Toby Walsh, and David Basin. Coloured rippling: An extension of a theorem proving heuristic. In A. G. Cohn, editor, *Proceedings of ECAI-94*, pages 85–89. John Wiley, 1994.

Generating Proof-Specific Strategies for PVS*

Pertti Kellomäki

Tampere University of Technology
pertti.kellomaki@tut.fi
http://www.cs.tut.fi/~pk

Abstract. We describe how generated PVS proof strategies were used to partially automate invariant proofs of joint action specifications. A user writes specifications using the DisCo specification language, and a compiler maps the specifications to PVS theories accompanied with custom strategies for verifying invariance theorems in the theories.

1 Introduction

This paper describes work that was done in 1995–1997 using PVS [10] versions up to 2.1. Some of the work has likely been rendered redundant by advances in PVS, especially by improvements in the *grind* strategy, but some of it still remains relevant.

In [7, 6] we present verification of invariant properties of systems specified using the *joint action* formalism. A joint action is an atomic change of state involving multiple participating objects. An action specifies assignments to the participants, and the rest of the world implicitly remains unchanged.

When mapped to the logic of PVS, joint actions give rise to conjunctions of quantified subformulas. The earlier versions of PVS we were working with did not handle quantifications very well automatically, but fortunately it turns out that the proofs have a regular high-level structure which depends on the structure of the specification for which invariants are being verified.

Our solution was to generate a custom PVS strategy for each invariant–action pair in a specification at the same time as mapping the specification to PVS logic. The user then used the generated strategies to drive the proof to a point where some form of human judgment was needed to proceed.

As the concrete vehicle for expressing specifications we used the DisCo [1, 4, 8] specification language. Figure 1 depicts the big picture. A user writes specifications using the DisCo language, and uses the DisCo compiler and an animation facility to validate the specification. Invariant properties can be expressed in the language, and the user is notified if the purported invariants are violated during animation.

If formal proofs of invariants are desired, the compiler can be instructed to generate a PVS theory corresponding to the DisCo specification. The theory is accompanied with a set of custom strategies to make verification easier. Verification is carried out in the usual manner by interacting with PVS, and the custom strategies can be used to automate the routine parts of invariant proofs.

The rest of the paper is organized as follows. Section 2 introduces the joint action approach to specification. Sections 3 and 4 describe mapping DisCo specifications and automation of invariant proofs, respectively. Section 5 discusses the work, and Section 6 concludes the paper.

2 Joint Action Specifications

A joint action [2] specification consists of a set of state variables, a predicate characterizing the initial state, and a set of state transitions called *joint actions*. State variables reside in *objects*, which are instances of *classes*. Joint actions are intended to be used to specify synchronizations of objects in distributed systems.

An action is *enabled*, if a combination of objects exists to participate in the *roles* of the action, such that the *guard* of the action is true. The selection of the action to be executed next and the selection of participants is nondeterministic. Concurrency is modeled via interleaving.

* This research was suported by the Academy of Finland, grant number 102536. Pertti Kellomäki is funded as a Research Fellow by the Academy of Finland.

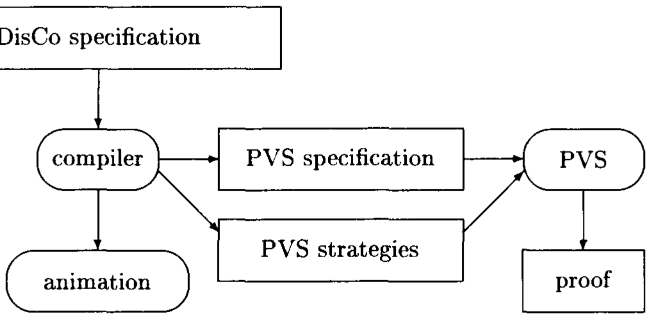

Fig. 1. The DisCo specification environment

The DisCo specification language has a programming language like notation for joint actions. Figure 2 illustrates a DisCo specification. The action *balance* models simple load balancing, and it is enabled for any pair of *node* objects a and b for which the guard $a.w < b.w$ is true. The action changes the values of $a.w$ and $b.w$ to be closer to one another, while the rest of the world remains unchanged.

```
system loadBalance is
  class C is
    w : integer;  -- models the work load of a node
  end;

  assert pos_w is ∀ cc : C :: cc.w ≥ 0;  -- implicitly also an initial condition

  action newJob by c1 : C is
  when c1.w < 10 do      -- an arbitrary upper limit
    c1.w := c1.w + 1;
  end;

  action balance by a, b : node is
  when a.w < b.w do
    a.w := a.w + 1;
    b.w := b.w - 1;
  end;
end;
```

Fig. 2. DisCo specification *loadBalance*

The semantics of joint action specifications can conveniently be given in Lamport's Temporal Logic of Actions [9] (TLA). Each joint action corresponds to a TLA action wrapped in existential quantifications corresponding to the roles of the action.

Conceptually the biggest difference between TLA and DisCo is the treatment of state variables. TLA has named state variable while in DisCo state variables are *fields* contained in anonymous objects. The only way to refer to objects is via quantification, either by referencing explicitly quantified variables or by referencing the implicitly quantified action roles. This indirection is also the main source of additional complexity in proofs.

The TLA meaning of a DisCo action depends on the context in which it is interpreted, because in logic one needs to make explicit which variables are not changed. In a specification which contains a single class *node* with a single attribute w, the TLA meaning of action *balance* is

$$\begin{aligned}
&\exists(a, b : \mathit{node}) : \\
&\quad a \neq b \land a.w < b.w \\
&\quad \land a.w' = a.w + 1 \\
&\quad \land b.w' = b.w - 1 \\
&\quad \land \forall(c : \mathit{node}) : c \neq a \land c \neq b \Rightarrow c.w' = c.w
\end{aligned} \tag{1}$$

where $a.w$ and $a.w'$ refer to the value of field w of participating object a before and after the action, respectively. The conjunct $a \neq b$ arises from the semantics of DisCo actions, which states that an object can only participate in one role of an action.

