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:
  - τ.seq_tree: the tree of sequents.
  - τ.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 τ.active represents a sequent $\Gamma \vdash \Delta$, and we will write: $\tau.\ \Gamma \vdash \Delta$ to refer to such a proof context.
  - τ.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:
  - τ.addLeaves($\Gamma_1 \vdash \Delta_1, \ldots, \Gamma_n \vdash \Delta_n$): 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 $\Gamma_i \vdash \Delta_i$, $i \in \{1, \ldots, n\}$, will be leaves, and the former active leaf of the old tree will become their common parent node.
  - τ.lowerPointer(i): moves the active pointer down (towards the root) in the tree, $i \geq 0$ being the depth of the move.
  - τ.raisePointerToLeaf(): moves the pointer up to the first (i.e., innermost leftmost) unproven leaf of the tree.
  - τ.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 $\varnothing$.
  - τ.setProgress(b): sets the corresponding flag to b.
  - τ.hasProgressed(): returns the value of the progress flag.
  - τ.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., τ.setLeafProven() $= \top$).
  - τ.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.

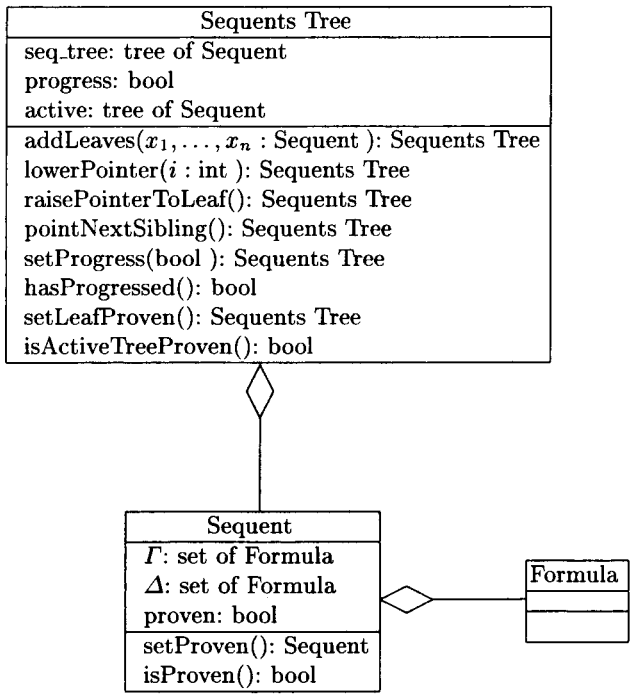

Fig. 1. Proof context objects

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

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

is transformed into the sequent tree

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

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:

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

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	$::=$	$expr.$	all expressions must end with "." .

And

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

4.2 Semantics

The values of the semantics are defined as:

v	$::=$	p	
	$\mid$	Fun $x \rightarrow e$	
	$\mid$	Rec $x_1\ x_2 \rightarrow e$.	

[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.

$$(\text{Fun } x \multimap e)(v) \;/\; \tau \xrightarrow{\epsilon} e[x \leftarrow\!\shortmid v] \;/\; \tau \;.$$
$$(\text{Rec f } x \multimap e)(v) \;/\; \tau \xrightarrow{\epsilon} e[x \leftarrow\!\shortmid v][\text{f} \leftarrow\!\shortmid (\text{Rec f } x \multimap e)] \;/\; \tau \;.$$

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

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

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 $i \in \{1, \ldots, n\}$, $\sigma_{p_i \leftarrow\!\shortmid t}$ 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:

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

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:

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

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

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

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:

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

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, $\overline{;_\tau}$, to allow potential backtracking.

