PVS#: Streamlined Tacticals for PVS [1]

Florent Kirchner [2]

Laboratoire d'Informatique de l'École Polytechnique
91128 Palaiseau Cedex, France

César Muñoz [3]

National Institute of Aerospace
Hampton VA 23666, USA

Abstract

The semantics of a proof language relies on the representation of the *state* of a proof after a logical rule has been applied. This information, which is usually meaningless from a logical point of view, is fundamental to describe the control mechanism of the proof search provided by the language. In this paper, we propose a datatype, called *proof monad*, to represent the state information of a proof and we illustrate its use in PVS. Furthermore, we show how this representation can be used to design a new set of powerful PVS tacticals that have simple and clear semantics. The implementation of these tacticals is called PVS#.

Key words: Monads, Proof languages, Tactics, Tacticals, Strategies, PVS.

1 Introduction

The representation of mathematical proofs has been an active research topic in computer science since the early 1970's, when the first theorem provers were designed. Several representations of the proof process have been proposed, from the simple collection of logical formulas [2] to typable lambda-terms (thanks to the Curry-De Bruijn-Howard isomorphism), where open terms are used to handle incomplete proofs [8, 9, 4]. However, as mechanical theorem

[1] This work was supported by the National Aeronautics and Space Administration at Langley Research Center under the Research Cooperative Agreement No. NCC-1-02043 awarded to the National Institute of Aerospace.
[2] Email: `florent.kirchner@inria.fr`
[3] Email: `munoz@nianet.org`

proving picked pace and proofs grew in complexity, the need for more involved ways to control their construction spawned larger and more refined proof languages.

In proof assistants such as Coq [3] and PVS [10], the proof language contains two kinds of instructions: *tactics*, which modify the proof tree by applying logical rules, and *tacticals*, which provide proof search control. In this work, we are mainly interested in tacticals and their semantics. We note that the words 'tactic' and 'tactical' are inherited from the first procedural theorem prover LCF. In PVS, tactics are called *proof rules* and tacticals are called *strategies*. For simplicity, we use the original LCF terminology.

A tactical is a tactic combinator whose behavior depends on the outcome of its tactic arguments. The outcome of a tactic usually contains a state information such as *success* or *failure* that signals whether the tactic has solved the current goal or has failed. A sophisticated proof language, such as the language of PVS or Coq, uses many other types of state information. For instance, consider the PVS tactical `try` that is at the same time a conditional and a backtracking combinator. Let $|.|$ be a semantic evaluator. The semantics of `try` [1] need five different types of state information: *failure*, *success*, *idem*, *subgoals*, *backtrack*.

$$|\texttt{(try A B C)}| = \begin{cases} |\texttt{C}| & \text{if } |\texttt{A}| \in \{\mathit{idem}, \mathit{backtrack}\} \\ |\texttt{A}| & \text{if } |\texttt{A}| \in \{\mathit{failure}, \mathit{success}\} \\ \mathit{backtrack} & \text{if } |\texttt{A}| = \mathit{subgoals}, \\ & \quad\; |\texttt{B}| \in \{\mathit{failure}, \mathit{backtrack}\} \\ \mathit{subgoals} & \text{if } |\texttt{A}| = \mathit{subgoals}, \\ & \quad\; |\texttt{B}| \in \{\mathit{idem}, \mathit{subgoals}\} \\ \mathit{success} & \text{if } |\texttt{A}| = \mathit{subgoals}, \\ & \quad\; |\texttt{B}| = \mathit{success}\ , \end{cases}$$

where

$$|\texttt{(skip)}| = \mathit{idem}$$
$$|\texttt{(fail)}| = \mathit{failure}\ .$$

In a previous attempt to formalize the semantics of the PVS proof language [5], the state of a proof was recorded by flags that were plainly added to the representation of the proof tree. In this paper, we show how the proof state information can be elegantly modeled by a monadic datatype, which is called *proof monad*.