In a specification where class *node* additionally has an attribute y, the meaning is

$$\begin{aligned}
&\exists(a, b : \mathit{node}) : \\
&\quad a \neq b \land a.w < b.w \\
&\quad \land a.w' = a.w + 1 \\
&\quad \land b.w' = b.w - 1 \\
&\quad \land \forall(c : \mathit{node}) : c \neq a \land c \neq b \Rightarrow c.w' = c.w \\
&\land \forall(c : \mathit{node}) : c.y' = c.y
\end{aligned} \tag{2}$$

Each additional state variable adds a new quantified expression as a conjunct. Every class in the specification must be taken into account, even those that are not mentioned in the action at all. While straightforward in principle, a large number of universally quantified subexpressions makes mechanical reasoning about actions rather tedious in practice.

3 Mapping DisCo to PVS

The mapping from DisCo to PVS closely follows the TLA formulation above. We represent classes as PVS record types, where each field is a function from state to the appropriate PVS type, i.e. a state variable. State is an uninterpreted type, and a behavior is a sequence of states. Figure 3 shows the PVS formulation of the fragment of temporal logic we use.

Figure 4 shows the PVS theories corresponding to the specification in Figure 2. Two theories are generated for a DisCo specification, one containing the definitions corresponding to DisCo classes and actions, and another one containing the theorems to be proved. In order to modularize proofs, a separate lemma is generated for each invariant-action pair.

With hindsight, a different mapping of DisCo to PVS would have been more amenable to mechanical verification. In particular, we could have used the PVS function override mechanism to express assignments to state variables. In the formulation we used, we had to instantiate a quantified formula for each unchanged state variable whose value in the next state we needed to compute. Some proof automation would still have been desirable, because the layered style of developing specification advocated by the DisCo method easily leads to specifications with many classes with many fields.

4 Generating PVS Strategies

The only way to refer to objects in DisCo is via quantification, so all the formulas in the initial sequent of a DisCo invariant proof are quantified. In order to get to the interesting parts of a proof, one has to do a series of skolemizations and instantiations. When an action guard contains quantifications, it is useful to apply the skolemization–instantiation process to it as well.

The skolemizations produce two sets of names in the proof state, one corresponding to the participants of the action and another corresponding to the objects denoted by the quantified variables of the invariant. Since there is a potentially large number of ways in which the two sets can overlap, the top level of an invariant proof is a possibly large case analysis.

Fortunately invariance proofs follow a fixed pattern, so constructing the top level of a proof can be taken care of automatically. A simple heuristic is often sufficient for dealing with quantifiers, and the necessary case analysis can be derived from the skolemizations performed.

```
disco: THEORY
  BEGIN
  state: TYPE
  behavior: TYPE = [nat -> state]
  temporal_formula: TYPE = [behavior -> bool]
  state_predicate: TYPE = [state -> bool]
  action: TYPE = [state, state -> bool]
  stutter(A : action) : action =
    LAMBDA (unprimed, primed : state) :
      A(unprimed, primed) or primed = unprimed
  []((F: temporal_formula), (b: behavior)):
      bool = FORALL (n: nat): F(suffix(b, n))
  statepred2temporal((P: state_predicate)):
      temporal_formula = LAMBDA (b: behavior): P(b(0))
  CONVERSION statepred2temporal
  action2temporal((A: action)): temporal_formula =
    LAMBDA (b: behavior): A(b(0), b(1))
  CONVERSION action2temporal
  invariant((P: state_predicate), (assumptions: bool),
            (I: state_predicate), (A: action)):
      bool = FORALL (b: behavior):
                assumptions AND I(b(0))
                AND [](stutter(A),b) => [](P,b)
  preserves((single_action: action), (P: state_predicate),
            (assumptions: bool), (I: state_predicate),
            (A: action)):
      bool =
    FORALL (b: behavior):
      assumptions AND I(b(0)) AND [](stutter(A),b)
          => FORALL (n: nat):
          P(b(n)) AND single_action(b(n), b(n+1)) => P(b(n+1))
  invariant_rule: THEOREM
        FORALL (P: state_predicate, assumptions: bool,
                I: state_predicate, A: action):
          (FORALL (b: behavior):
             NOT (I(b(0)) AND [](stutter(A),b))
                 OR
               ((assumptions AND I(b(0)) => P(b(0)))
                   AND FORALL (n: nat):
                   assumptions AND P(b(n)) AND A(b(n), b(n + 1))
                      => P(b(n + 1))))
              => invariant(P, assumptions, I, A)
  objid: TYPE
END disco
```

