Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 174 (2007) 47–58

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

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 present a monadic datatype to represent the state information of a proof and we illustrate its use in the PVS theorem prover. We show how this representation can be used to design a new set of powerful tacticals for PVS, called PVS#, that have a simpler and clearer semantics compared to the semantics of standard PVS tacticals.

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

[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

doi:10.1016/j.entcs.2006.10.057

In proof assistants such as Coq [3] and PVS [10], the proof language contains two kinds of *proof commands*: *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 *state* of the proof after the application of its arguments. The state of a proof usually contains non-logical 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 languages of PVS and 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: $(\mathtt{try}\ \mathtt{t}_1\ \mathtt{t}_2\ \mathtt{t}_3)$ applies its first argument $\mathtt{t}_1$ to the goal, and if it generates subgoals, it applies $\mathtt{t}_2$ to the subgoals, else it applies $\mathtt{t}_3$. Furthermore, if $\mathtt{t}_2$ fails, for example, because $\mathtt{t}_2 = (\mathtt{fail})$, then it initiates a backtracking sequence, which is propagated until it is evaluated as the first member of another `try` construct, in which case it evaluates its third argument.

The semantics of `try` [1] need five different types of state information: *failure*, *success*, *skip*, *subgoals*, *backtrack*. Informally, if $|.|$ is a semantic evaluator, the semantics of `try` can be expressed as follows:

$$|(\mathtt{try}\ \mathtt{t}_1\ \mathtt{t}_2\ \mathtt{t}_3)| = \begin{cases} |\mathtt{t}_3| & \text{if } |\mathtt{t}_1| \in \{\mathit{skip}, \mathit{backtrack}\} \\ |\mathtt{t}_1| & \text{if } |\mathtt{t}_1| \in \{\mathit{failure}, \mathit{success}\} \\ \mathit{backtrack} & \text{if } |\mathtt{t}_1| = \mathit{subgoals}, \\ & \quad |\mathtt{t}_2| \in \{\mathit{failure}, \mathit{backtrack}\} \\ \mathit{subgoals} & \text{if } |\mathtt{t}_1| = \mathit{subgoals}, \\ & \quad |\mathtt{t}_2| \in \{\mathit{skip}, \mathit{subgoals}\} \\ \mathit{success} & \text{if } |\mathtt{t}_1| = \mathit{subgoals}, \\ & \quad |\mathtt{t}_2| = \mathit{success}\ , \end{cases}$$

where

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

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 simple monadic datatype. The datatype and its properties are defined in Section 2. In Section 3, we illustrate the application of this framework to the design of a new set of tacticals for PVS, which we call PVS#. Finally, the implementation of the monadic datatype in PVS# is described in Section 4.

2 A Monadic Datatype for Proof State Representation

Monads are a popular way to describe imperative features, such as side effects and exceptions, in functional programming languages [11]. The main idea is to view a program P, not as a pure function, e.g., from A to B, but as a morphism from values A to a datatype $\mathcal{M}B$, where $\mathcal{M}B$ represents the conjunction of side-effects in P and its return value, which is of type B. *Monadic operators*, that obey *monad laws*, are associated to the datatype and provide a way to build and compose programs.

In general, proof commands can be seen as functional programs that act on proof objects. However, proof assistants, such as PVS, also provide tacticals that are not purely functional, e.g., `fail` and `try`, which raise and catch exceptions, respectively. Furthermore, the effects of tacticals on the state of a proof may also be seen as side effects on the proof object. Based on these observations, we define a monadic datatype that allows us to give a denotational semantics of tacticals with imperative features. Strictly speaking, our datatype is not a monad as it is not fully polymorphic. This prohibits the "stacking" of monadic structures and the definition of mapping and joining operations. However, these features are not used in the scope of this paper, and their absence do not hamper the expressiveness of the proof language.

2.1 Monadic Datatype

We call *proof object* the concrete representation of a possibly incomplete proof tree. The formalism presented here makes use of a coarse abstraction of this representation: we only assume that proof objects provide means to distinguish the set of current goals among all open goals. We take X as the type of the proof objects, and x, y, z as inhabitants of X, i.e., proof object variables. We define the monadic datatype $\mathcal{M}X$ as follows:

$$\begin{aligned} \textbf{datatype}\ \mathcal{M}X = {} & \mathit{success} : X \to \mathcal{M}X \\ \mid {} & \mathit{subgoals} : \mathbb{N} \to X \to \mathcal{M}X \\ \mid {} & \mathit{exception} : \mathbb{S} \to \mathcal{M}X \ , \end{aligned}$$

where $\mathbb{N}$ is the type of natural numbers and $\mathbb{S}$ is the type of symbols. We use the meta-variables $m, m_1, m_2, \ldots$ to range over objects of the type $\mathcal{M}X$. The intended semantics of the datatype constructors is the following:

- *success* indicates that the tactic has discharged (proved) the current goals;
- *subgoals* n indicates that n subgoals have been generated by the tactic. By convention *subgoals* 0 means that the current goal was not modified;
- *exception* s indicates that the tactic has raised the exception s.