The rest of this paper is organized as follows. In Section 2, we give an overview of monads and provide a description of PVS's proof state the form of a proof monad. In Section 3, we propose a simplification of this framework that still obeys the monadic laws. This simplification is the base of the PVS#

proof language that is presented in Section 4.

2 The PVS Proof Monad

Monads were originally used to provide imperative features, such as side effects, to pure functional languages [11]. The basic idea is to view a program, not as a pure function e.g. from A to B, but as a morphism from values A to computations $\mathcal{M}B$, where $\mathcal{M}B$ represents the conjunction of side-effects and a return value, which is of type B. *Monadic operators* are systematically associated to that datatype and provide a way to build and combine programs. The monad we present here, called *proof monad*, allows us to describe PVS tactics as functions over the proof state.

Assume that X is the type of the proof tree objects, and x is an inhabitant of X, i.e., a proof tree variable. We define PVS tactics as functions of type $X \to \mathcal{M}X$, where $\mathcal{M}X$ denotes the following inductive datatype:

$$\begin{aligned}\textbf{datatype } \mathcal{M}X &= \textit{success } x\\ &\mid \textit{subgoals } x\\ &\mid \textit{idem } x\\ &\mid \textit{backtrack } x\\ &\mid \textit{failure } x \ .\end{aligned}$$

Note that the type X is abstract to this formalization of proof monads; it can instantiated with any concrete representation of proof trees: typable open lambda-term, metavariable signature, collection of sequents, etc. Also remark that, as reflected by the semantics of the tactical `try`, the proof monad datatype has five constructors.

We proceed to introduce the following monadic operators:

- The function *unit*, of type $X \to \mathcal{M}X$, maps a proof tree into an element of the proof monad:

  $$\textit{unit } x = \textit{idem } x \ .$$

- The function $\star$, of type $\mathcal{M}X \to (X \to \mathcal{M}X) \to \mathcal{M}X$, provides a way to apply a tactic t to the proof tree resulting from the application of another tactic:

  $$m \star t = \begin{cases} !\ (t\ x) & \text{if } m = \textit{subgoals } x \\ t\ x & \text{if } m = \textit{idem } x \\ m & \text{otherwise} \ , \end{cases}$$

  where ! is an auxiliary function, of type $\mathcal{M}X \to \mathcal{M}X$, defined as:

$$! \; m = \begin{cases} \mathit{subgoals}\; x & \text{if } m = \mathit{idem}\; x \\ m & \text{otherwise} \;. \end{cases}$$

In PVS, *unit* corresponds to the semantics of the tactical `skip`. The function $\star$ describes the semantics of a tactical that combines its arguments in sequence, applying tactic $n+1$ unless tactic n has raised an exception or proved the goal. The function ! makes sure that the state *subgoals* is not overwritten by the state *idem*, when tactics are applied in sequence.

3 A Simpler Proof Monad for PVS

The use of two different exception mechanisms (*backtrack* and *failure*) makes the semantics of the PVS tacticals quite complex. In particular, the widely used PVS tacticals `then`, `else`, `repeat`, `spread` are defined in terms of `try`, and their behavior with respect to failure propagation and backtracking is not easy to characterize.

We propose a stripped-down version of the PVS proof monad that has only three types of state information, including only one type of exceptions.

$$\begin{aligned} \textbf{datatype}\; \mathcal{M}X &= \mathit{success}\; x \\ &\mid \mathit{subgoals}\; n\; x \\ &\mid \mathit{exception}\; s\; x \;, \end{aligned}$$

where n is an integer denoting the number of subgoals, and s is a string allowing for exception naming.