Fig. 3. Formalization of temporal logic in PVS

```
loadBalance: THEORY BEGIN
  discolib: LIBRARY = "/home/korppi-a/pk/vaikkari/pdp/pvs/"
  IMPORTING discolib@disco

  c: TYPE FROM [# w: [state -> int], ref: objid #]

  newJob_guard((c1: c),(other: state)): bool = w(c1)(other) < 10

  newJob((unprimed, primed: state)): bool =
    (EXISTS (c1: c):
       (newJob_guard(c1, unprimed)) AND
            ((w(c1)(primed) = (w(c1)(unprimed) + 1)) AND
            ((FORALL (other: c):
                ((other) /= (c1)) IMPLIES
                   (w(other)(primed) = (w(other)(unprimed))))))

  ⟨action balance omitted⟩

  END loadBalance

loadBalance_assertions: THEORY BEGIN
  IMPORTING example
  c_unique_reference: bool =
    FORALL (obj1, obj2: c): ref(obj1) = ref(obj2)
             IMPLIES obj1 = obj2

  pos_w_body((cc: c), (other: state)): bool = w(cc)(other) >= 0
  pos_w((other: state)): bool =
    (FORALL (cc: c): pos_w_body(cc, other))
  ASSUMPTIONS: bool = c_unique_reference
  INIT((other: state)): bool = pos_w(other)
  ACTIONS((s, sp: state)): bool =
     newJob(s, sp) OR balance(s, sp)

  newJob_preserves_pos_w:
    LEMMA preserves(newJob, pos_w, ASSUMPTIONS, INIT, ACTIONS)

  balance_preserves_pos_w:
    LEMMA preserves(balance, pos_w, ASSUMPTIONS, INIT, ACTIONS)

  pos_w_is_invariant:
    THEOREM invariant(pos_w, ASSUMPTIONS, INIT, ACTIONS)

  END loadBalance_assertions
```

Fig. 4. Mapping of specification *loadBalance* to PVS

In a programmable LCF-style theorem prover such as HOL [3], we could write a program in the prover metalanguage to construct the high level proof. PVS is not programmable in this sense, and it is not possible to write a general strategy that would handle all invariance proofs. Instead, we use an auxiliary program to generate offline a specific instance of the general strategy for each specification. Since a specification and the generated strategies are very closely tied together, it is convenient to merge the tools for translating from DisCo to PVS and for generating proof strategies. Proof strategies for a specification are normally generated at the same time as the specification is translated to PVS.

We have not attempted to write decision procedures in the sense that the generated strategies would try automatically to prove some subgoals. Rather, the strategies simplify the proof up to a point where the subgoals are ready for the user to take control. The user can of course use the PVS strategy language to advice PVS to apply some strategy to each of the subgoals before resorting to interactive proof. For example, when verifying an invariant involving large amounts of propositional logic, one could start the proof with the command

(then* (*generated strategy*) (bddsimp))

which would first construct the top level of the proof using the generated strategy, and try to resolve each of the resulting subgoals using the *bddsimp* strategy of PVS. Any subgoals not discharged by *bddsimp* would then be presented to the user for interactive proof.

4.1 Example

Consider verifying the assertion *pos_w* in DisCo specification *loadBalance*. In order to verify that *pos_w* is an invariant, we need to establish that it holds in the initial state, and that the actions of the specification preserve it.

Figure 5 shows the generated top level strategy. The strategy first introduces the invariant rule to the sequent and instantiates it suitably (lines 2 and 3). The proof is then split into two parts: the initial condition (lines 7 and 8) and the preservation of *pos_w* (line 9 onward). The former is trivially taken care of by expanding the definition of *INIT* (the DisCo semantics includes all assertions as conjuncts in the initial condition), and the latter by introducing lemmas and instantiating them suitably.

Figure 6 shows the strategy generated for establishing that *pos_w* is preserved by action *newJob*. Lines 2 to 14 set the stage for the actual proof by expanding definitions and tidying up the sequent. When the strategy is applied to lemma *newJob_preserves_pos_w* and control has reached line 14 of the strategy, the current sequent is

```
[-1]    ASSUMPTIONS
[-2]    INIT(b!1(0))
[-3]    [] action2temporal(stutter(ACTIONS))
[-4]    (FORALL (cc: c): pos_w_body(cc, now))
{-5}    (EXISTS (c1: c):
         (newJob_guard(c1, now))
             AND
           ((w(c1)(next) = 1 + w(c1)(now))
                AND
             ((FORALL (other: c):
                 ((other) /= (c1)) IMPLIES
                    (w(other)(next) = (w(other)(now)))))))
  |-------
[1]    (FORALL (cc: c): pos_w_body(cc, next))
```

The existential quantification in formula -5 and the universal quantification in formula 1 are then skolemized by the steps in lines 18 and 22.

When the work was being carried out, the only way to refer to formulas in PVS was by their number in the current sequent. Since we did not want to try to predict this number, we include the formulas as strings in the generated strategies. Each string is parsed and type checked when the strategy is run, and the resulting formula is compared for equality with each formula in the sequent. This is done by escaping to Common Lisp from the PVS strategy language.

```
(defstep pos_w_is_invariant ()
  (then* (lemma "invariant_rule")
         (repeat (inst?))
         (split)
         (skosimp*)
         (branch (split)
                 ((then* (expand "INIT")
                         (ground))
                  (then* (skosimp*)
                          (lemma "newJob_preserves_pos_w")
                          (expand "preserves")
                          (inst -1 "b!1")
                          (split -1)
                          (inst -1 "n!1")
                          (lemma "balance_preserves_pos_w")
                          (expand "preserves")
                          (inst -1 "b!1")
                          (split -1)
                          (inst -1 "n!1")
                          (expand "ACTIONS")
                          (bddsimp)
                          (ground)))))
  "Top level strategy."
  "Top level strategy.")
```

Fig. 5. Top level strategy for *pos_w*

Next, the proof is split into two cases depending on whether the two skolem constants just introduced denote the same value or not. In DisCo terms, we are investigating separately the object that participates in the action, and those that do not participate. This is done by the steps in lines 26 and 27. Figures 7 and 8 show the details suppressed in lines 28 and 29.

Figure 7 shows the strategy fragment taking care of participating objects. First, formula -4 is instantiated by line 1 in the strategy. Line 2 expands the definition of *pos_w_body*. At line 4, the equality $w(c1)(next) = 1 + w(c1)(now)$ is used as a rewrite rule, line 5 removes the now redundant universally quantified antecedent, and finally the guard of the action is expanded at line 6.