Overall, this representation is focused on the three fundamental proof states that are relevant to the user: whether goals were closed, goals were generated, or something went wrong.

Example 2.1 In PVS, the tactic `(split)` corresponds to a function that returns *subgoals* 2 when applied to a goal that is a simple conjunction, *success* in the special case where splitting this conjunction yields two tautologies that are automatically discharged, and *subgoals* 0 if the goal is not a conjunction.

In the following, we represent PVS proof commands as functions of type $X \to \mathcal{M}X$. More precisely, let $|\texttt{.}|_{.}$ be a *semantic evaluator* of PVS proof commands into objects of type $\mathcal{M}X$. This evaluator is defined for each particular proof command. We write $|\texttt{t}|_x$ the evaluation of the proof command `t` on the proof object x. Therefore, $|\texttt{t}|_{\_}$ has the type $X \to \mathcal{M}X$, and can be considered as a function $t = \lambda x.m$. Henceforth, we distinguish the proof commands, such as `t`, from their mathematical representation, i.e., t, by enforcing their typesetting in, respectively, typewriter and math fonts.

2.2 Monadic Operators

Figure 1 introduces the operators for our monadic datatype:

- the function *unit*, of type $X \to \mathcal{M}X$, maps a proof object into an element of our datatype,
- 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 object resulting from the application of another tactic.

$$\boxed{\begin{aligned} &\mathit{unit}\ x = \mathit{subgoals}\ 0\ x\ \ , \\ &m \star t = \begin{cases} \mathit{subgoals}\ n\ y & \text{if } m = \mathit{subgoals}\ n\ x \text{ and } (t\ x) = \mathit{subgoals}\ 0\ y \\ (t\ x) & \text{if } m = \mathit{subgoals}\ n\ x \text{ and } (t\ x) \neq \mathit{subgoals}\ 0\ y \\ m & \text{otherwise}\ \ , \end{cases} \end{aligned}}$$

Fig. 1. Monadic operators

These operators satisfy the monad laws.

Proposition 2.2 *The operators satisfy the left and right unit properties:*

$$\begin{aligned} (\mathit{unit}\ x) \star t &= t\ x \\ m \star \lambda x.\mathit{unit}\ x &= m\ \ , \end{aligned}$$

and the operator $\star$ is associative:

$$m_1 \star (\lambda x.m_2 \star \lambda y.m_3) = (m_1 \star \lambda x.m_2) \star \lambda y.m_3\ \ .$$

Proof. The left and right unit properties are trivial, one can check them simply by unfolding the definition of $\star$. Associativity in the case of *success* or *exception* of one of the tactics is direct. In the case of *subgoals*, it is inferred from the associativity of the boolean addition between zero and non-zero subgoals. □

In PVS, *unit* corresponds to the semantics of the tactical `skip`:

$$|\texttt{(skip)}|_x = \mathit{unit}\ x\ \ .$$

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. Hence, it corresponds to the semantics of a binary sequence. Let t_1 and t_2 be the semantic evaluations of tactics $\texttt{t}_1$ and $\texttt{t}_2$,

$$|\texttt{(then2 t}_1\ \texttt{t}_2\texttt{)}|_x = (t_1\ x) \star t_2\ \ .$$

We note that this does not correspond to the semantics of PVS's proof command `then`, which is based on `try`.

Example 2.3 The semantics of PVS's `try` requires two types of exceptions that handle the "failure" and "backtracking" mechanisms. Let t_1, t_2, t_3 be the semantic evaluations of $\texttt{t}_1, \texttt{t}_2, \texttt{t}_3$, respectively,