Adapting the monadic operators to this structure is fairly simple, as illustrated by Fig. 1.

$$\mathit{unit}\; x = \mathit{subgoals}\; 0\; x \;,$$

$$m \star t = \begin{cases} !\; n\; (t\; x) & \text{if } m = \mathit{subgoals}\; n\; x \\ m & \text{otherwise} \;, \end{cases}$$

$$!\; n\; m = \begin{cases} \mathit{subgoals}\; n\; x & \text{if } m = \mathit{subgoals}\; 0\; x \\ m & \text{otherwise} \;. \end{cases}$$

Fig. 1. Monadic operators for the simple PVS monad

Based on this simplified proof monad, we propose in the next section a new set of PVS tacticals, called PVS#.

4 PVS#

PVS# is a new set of tacticals that replace the native backtracking and failure mechanisms provided by the PVS tacticals `try` and `fail`. The new set of tacticals features an error handling mechanism, based on *catch* and *throw*, typical of programming languages.

Tacticals in PVS# are simpler to combine as their semantics only require one type of state information for exceptions. Thus, the functionalities of `try` and `fail` have been split in three different tacticals: one tactical for throwing an exception (`#throw`), one tactical for catching an exception (`#catch`), and one tactical for testing progress (`#ifsubgoals`). PVS's basic tacticals defined via `try` and `fail`, such as `then`, cannot be combined with PVS# tacticals. For this reason, PVS# also provides tacticals for sequencing (`#then`), conditionals (`#when`), etc.

4.1 Exception Handling and Progress Testing

`(#throw tag)` This tactical returns the proof tree unchanged with the proof state set to *exception tag*.

`(#catch step1 &optional tag step2)` This tactical behaves as `step1` if `step1` does not raise an exception. Otherwise, if the result is an exception named `tag` then it evaluates `step2`. If `tag` does not correspond to the name of the exception, then the exception is propagated.
Usage: The proof script

```
(#catch (#throw "exn") "exn" (flatten))
```

will result in the evaluation of `(flatten)`, but

```
(#catch (#throw "div0") "exn" (flatten))
```

will propagate the exception named `"div0"`.

`(#ifsubgoal step step1 step2)` This tactical calls either `step1` or `step2`, depending on the progress of `step`. If `step` generates subgoals, then it applies `step1` to all the subgoals. Otherwise, it applies `step2`.
Usage: The proof script

```
(#ifsubgoal (flatten) (propax) (split))
```

applies `(flatten)` to the current goal. If the goal does simplify, then `(propax)` is applied to the resulting subgoal. Otherwise, `(split)` is applied to the current goal.

4.2 Other Tacticals

`(#then &rest steps)` Let `step1` be the first element of the list `steps`, and `rest-steps` be the remaining elements. This tactical first applies `step1` to the current goal, and then `(#then rest-step)` to all of the generated subgoals, if any, or to the original goal if `step1` had no effect. This tactical implements the semantics of the monadic operator $\star$.

`(#when condition &rest steps)` This tactical evaluates its first argument; if it results in `nil` then nothing is done and the result of this tactical is a `(skip)`. Otherwise, it applies `steps` in sequence using `#then`. The definition of `(#when condition steps)` is just `(if condition (#then steps) (skip))`.
Usage: The proof script

```
(#when (equal (get-goalnum *ps*) 1) (ground))
```

applies `(ground)` if the current goal is the first subgoal, otherwise it does nothing.

`(#first steps)` This tactical applies the first tactic in `steps` that does not raise an exception, if any. Otherwise, it does nothing.
Usage: The proof script

```
(#first (#throw "fault1") (bddsimp) (#throw "fault2"))
```

applies `(bddsimp)` to the current goal.

`(#solve steps)` This tactical applies the first tactic in `step` that proves the current goal, if any. Otherwise, it does nothing.
Usage: The proof script

```
(#solve (case "y > 0") (bddsimp))
```

tries to apply the case analysis command to the current goal, if it does not completely prove the current goal it applies `(bddsimp)`. If this tactic also fails to discharge the current goal, it does nothing.

4.3 Semantics

The semantics of PVS# tacticals has been fully formalized using the simplified proof monad described in Section 3. As an example, Fig. 2 shows the semantics of the tacticals of Section 4.1.