After this part of the strategy has been run, the current sequent is

```
[-1]    cc = c1
[-2]    ASSUMPTIONS
[-3]    INIT(b!1(0))
[-4]    [] action2temporal(stutter(ACTIONS))
[-5]    w(c1)(now) >= 0
{-6}    w(c1)(now) < 10
  |-------
[1]    1 + w(c1)(now) >= 0
```

which the decision procedures of PVS recognize as valid.

The strategy fragment for nonparticipating objects (Figure 8) is similar. In this branch of the proof, the equality to be used as a rewrite rule is contained within a universal quantification, so it must be suitably instantiated before use.

This part of the strategy produces the sequent

```
[-1]    (w(cc)(next) = (w(cc)(now)))
[-2]    ASSUMPTIONS
[-3]    INIT(b!1(0))
[-4]    [] action2temporal(stutter(ACTIONS))
```

```
(defstep newJob_preserves_pos_w ()
  (then* (expand "preserves")
     (skosimp*)

     ;; introduce names now and next for the unprimed
     ;; and primed states
     (name "now" "b!1(n!1)") (replace -1 *) (hide -1)
     (name "next" "b!1(n!1+1)") (replace -1 *) (hide -1)

          ;; expand the definitions of the assertion
      ;; and the action
      (expand "pos_w") (expand "newJob") (flatten)

      (then*

       ;; introduce skolem constant c1 for the
       ;; participant c1
       (skolemize-action ("c1")) (flatten)

       ;; introduce skolem constant cc for the
       ;; quantified variable in primed pos_w
       (skolemize "cc" nil "(FORALL(cc:c):
                                      pos_w_body(cc,next))")

       ;; consider separately the cases for an object that
       ;; participates or does not participate in the action
       (spread
        (case-replace "cc = c1")
        ( handle participating objects
          handle all other objects))))
"Automatically generated strategy for proving:
         pos_w and newJob => pos_w'"
"Try to prove: pos_w and newJob => pos_w'")
```

Fig. 6. Strategy to establish that *newJob* preserves *pos_w*

```
(then* (pdp-instantiate-not "c1" nil
                "(FORALL(cc:c):pos_w_body(cc,now))")
       (expand "pos_w_body")
       (flatten)
       (object-replace "c1" "w" t)
       (hide-quantifications)
       (expand "newJob_guard"))
```

Fig. 7. Strategy fragment for participating objects

```
(then* (pdp-instantiate-not
        "cc" nil
        "(FORALL(cc:c):pos_w_body(cc,now))")
       (pdp-instantiate-not
        "cc" nil
        "(FORALL(other:c):((other)/=(c1))
              IMPLIES(w(other)(next)= (w(other)(now))))")
       (spread (split)
            ((then* (expand "pos_w_body")
                    (flatten)
                    (object-replace "cc" "w" t)
                    (hide-quantifications)
                    (expand "newJob_guard"))
             (ground))))
```

Fig. 8. Strategy fragment for nonparticipating objects

```
{-5}   (FORALL (cc: c): w(cc)(now) >= 0)
{-6}   w(cc)(now) >= 0
[-7]   (newJob_guard(c1, now))
[-8]   (w(c1)(next) = 1 + w(c1)(now))
[-9]   ((FORALL (other: c):
         ((other) /= c1) IMPLIES
           (w(other)(next) = (w(other)(now)))))
  |-------
[1]   cc = c1
{2}   w(cc)(now) >= 0
```

which is immediately recognized as valid by the decision procedures (this sequent is not even shown to the user).

4.2 Removing Quantifications

The strategy generator uses a simple approach to produce ground formulas from the quantified formulas in a sequent. Let

$$\mathcal{Q}_1 o_1 : \mathcal{Q}_2 o_2 : \ldots \mathcal{Q}_n o_n : P(o_1, \ldots, o_n) \tag{3}$$

be a DisCo assertion where each $\mathcal{Q}_i$ denotes either universal or existential quantification and each o_i denotes a formal name. A DisCo action maps to an existentially quantified formula of the form

$$\exists p_1, \ldots, p_m : \mathcal{A}(p_1, \ldots, p_m). \tag{4}$$

Consider the proof obligation

$$\begin{array}{l} [-1]\ \mathcal{Q}_1 o_1 : \mathcal{Q}_2 o_2 : \ldots \mathcal{Q}_n o_n : P(o_1, \ldots, o_n) \\ [-2]\ \exists p_1, \ldots, p_m : \mathcal{A}(p_1, \ldots, p_m) \\ \vdash \\ [1]\quad \mathcal{Q}_1 o_1 : \mathcal{Q}_2 o_2 : \ldots \mathcal{Q}_n o_n : P'(o_1, \ldots, o_n) \end{array} \tag{5}$$

where P' stands for the value of P in the next state. Our ultimate goal is to use the equations within $\mathcal{A}$ to rewrite P' in terms of unprimed variables. We can, however, only do this for ground formulas. The problem then is to perform a series of skolemizations and instantiations to produce suitable ground formulas in the sequent.

Skolemizing the action in (5) leaves us with

$$\begin{array}{l} [-1]\ \mathcal{Q}_1 o_1 : \mathcal{Q}_2 o_2 : \ldots \mathcal{Q}_n o_n : P(o_1, \ldots, o_n) \\ [-2]\ \mathcal{A}(p_1, \ldots, p_m) \\ \vdash \\ [1]\quad \mathcal{Q}_1 o_1 : \mathcal{Q}_2 o_2 : \ldots \mathcal{Q}_n o_n : P'(o_1, \ldots, o_n) \end{array} \tag{6}$$

where we have used the names $p_1, \ldots, p_n$ of the quantified variables as names of the skolem variables.