$$|\texttt{(try t}_1\ \texttt{t}_2\ \texttt{t}_3\texttt{)}|_x = \begin{cases} (t_3\ x) & \\ \quad \text{if} \quad (t_1\ x) = \mathit{exception\ backtrack} & \\ \quad \text{or} \quad (t_1\ x) = \mathit{subgoals}\ 0\ y & \\ (t_1\ x) & \\ \quad \text{if} \quad (t_1\ x) = \mathit{exception\ failure} & \\ \quad \text{or} \quad (t_1\ x) = \mathit{success}\ y & \\ \mathit{exception\ backtrack} & \\ \quad \text{if} \quad (t_1\ x) = \mathit{subgoals}\ n\ y, \quad n > 0, & \\ \quad \text{and} \quad (t_2\ y) = \mathit{exception\ failure} & \\ \quad\quad \text{or} \quad (t_2\ y) = \mathit{exception\ backtrack} & \\ \mathit{subgoals}\ n\ z & \\ \quad \text{if} \quad (t_1\ x) = \mathit{subgoals}\ n\ y, \quad n > 0, & \\ \quad \text{and} \quad (t_2\ y) = \mathit{subgoals}\ 0\ z & \\ \mathit{subgoals}\ n'\ z & \\ \quad \text{if} \quad (t_1\ x) = \mathit{subgoals}\ n\ y, \quad n > 0, & \\ \quad \text{and} \quad (t_2\ y) = \mathit{subgoals}\ n'\ z & \\ \mathit{success}\ z & \\ \quad \text{if} \quad (t_1\ x) = \mathit{subgoals}\ n\ y, \quad n > 0, & \\ \quad \text{and} \quad (t_2\ y) = \mathit{success}\ z\ \ . & \end{cases}$$

This formalization of the semantics of `try` is clearly more space-consuming than the one presented in the introduction. This is partly due to the verbosity of the *exception* construct. But it also reflects the complexity of this specific tactical, which provides several features: sequencing, progress testing, backtracking, and error catching. This complexity is inherited by the tacticals that are derived from `try`, e.g., `then` and `else`:

$$|(\texttt{then t}_1\ \texttt{t}_2)|_x = |(\texttt{try t}_1\ \texttt{t}_2\ \texttt{t}_2)|_x$$
$$|(\texttt{else t}_1\ \texttt{t}_2)|_x = |(\texttt{try t}_1\ \texttt{(skip) t}_2)|_x\ .$$

As illustrated, the monadic datatype allows for a formal description of PVS's tacticals such as `skip`, `then`, `try`, etc. The next section will propose a simpler set of tacticals that can be derived from our formalism.

3 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 `#throw` for throwing an exception, one tactical `#catch` for catching an exception and implicitly backtracking, and one tactical `#ifsubgoals` for testing progress. PVS's tacticals defined via `try` and `fail` cannot be combined with PVS# tacticals. For this reason, PVS# also provides replacement for `try`-based PVS tacticals such as `then` and `else`.

All the tacticals in PVS# are designed to have simple, if not atomic, interpretations in our framework. In the rest of this section, we will describe these new tacticals, coupling their traditional informal description with their formal semantics.

Henceforth, we will assume that t_i is the semantic evaluation of a proof command $\texttt{t}_i$ for any index i.

3.1 Exception Handling and Progress Testing

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

$$|(\texttt{\#throw}\ \mathit{tag})|_x = \mathit{exception\ tag}\ .$$

(`#catch` $\texttt{t}_1$ `&optional` *tag* $\texttt{t}_2$) This tactical behaves as $\texttt{t}_1$ if $\texttt{t}_1$ does not raise an exception. Otherwise, if the result is an exception named *tag* then it evaluates $\texttt{t}_2$. 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"`.
Semantics:

$$|\texttt{(\#catch t}_1\ \textit{tag}\ \texttt{t}_2\texttt{)}|_x = \begin{cases} (t_2\ x) & \text{if } (t_1\ x) = \textit{exception tag} \\ (t_1\ x) & \text{otherwise} \ . \end{cases}$$

(#ifsubgoal t t$_1$ t$_2$) This tactical calls either t$_1$ or t$_2$, depending on the progress of `t`. If `t` generates subgoals, then it applies t$_1$ to all the subgoals. Otherwise, it applies t$_2$.
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.
Semantics:

$$|\texttt{(\#ifsubgoals t}_1\ \texttt{t}_2\ \texttt{t}_3\texttt{)}|_x = \begin{cases} (t_2\ y) & \text{if } (t_1\ x) = \textit{subgoals}\ n\ y \text{ and } n > 0 \\ (t_3\ y) & \text{if } (t_1\ x) = \textit{subgoals}\ 0\ y \\ (t_1\ x) & \text{otherwise} \ . \end{cases}$$