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

and

$$\begin{aligned} \overline{;_\tau} v_1, e_2\ldots,e_n \;/\; \tau' &\xrightarrow{\epsilon} \overline{;_\tau} e_2,\ldots,e_n \;/\; (v_1\%\tau').\,\text{pointNextSibling}() \\ &\qquad \text{if } \forall n \geq 0\ (v_1\%\tau') \neq \bot_n\ , \\ \overline{;_\tau} v_1, e_2\ldots,e_n \;/\; \tau' &\xrightarrow{\epsilon} (\text{Fail } 0) \;/\; \tau \quad \text{if } \exists n \geq 0\ (v_1\%\tau') = \bot_n\ , \\ \overline{;_\tau} v_n \;/\; \tau' &\xrightarrow{\epsilon} (\text{Fail } 0) \;/\; \tau \quad \text{if } \tau' = \varnothing \\ &\qquad \text{or } (v_n\%\tau').\,\text{pointNextSibling}() \neq \varnothing\ , \\ \overline{;_\tau} v_n \;/\; \tau' &\xrightarrow{\epsilon} \text{Idtac} \;/\; (v_n\%\tau').\,\text{lowerPointer}(1) \quad \text{if } \tau' \neq \varnothing \\ &\qquad \text{and } (v_n\%\tau').\,\text{pointNextSibling}() = \varnothing\ . \end{aligned}$$

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

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

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

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

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 $\overline{\text{Do}_e}$.

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

with

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

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

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

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

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

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

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

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

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

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).

$$\begin{aligned}
\mathcal{M}.\text{newTactical}(x, e) &\longrightarrow \mathcal{M}\{x \mapsto e\} \ , \\
& \qquad \text{if } x \notin \text{Dom}(\mathcal{M}). \\
\mathcal{M}.\text{getTactical}(x) &\longrightarrow \mathcal{M}(x) \ .
\end{aligned}$$

The declaration of new commands simply writes:

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

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:

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

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 $\perp_0$ as a return value for any undefined point.

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

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

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

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

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:

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

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 (¶):

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

And

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

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:

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

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.

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

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 $\overline{\text{try}_\tau}$.
Remark that the sequencial tactical `then` is simply defined as `(then` v_1 v_2`)` = `(try` v_1 v_2 v_2`)`.

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

with

$$\begin{aligned}
&(\overline{\text{try}_\tau}\ v)\ /\ \tau' \xrightarrow{\epsilon} v\ /\ \tau' && \text{if } \forall n \geq 0\ (v\%\tau') \neq \bot_n \\
&(\overline{\text{try}_\tau}\ v)\ /\ \tau' \xrightarrow{\epsilon} \texttt{(skip)}\ /\ \tau && \text{if } \exists n \geq 0\ (v\%\tau') = \bot_n\ .
\end{aligned}$$

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.

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

with

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

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

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

with

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

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.

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

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

$$\begin{aligned}
&\overline{\text{spread}_\tau^{l}} v_1, e_2 \ldots, e_n\ /\ \tau' \xrightarrow{\epsilon} \\
&\overline{\text{spread}_\tau^{l}} e_2, \ldots, e_n\ /\ (v_1\%\tau').\text{pointNextSibling()} \\
&&& \text{if } \forall n \geq 0\ (v_1\%\tau') \neq \bot_n\ ,
\end{aligned}$$

and

$$\overline{\text{spread}^{l}_{\tau}} v_1, e_2 \ldots, e_n \;/\; \tau' \xrightarrow{\epsilon} \texttt{(fail)} \;/\; \tau \quad \text{if } \exists n \geq 0 \; (v_1 \% \tau') = \bot_n \;,$$

and

$$\frac{\overline{\text{spread}^{v_0, e_1, \ldots, e_n}_{\tau}} v_n \;/\; \tau' \xrightarrow{\epsilon}}{\overline{\text{spread}^{v_0, e_1, \ldots, e_{n-1}}_{\tau}} v_0, e_1, \ldots, e_{n-1} \;/\; \tau} \quad \text{if } \tau' = \varnothing \;,$$

and

$$\overline{\text{spread}^{l}_{\tau}} v_n \;/\; \tau' \xrightarrow{\epsilon} \texttt{(skip)} \;/\; (v_n \% \tau').\text{lowerPointer}(1) \quad \text{if } \tau' \neq \varnothing \text{ and } (v_n \% \tau').\text{pointNextSibling}() = \varnothing \;,$$