We can now skolemize either formula [-1] or formula [1], depending on whether the outermost quantification is existential or universal. For the sake of illustration, let us assume that o_1 is existentially quantified. After skolemization we get the following sequent:

$$\begin{array}{l} [-1]\ \mathcal{Q}_2 o_2 : \ldots \mathcal{Q}_n o_n : P(o_1, \ldots, o_n) \\ [-2]\ \mathcal{A}(p_1, \ldots, p_m) \\ \vdash \\ [1]\quad \exists o_1 : \mathcal{Q}_2 o_2 : \ldots \mathcal{Q}_n o_n : P'(o_1, \ldots, o_n). \end{array} \tag{7}$$

where o_1 is a new skolem constant.

Having introduced o_1, we can now instantiate formula [1] with it. This gives us

$$\begin{array}{l} [-1]\ \mathcal{Q}_2 o_2 : \ldots \mathcal{Q}_n o_n : P(o_1, \ldots, o_n) \\ [-2]\ \mathcal{A}(p_1, \ldots, p_m) \\ \vdash \\ [1]\quad \exists o_1 : \mathcal{Q}_2 o_2 : \ldots \mathcal{Q}_n o_n : P'(o_1, \ldots, o_n) \\ [2]\quad \mathcal{Q}_2 o_2 : \ldots \mathcal{Q}_n o_n : P'(o_1, \ldots, o_n) \end{array} \tag{8}$$

Ignoring formula [1] for the moment, we have now effectively removed the outermost quantification from the original formulas. Repeating this process finally results in ground instances of P and P'. The ground instances of P' can then be rewritten using the equations in the action.

If further skolemizations introduce skolem constants with the same type as o_1, formula [1] can be instantiated with them, possibly facilitating new skolemizations. The skolem constants $p_1, \ldots, p_n$ introduced when skolemizing the action can also be used for instantiating formulas.

This skolemization–instantiation process either continues forever, or it terminates with all possible combinations of skolem constants for P and P'. It is easy to construct an example where our algorithm loops forever: any assertion containing both a universal and an existential quantification over the same type results in a loop.

This kind of looping is easily detected when generating strategies. When a skolem constant is being introduced, the strategy generator tries to use the name of the quantified variable. If a skolem constant with the same name already exists, the generator prompts the user for a fresh name. A loop thus manifests itself as constant prompting for fresh names. Such a failure of the strategy generator suggests that a straightforward proof attempt using the same ideas will also fail.

5 Discussion

The main reason we chose to use PVS was the high degree of low level automation, as our proofs mostly consist of propositional logic and simple arithmetic reasoning. We were not willing to give up the low level automation for a more expressive strategy language.

We compensate for the limited expressiveness of the PVS strategy language by performing the required computations beforehand. The effects of the computation are carried out when the generated strategy is executed. The approach is not very elegant, but it demonstrates that in some cases it is possible to use offline computation to overcome the constraints of a restricted strategy language.

The largest case study done with the system is a distributed communication protocol [5, 11]. The protocol implements a token ring over a shared bus, and it recovers from station failures by removing failed stations from the ring. We verified an invariant stating that all the active stations agree on which stations are still in the ring, with the exception of those stations that have not yet received a broadcast message updating the list of active stations.

Verifying the invariant took approximately one and a half hours run time on a Sparc Server 670 MP with 114MB memory. Four subgoals were left for the user to handle, all of them requiring instantiation of an existentially quantified variable. Completing these subgoals took about twenty minutes, bringing the elapsed time to around two hours. This does not include the time spent in finding the proof for the four subgoals.

The major problem with our approach is the size of the generated strategies. The size of the specification for the case study is little over 500 lines of DisCo, which maps to about 7kB of unformatted, uncommented PVS. The specification consists of nineteen actions. The strategies generated for this specification are approximately 100kB in size. The size could be decreased significantly by using the formula labeling feature present in later versions of PVS, but still the size of strategies would be problematic for larger specifications.

A much better approach would be to generate strategies on the fly, by escaping to Common Lisp to construct strategies to be executed by PVS as strategy language forms. However, in constructing the strategies one might require information not readily available in the current sequent (e.g. the set of skolem constants denoting participants of the action). This information would need to be passed down into subgoals.

Unfortunately the strategy generator is obsolete, as it depends on a now abandoned version of the DisCo compiler. It would not be technically difficult to write a new back end for the current DisCo compiler to produce PVS, more difficult would be to obtain academic funding for the work.

6 Conclusions

We have described how invariant proofs of DisCo specifications were partly automated by generating PVS strategies. Each generated strategy contains the proof commands needed to carry out the top levels of a proof that a specific action preserves a specific invariant.

A user can instruct PVS first to run a generated strategy, and then to apply e.g. *ground* or *grind* to each of the subgoals produced by the strategy. Only the subgoals not resolved by PVS are then presented to the user for interactive proving.

The work demonstrates that strategy generation can be of practical help in verification. However, the size of the generated strategies suggests that strategy generation should occur on the fly rather than offline.