3.2 Identity, Sequencing and Repeating

(#skip) As in PVS, this tactical has no effect. Actually, `(#skip)` is strictly equal to `(skip)`, this alias being provided for the sake of uniformity.
Semantics:

$$|\texttt{(\#skip)}|_x = \textit{unit}\ x = \textit{subgoals}\ 0\ x \ .$$

(#then t$_1$ ... t$_n$) This tactical first applies `t1` to the current goal, and then (#then t$_2$ ... t$_n$) to all of the generated subgoals, if any, or to the original goal if `t1` had no effect.
Semantics:

$$|\texttt{(\#then t}_1\ \texttt{t}_2 \ldots \texttt{t}_n\texttt{)}|_x = (t_1\ x) \star t_2 \star \ldots \star t_n \ .$$

`(#repeat t)` Iteratively apply `t` to the current goal until it fails, proves the goal or does nothing. This tactical may not terminate.
Semantics:

$$|\texttt{(\#repeat t)}|_x = \begin{cases} |\texttt{(\#repeat t)}|_y & \text{if } (t\ x) = \textit{subgoals}\ n\ y \text{ and } n > 0 \\ (t\ x) & \text{otherwise} \ , \end{cases}$$

3.3 Other Tacticals

`(#if` *expr* $\texttt{t}_1$ $\texttt{t}_2$`)` As in PVS, the Lisp expression *expr* is evaluated against the current goal. If $\texttt{t}_1$ and $\texttt{t}_2$ were elements of the PVS proof language, this construct is equivalent to PVS's tactical `if`.
Usage: The proof script

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

applies `(ground)` if the current goal is the first subgoal of its parent, else it applies `(prop)`.
Semantics:

$$|\texttt{(\#if}\ \textit{expr}\ \texttt{t}_1\ \texttt{t}_2\texttt{)}|_x = \begin{cases} (t_1\ x) & \text{if } \textit{expr} =_{\textit{Lisp}} \textit{\textbf{nil}} \\ (t_2\ x) & \text{otherwise} \ , \end{cases}$$

`(#when` *expr* $\texttt{t}_1 \ldots \texttt{t}_n$`)` This tactical evaluates *expr*, if it results in `nil` then nothing is done and this tactical behaves as `skip`. Otherwise, it applies $\texttt{t}_1 \ldots \texttt{t}_n$ in sequence using `#then`.
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.
Semantics:

$$|\texttt{(\#when}\ \textit{expr}\ \texttt{t}_1 \ldots \texttt{t}_n\texttt{)}|_x = |\texttt{(\#if}\ \textit{expr}\ \texttt{(\#then t}_1 \ldots \texttt{t}_n\texttt{) (\#skip))}|_x$$

`(#first` $\texttt{t}_1 \ldots \texttt{t}_n$`)` This tactical applies the first tactic in $\texttt{t}_1 \ldots \texttt{t}_n$ 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.
Semantics:

- If $n = 1$

$$|\texttt{(\#first t)}|_x = \begin{cases} \textit{subgoals}\ 0\ x & \text{if } (t\ x) = \textit{exception } s \\ (t\ x) & \text{otherwise} \ . \end{cases}$$

- If $n > 1$,

$$|\texttt{(\#first t}_1\ \texttt{t}_2 \ldots \texttt{t}_n\texttt{)}|_x = \begin{cases} |\texttt{(\#first t}_2 \ldots \texttt{t}_n\texttt{)}|_x & \text{if } (t_1\ x) = \textit{exception } s, \\ (t_1\ x) & \text{otherwise} \ , \end{cases}$$

`(#solve t`$_1$ `...` `t`$_n$`)` This tactical applies the first tactic in `t`$_1$ `...` `t`$_n$ 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.
Semantics:
- If $n = 1$

$$|\texttt{(\#solve t)}|_x = \begin{cases} (t\ x) & \text{if } (t\ x) = \textit{success } x \\ \textit{subgoals}\ 0\ x & \text{otherwise} \ . \end{cases}$$

- If $n > 1$,

$$|\texttt{(\#solve t}_1\ \texttt{t}_2 \ldots \texttt{t}_n\texttt{)}|_x = \begin{cases} (t_1\ x) & \text{if } (t_1\ x) = \textit{success } x \\ |\texttt{(\#solve t}_2 \ldots \texttt{t}_n\texttt{)}|_x & \text{otherwise} \ , \end{cases}$$

4 Implementation

This section presents the internal tacticals that were used to deal with the implementation in PVS of the monadic datatype. They are separated into two different categories, the constructors and destructors of $\mathcal{M}X$.

4.1 Constructors

`(piks)` This is the constructor for *subgoals* $1\ x$. It generates one subgoal, which is identical to the original goal.
Semantics:

$$|\texttt{(piks)}|_x = \textit{subgoals}\ 1\ y$$

where y is x with the current goal being inferred from itself.