5 Conclusion

We have defined a representation of proof state information via proof monads, which is independent from the concrete representation of proof trees. This has allowed us to give a synthetic representation of the PVS proof state, and to formulate an arguably simpler alternative to this data structure.

Based on that representation, we have designed a new set of PVS tacticals, called PVS#, that has more typical semantics with respect to error handling. A preliminary prototype of PVS# is available at [7].

The topic of this paper is the subject of ongoing work, including, in particular, the development of new tacticals for PVS#, the meta-theoretical study of proof monads, and the extension of this formalism to other theorem provers. For instance, another implementation of the proof monad concept was already carried out in the Fellowship proof assistant [6]. In the long term, we believe

$$\texttt{\#throw}\ s\ x = \mathit{exception}\ s\ x\ ,$$

$$\texttt{\#catch}\ t_1\ s\ t_2\ x = \texttt{\#catch'}\ s\ t_2\ (t_1\ x)\ ,$$

$$\texttt{\#catch'}\ s\ t\ m = \begin{cases} t\ x & \text{if } m = \mathit{exception}\ s\ x \\ m & \text{otherwise}\ , \end{cases}$$

$$\texttt{\#ifsubgoals}\ t\ t_1\ t_2\ x = \texttt{\#ifsubgoals'}\ t_1\ t_2\ (t\ x)\ ,$$

$$\texttt{\#ifsubgoals'}\ t_1\ t_2\ m = \begin{cases} t_1\ x & \text{if } m = \mathit{subgoals}\ n\ x \text{ and } n \geq 1 \\ t_2\ x & \text{if } m = \mathit{subgoals}\ 0\ x \\ m & \text{otherwise}\ . \end{cases}$$

Fig. 2. The semantics of exception handlers and progress tests

that the concept of proof monad will play a central role in the design and semantics of proof languages for procedural theorem provers.

References

[1] Archer, M., B. Di Vito and C. Muñoz, *Developing user strategies in PVS: A tutorial*, in: *Proceedings of Design and Application of Strategies/Tactics in Higher Order Logics STRATA'03*, NASA/TP-2003-212448, NASA LaRC,Hampton VA 23681-2199, USA, 2003, pp. 16–42.

[2] Boyer, R. S. and J. S. Moore, "A Computational Logic," Academic Press, New York, 1979.

[3] The Coq Development Team, LogiCal Project, INRIA, "The Coq Proof Assistant: Reference Manual, Version 8.0," (2004).
URL coq.inria.fr/doc/main.html

[4] Jojgov, G. I., *Holes with binding power*, in: H. Geuvers and F. Wiedijk, editors, *TYPES*, Lecture Notes in Computer Science **2646** (2002), pp. 162–181.
URL
link.springer.de/link/service/series/0558/bibs/2646/26460162.htm

[5] Kirchner, F., *Coq tacticals and PVS strategies: A small-step semantics*, in: M. A. et al., editor, *Design and Application of Strategies/Tactics in Higher Order Logics* (2003), pp. 69–83.

[6] Kirchner, F., "Fellowship: who needs a manual anyway?" (2005).
URL www.lix.polytechnique.fr/Labo/Florent.Kirchner/fellowship

[7] Kirchner, F., "Programmation Tacticals," (2005).
URL research.nianet.org/fm-at-nia/Practicals/

[8] Magnusson, L., "The Implementation of ALF—A Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution," Ph.D. thesis, Chalmers University of Technology and Göteborg University (1995).

[9] Muñoz, C., *A calculus of substitutions for incomplete-proof representation in type theory*, Technical Report RR-3309, Unité de recherche INRIA-Rocquencourt (1997).

[10] Owre, S., J. M. Rushby and N. Shankar, *PVS: A prototype verification system*, in: D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, Lecture Notes in Artificial Intelligence **607** (1992), pp. 748–752.

[11] Wadler, P., *Monads for functional programming*, in: M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School* (1993).
URL citeseer.ist.psu.edu/wadler95monads.html