References

1. The DisCo project WWW page. At http://disco.cs.tut.fi on the World Wide Web, 2003.
2. R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, October 1988.
3. Michael J. C. Gordon. HOL: A proof generating system for higher-order logic. In Graham Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Boston Kluwer Academic Publishers, 1988.
4. H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proceedings of the 12th International Conference on Software Engineering*, pages 63–71. IEEE Computer Society Press, 1990.
5. Pertti Kellomäki. Analysis of a stabilizing protocol. Licentiate of Technology thesis, Tampere University of Technology, 1994. http://www.cs.tut.fi/~pk/papers.html.
6. Pertti Kellomäki. *Mechanical Verification of Invariant Properties of DisCo Specifications*. PhD thesis, Tampere University of Technology, 1997.
7. Pertti Kellomäki. Verification of reactive systems using DisCo and PVS. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in Lecture Notes in Computer Science, pages 589–604. Springer–Verlag, 1997.
8. Reino Kurki-Suonio. Fundamentals of object-oriented specification and modeling of collective behaviors. In H. Kilov and W. Harvey, editors, *Object-Oriented Behavioral Specifications*, pages 101–120. Kluwer Academic Publishers, 1996.
9. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

10. Sam Owre, John Rushby, Natrajan Shankar, and Friedrich von Henke. Formal verification of fault-tolerant architecures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
11. Leo Sintonen. Event driven bus architecture for bounded area networks. In *Proceedings of the 16th Annual Conference of IEEE Industrial Electronics Society*, pages 539–541, Pacific Grove, California, November 27–30 1990.

Developing Strategies for Specialized Theorem Proving about Untimed, Timed, and Hybrid I/O Automata *

Sayan Mitra[1] and Myla Archer[2]

[1] MIT Laboratory for Computer Science,
200 Technology Square, Cambridge, MA 02139
mitras@theory.lcs.mit.edu,
[2] Code 5546, Naval Research Laboratory,
Washington, DC 20375
archer@itd.nrl.navy.mil

Abstract. In this paper we discuss how we intend to develop a specialized theorem proving environment for the Hybrid I/O Automata (HIOA) framework [7] over the PVS [11] theorem prover, and some of the issues involved. In particular, we describe approaches to using PVS that allow and encourage the development of useful proof strategies, and note some desired PVS features that would further help us to do so for our HIOA environment.

1 Introduction

Interest in specialized theorem proving environments has emerged from various application domains [3,4,6, 1]. A major motivation for developing such environments is to relieve the developers and verification engineers from mastering the specification language and the proof commands of a general theorem prover. Specialized environments also help expert users of theorem provers by replacing repetitive proof patterns with strategies, and by making it possible to generate human readable proofs.

We plan to develop a specialized theorem proving environment to be used with the Hybrid I/O Automata (HIOA) framework. HIOA is a very general framework for modeling systems with both discrete and continuous behavior, and subsumes both the timed and untimed I/O automata models. Therefore, any strategies and metatheories for HIOA would be applicable to timed and untimed I/O automata as well. A theory template for specifying HIOAs has been presented in [9]. This formalization of HIOA in PVS is similar to the formalization of Lynch-Vaandrager (LV) timed automaton [8] in the Timed Automaton Modeling Environment (TAME) [1]. However, important differences arise in the two formalizations because LV-timed automata communicate via shared actions alone, whereas HIOAs also communicate via shared variables. Therefore, the evolution of continuous variables is modeled in TAME using time passage actions to capture cumulative changes over an interval, while in the HIOA model, the evolution of the continuous state variables over time is modeled using trajectories. Our HIOA environment must allow for these differences.

The rest of this paper is organized as follows: In Section 2 we discuss the main types of proofs which will be supported by our HIOA environment and the design issues involved in developing proof strategies for each type. In Section 3 we suggest certain new features of PVS which would aid the development of strategies for PVS. Finally, we summarize and conclude in Section 4.

2 Supported Proof Types

Apart from simplifying direct proofs of properties, the HIOA proving environment will provide special strategies for mechanizing inductive invariant proofs and abstraction (e.g., simulation) proofs for timed, hybrid and untimed I/O automata. Apart from TAME, another theorem proving environment has been developed, based on Isabelle, which mechanizes invariant proofs for I/O automata [10]. In [2], the authors present a simulation proof of a leader election protocol in PVS. However, we have not come across any work which addresses the development of strategies for mechanizing simulation proofs.

* Funding for this research has been provided by ONR

2.1 Inductive Proofs

The approach we intend to take for supporting inductive invariant proofs is derived from the Timed Automaton Modeling Environment (TAME) [1]. As in TAME, we will develop a parameterized theory `machine` which defines the reachable states of an automaton in terms of its states, initial states, actions and (in case of hybrid I/O automata) activities [9]. This theory will also establish the theorem that allows proving invariants inductively. We will also develop a general theory template which can be instantiated with particular state variables and actions (optionally, activities) to obtain an *automatonName*`_decls` theory describing the automaton. The *automatonName*`_decls` theory will import an instance of the theory `machine` with the declared states and actions as parameters. Instantiation of the theory `machine` defines reachability and the induction theorem for the particular automaton. All the invariants and the associated lemmas of an automaton will be collected and proved in a theory named *automatonName*`_inv`.

The advantages of this (TAME) approach are as follows: (1) It is possible to write generic strategies which work for all automata specified using the template. The strategies for induction are tailored for the defined automaton template, and are defined in the file `pvs-strategies`. Therefore, (2) the user can use the specialized environment from within the PVS system. Finally, (3) it is easy to generate human readable proofs using the generic strategies, provided that the strategies implement proof steps meaningful to humans.

A slightly different approach has been taken by the developers of DisCo [6, 5], where the PVS specification of the automaton is processed by a "generator" to produce the proof scripts. One advantage of this approach, due to the clearly defined interface between the theorem prover (PVS) and the specialized environment (DisCo), is that the generated proof scripts are relatively insensitive to the modifications of the internals of theorem prover commands and data structures.