`(backtrack)` This generates the *exception backtrack* proof state, identical to the one created by `try`.
Semantics:

$$|\texttt{(backtrack)}|_x = \textit{exception backtrack} \ .$$

4.2 Destructors

`flag` This is a simple list structure, with five boolean fields: `success`, `subgoal` which is short for *subgoals* $n \geq 1$, `skip` which is short for *subgoals* 0, `backtrack` which is short for *exception backtrack*, and `fail` which is short for *exception failure*. It is used to record the outcome of tactics that are tested by the following tacticals.

`(figure t otc)` This helper is the core of the destructor. It applies `t` to the current goal, analyzes its outcome, and fills in `otc` (an instance of `flag`) with it. However, if `t` proves the goal, then `otc` is not updated (because no computation can be done after the goal was proven). This tactical returns either *success* (if `t` was a *success*), *exception backtrack* (if `t` was a *subgoals* 0, an *exception backtrack* or an *exception failure*), or *subgoals* $n \geq 1$ (if `t` returned *subgoals* $n \geq 1$).
Semantics:

$$|\texttt{(figure t } otc\texttt{)}|_x = \begin{cases} \textit{exception backtrack} \\ \qquad \text{if} \quad (t\ x) = \textit{subgoals}\ 0\ y \\ \qquad \text{or} \quad (t\ x) = \textit{exception backtrack} \\ \qquad \text{or} \quad (t\ x) = \textit{exception failure} \\ (t\ x) \text{ otherwise} \ . \end{cases}$$

`(inspect t otc)` This helper applies `(figure t otc)` to a dummy goal, then discards it, fills in completely the outcome in `otc` and returns to the original goal.
Semantics:

$$|\texttt{(inspect t } otc\texttt{)}|_x = \textit{subgoals}\ 0\ x \ .$$

`(info t)` This is the easiest tactical written with `inspect`. It simulates the application of `t`, fills in an instance of `flag`, and prints it out.
Semantics:

$$|\texttt{(info t)}|_x = \textit{subgoals}\ 0\ x \ .$$

`(test-case t t`$_1$ `t`$_2$ `t`$_3$ `t`$_4$ `t`$_5$`)` This is the destructor of the datatype. It analyzes `t` using `inspect`, and according to the result applies one of the `ti`.
Semantics:

$$|\texttt{(test-case t t}_1\texttt{ ... t}_5\texttt{)}|_x = \begin{cases} (t_1\ x) & \text{if } (t\ x) = \mathit{success}\ y \\ (t_2\ x) & \text{if } (t\ x) = \mathit{subgoals}\ n\ y \text{ and } n > 0 \\ (t_3\ x) & \text{if } (t\ x) = \mathit{subgoals}\ 0\ y \\ (t_4\ x) & \text{if } (t\ x) = \mathit{exception\ backtrack} \\ (t_5\ x) & \text{if } (t\ x) = \mathit{exception\ failure}\ . \end{cases}$$

`(testsuccess t t1 t2)`

`(testsubgoalsN t t1 t2)`

`(testsubgoals0 t t1 t2)`

`(testexceptionbacktrack t t1 t2)`

`(testexceptionfail t t1 t2)` These tacticals are instances of the `test-case` tactical, they apply their first argument. If its outcome is the one expected they apply their second argument. Else they apply their third argument.
Semantics:

$$|\texttt{(testsuccess t t}_1\texttt{ t}_2\texttt{)}|_x = \begin{cases} (t_1\ x) & \text{if } (t\ x) = \mathit{success}\ y \\ (t_2\ x) & \text{otherwise}\ . \end{cases}$$

The semantics of the other testing strategies are analogous.

5 Conclusion

We have defined a representation of proof state information via a monadic datatype, which is orthogonal to the physical representation of proof objects. This has allowed us to give a synthetic representation of the PVS proof state.

The formal description of PVS tacticals in our formalism has revealed unnecessary complexities in the PVS proof language. Therefore, we have proposed a new set of PVS tacticals implemented on top of the existing proof language, called PVS#, that arguably have a simpler semantics with respect to error handling and sequencing. 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 monads in proof languages, and its application to other theorem provers. In particular, another implementation of the monadic datatype was already carried out in the Fellowship proof assistant [6]. In the long term, we believe that the concept of monads 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., "Un calcul de substitutions pour la représentation de preuves partielles en théorie de types," Thèse de doctorat, Université Paris 7 (1997), English version available as INRIA research report RR-3309.

[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: J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS **925**, Springer Verlag, 1995 .