However, we would like our strategies to be directly applicable to all automata specified with our template theory. The success of our approach does depend on access to the data-structures in the proof state maintained by PVS, and the consistency of the behavior of PVS proof commands. We discuss the PVS support necessary to achieve this in Section 3.

2.2 Abstraction Proofs

Given automata A and C, it is often useful for the purposes of verification to show that there exists an *abstraction relation* between them. Several kinds of abstraction relation, e.g., homomorphism, refinement, forward and backward simulation, etc., are described in the literature, and there may also be other such relations of interest.

Abstraction proofs can be performed directly by specifying both automata A and C, and the abstraction relation between them, within the same PVS theory. However, this approach makes it difficult to construct generic strategies for automating the proofs, and to use invariants which have been proved separately for the individual automata.

Instead, we intend to make use of PVS support for *theory parameters*, as follows. Two parameters `A` and `C` of the type `automaton` theory (Figure 1) can be passed as parameters to the theory `abstraction` (Figure 2), which also takes the abstraction relation `absrel` and the action map `actmap` as parameters. The theory `abstraction`, which somewhat resembles the theory `group_homomorphism` in [12] for setting up proofs of homomorphism between groups, defines the abstraction relations between the two interpretations of the `automaton` theory. To pass actual theory parameters to `group_homomorphism`, the various elements of the group theories must be named: the members of the groups, identities and composition operators, etc. But, when individual automata follow the same naming conventions as in the theory `automaton`, a shortcut is in principle possible in passing actual theory parameters to `abstraction`: because the various elements of the actual parameters can be matched to the formal parameters *syntactically*, only the actual theory names need to be provided. A modification to PVS that will allow this shortcut is under construction at SRI.

The `actmap` relation in the theory `abstraction` maps an action of the concrete automaton C to an action of the abstract automaton A. The axioms `vis_ax` and `invis_ax` that indicate that the visible actions in C map to visible actions in A and invisible (i.e., internal) actions in C map to the stutter step in A, become proof obligations when `abstraction` is instantiated. At the same time, the axioms `stutter_trans_ax` and `stutter_enabled_ax` from the theory `automaton` will become proof obligations with respect to both `automaton` theory instances.

```
automaton: THEORY

BEGIN
   actions : TYPE+;
   stutter: actions;
   visible (a:actions) : bool;

   states : TYPE+;

   start (s:states) : bool;
   enabled (a:actions, s:states) : bool;
   trans (a:actions, s:states) : states;

   stutter_trans_ax: AXIOM (FORALL (s:states): (trans(stutter,s) = s));
   stutter_enabled_ax: AXIOM (FORALL (s:states): (enabled(stutter,s)));

   reachable (s:states) : bool;
   equivalent (s1, s2: states) : bool;
END automaton
```

Fig. 1. The automaton abstract theory

```
abstraction [ A, C: automaton,
              actmap: [C.actions -> A.actions],
              absrel: [C.states, A.states -> bool] ] : THEORY
  BEGIN
    a_C : VAR C.actions;
    a_A : VAR A.actions;
    s_C, s1_C, s2_C: VAR C.states;
    s_A : VAR A.states;

    vis_ax: AXOIM
      (FORALL a_C: C.visible(a_C) => A.visible(actmap(a_C)));

    invis_ax: AXIOM
      (FORALL a_C: NOT(C.visible(a_C)) => (actmap(a_C) = A.stutter));

    weak_refinement_base : bool =
      (FORALL s_C, s_A:
         C.start(s_C) & absrel(s_C, s_A)
          => A.start(s_A));

    weak_refinement_step : bool =
      (FORALL s_C, s1_C, a_C, s_A:
         C.reachable(s_C) &
         C.equivalent(s_C, s1_C) & C.visible(a_C) & C.enabled(a_C, s1_C) &
         A.reachable(s_A) &
         absrel(s1_C, s_A)
          => A.enabled(actmap(a_C), s_A) &
             (EXISTS (s2_C: C.states):
                C.equivalent(C.trans(a_C, s1_C), s2_C) &
                absrel(s2_C, A.trans(actmap(a_C), s_A))));

    weak_refinement : bool = weak_refinement_base &.weak_refinement_step;
  END abstraction
```

Fig. 2. The abstraction theory

For abstraction proofs the theory `abstraction` assumes a role analogous to that of the theory `machine` in the case of induction proofs, in that it will define the abstraction relations and also establish the theorems (e.g., concerning trace inclusion) that are the consequences of the existence of such relations between pairs of automata. In Figure 2, only one sort of refinement relation has been defined; in practice, the theory `abstraction` will define all possible useful abstraction relations between the two automata. The theory `abstraction` will thus provide us with a starting point for developing generic strategies for proving abstraction relations.

3 PVS Support

In this section we suggest some PVS features which would be helpful for writing strategies, particularly for the above types of proofs.

1. **Naming in theory interpretations.** The abstraction proofs involve many related theories, for example different instances of *automatonName*_`decl`, *automatonName*_`inv`, `machine`, etc. It is difficult to write general strategies that involve formulas or definitions in multiple theories: the user often has to identify the particular theory instances explicitly. It would be useful for strategy writers if PVS provided well documented naming conventions and functions for determining theory instances associated with names, and supported the automatic context-based selection by user strategies of appropriate theory instances for names.
2. **Functions to access information in specification and in proof states.** A strategy often depends on the nature of the automaton specification. It can also make choices based on the current proof state. The CLOS structure used by PVS provides functions to access various slots of the current proof state object. However, these are not guaranteed to be fixed, and indeed can sometimes change dynamically. For writing strategies it would be helpful if functions to access the definitions in a particular theory—for example the invariance lemmas or the action definitions—and functions for accessing parts of a sequent, formulae, etc., were provided as a part of a PVS strategy language.
3. **Documentation of implementation details in PVS proof commands.** The LISP/CLOS functions used in writing the internal PVS strategies (e.g., `induct`) are not well documented. Many of these functions, for example `typep`, `tc-eq`, can be useful for writing new strategies. Therefore, proper documentation of these functions would save effort and help new strategy writers learn the art.
4. **Improved support for maintaining compatibility with PVS.** The effects of some basic PVS commands (e.g. `SKOLEM, EXPAND`) have altered over PVS versions owing largely to changes in PVS's decision procedures and their use in conjunction with such basic steps. As a result, strategies developed for older versions of PVS do not always work in the newer PVS versions. Therefore, it is highly desirable to provide a feature in future versions of PVS that would allow strategies to invoke prover commands and get the same result as in some specified previous version. Because most changes in effects appear to involve the decision procedures and their hidden uses, there should at the very least be optional versions of proof steps that decouple them from any use of these procedures.

4 Conclusions

Domain specific theorem proving is a practical means for harnessing the power of mechanical theorem provers for system design and analysis. In this paper we have outlined design principles for the development of proof strategies of a specialized theorem proving environment for hybrid I/O automata based on PVS. Our aim is to make the more complex component of the environment—the proof strategies—generic, based on a specific HIOA template, leaving the simpler component—the specification—to be written by instantiating the template. We have outlined the support we believe would help us develop effective generic strategies.

Acknowledgements

We wish to thank John Rushby and Natarajan Shankar of SRI for helpful discussions about our plans for a framework supporting generic strategies for abstraction relations between automata. We thank Sam Owre and Natarajan Shankar for undertaking enhancements to PVS that will support our plans. We also thank Nancy Lynch of MIT for helpful discussions and her comments about the design of the specification language for HIOA.

References

1. Myla Archer. TAME: PVS Strategies for special purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1/4), February 2001.
2. M. Devillers, D. Griffioen, J. Romijn, and F. Vaandrager. Verification of a leader election protocol—formal methods applied to IEEE 1394. *Formal Methods in System Design*, 16(3):307–320, June 2000.
3. Urban Engberg. *Reasoning in the Temporal Logic of Actions - The Design and Implementation of an Interactive Computer System.* PhD thesis, University of Aarhus, Denmark, 1995.
4. S. Kalvala. A Formulation of TLA in Isabelle. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 971, pages 214–228, Aspen Grove, Utah, USA, 1995. Springer-Verlag.
5. Pertti Kellomäki. Mechanizing invariant proofs of joint action systems. In *Proceedings of the Fourth Symposium on Programming Languages and Software Tools*, pages 141–152, Visegrad, Hungary, June 1995.
6. Pertti Kellomäki. Mechanical verification of DisCo specifications. In *Israeli-Finnish Binational Symposium on Specification, Development, and Verification of Concurrent Systems*, Technion, Haifa, January 1996.
7. Nancy Lynch, Roberto Segala, and Frits Vaandraager. Hybrid I/O automata. To appear in *Information and Computation.* Also, Technical Report MIT-LCS-TR-827d, MIT Laboratory for Computer Science Technical Report, Cambridge, MA 02139, January 13, 2003. `theory.lcs.mit.edu/tds/papers/Lynch/HIOA-final.ps`.
8. Nancy Lynch and Frits Vaandrager. Forward and backward simulations - part ii: Timing-based systiems. *Information and Computation*, 128(1):1–25, July 1996.
9. Sayan Mitra. HIOA+: Specification language and proof tools for hybrid systems, 2003. Submitted for publication, http://theory.lcs.mit.edu/ mitras/research/LCPTHIOA.ps.
10. Olaf Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory.* PhD thesis, Technische Universität München, Sept. 1998.
11. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.
12. S. Owre and N. Shankar. Theory interpretations in PVS. Technical report, Computer Science Lab., SRI Intl., Menlo Park, CA, 2001.

Author Index

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704–0188*

1. REPORT DATE *(DD-MM-YYYY)* 01-09-2003	**2. REPORT TYPE** Conference Publication	**3. DATES COVERED** *(From - To)*

4. TITLE AND SUBTITLE Design and Application of Strategies/Tactics in Higher Order Logics	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Myla Archer, Ben Di Vito, and César Muñoz (Editors)	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER 23-704-03-50
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Langley Research Center Hampton, VA 23681-2199	**8. PERFORMING ORGANIZATION REPORT NUMBER** L–18328
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	**10. SPONSOR/MONITOR'S ACRONYM(S)** NASA
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) NASA/CP–2003–212448

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 59
Availability: NASA CASI (301) 621-0390 Distribution: Standard

13. SUPPLEMENTARY NOTES
An electronic version can be found at http://techreports.larc.nasa.gov/ltrs/ or http://ntrs.nasa.gov.

14. ABSTRACT
STRATA 2003, the First International Workshop on Design and Application of Strategies/Tactics in Higher Order Logics, was held in Rome, Italy on September 8, 2003. This event was a satellite workshop of the 16th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2003. Papers were solicited on all aspects of development and application of user-defined strategies or tactics in higher order logic theorem provers. A particular emphasis was placed on contributions that describe either experience with PVS strategies or comparisons with strategy and tactic development in other theorem proving systems. Included was an invited paper from the staff of SRI International, the developers of the PVS tools. Also included was the text of a PVS strategy-writing tutorial. During the workshop, each paper was presented by one of its authors.

15. SUBJECT TERMS
Formal Methods, Theorem Proving, Proof Strategies

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STI Help Desk (email: help@sti.nasa.gov)
U	U	U	UU	118	**19b. TELEPHONE NUMBER** ***(Include area code)*** (301) 621-0390