and

$$\frac{\overline{\text{spread}^{v_0, e_1, \ldots, e_n}_{\tau}} v_n \;/\; \tau' \xrightarrow{\epsilon}}{\overline{\text{spread}^{v_0, e_1, \ldots, e_n, (skip)}_{\tau}} v_0, e_1, \ldots, e_n, \texttt{(skip)} \;/\; \tau} \quad \text{if } (v_n \% \tau').\text{pointNextSibling}() \neq \varnothing \;,$$

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

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

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

$$\frac{\overline{\text{branch}^{v_0, e_1, \ldots, e_n}_{\tau}} v_n \;/\; \tau' \xrightarrow{\epsilon}}{\overline{\text{branch}^{v_0, e_1, \ldots, e_n, e_n}_{\tau}} v_0, e_1, \ldots, e_n, e_n \;/\; \tau} \quad \text{if } (v_n \% \tau').\text{pointNextSibling}() \neq \varnothing \;,$$

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.

$$\frac{\texttt{(try-branch } v_0 \; (e_1 \ldots e_n) \; e') \;/\; \tau \xrightarrow{\epsilon}}{\overline{\text{(try-branch}^{v_0, e_1, \ldots, e_n}_{\tau}} \; e_1 \ldots e_n) \;/\; (v_0 \% \tau)} \quad \text{if } (v_0 \% \tau).\text{hasProgressed}() \text{ and } \forall n \geq 0 \; (v_0 \% \tau) \neq \bot_n \;.$$

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

$$\texttt{(try-branch } v_0 \; (e_1 \ldots e_n) \; e') \;/\; \tau \xrightarrow{\epsilon} e' \;/\; \tau \quad \text{if } \neg (v_0 \% \tau).\text{hasProgressed}(),$$

with

$$\frac{\overline{\text{(try-branch}^{l}_{\tau}} \; v_1 e_2 \ldots e_n) \;/\; \tau' \xrightarrow{\epsilon}}{\overline{\text{(try-branch}^{l}_{\tau}} \; e_2 \ldots e_n) \;/\; (v_1 \% \tau').\text{pointNextSibling}()} \quad \text{if } \forall n \geq 0 \; (v_1 \% \tau') \neq \bot_n \;,$$

and

$$\overline{(\text{try-branch}^{l}_{\tau}\ v_1 e_2 \ldots e_n)}\ /\ \tau' \xrightarrow{\epsilon} (\texttt{skip})\ /\ \tau \quad \text{if } \exists n \geq 0\ (v_1 \% \tau') = \bot_n\ ,$$

and

$$\frac{\overline{\text{try-branch}^{v_0, e_1, \ldots, e_n}_{\tau}}\ v_n\ /\ \tau' \xrightarrow{\epsilon}}{\overline{\text{try-branch}^{v_0, e_1, \ldots, e_{n-1}}_{\tau}}\ e_1, \ldots, e_{n-1}\ /\ (v_0 \% \tau)} \quad \begin{array}{l}\text{if } \tau' = \varnothing \\ \text{or } (v_n \% \tau').\text{pointNextSibling}() \neq \varnothing\ ,\end{array}$$

and

$$\overline{\text{try-branch}^{l}_{\tau}}\ v_n\ /\ \tau' \xrightarrow{\epsilon} (\texttt{skip})\ /\ (v_n \% \tau').\text{lowerPointer}(1) \quad \begin{array}{l}\text{if } \tau' \neq \varnothing \\ \text{and } (v_n \% \tau').\text{pointNextSibling}() = \varnothing\ ,\end{array}$$

and

$$\frac{\overline{\text{try-branch}^{v_0, e_1, \ldots, e_n}_{\tau}}\ v_n\ /\ \tau' \xrightarrow{\epsilon}}{\overline{\text{try-branch}^{v_0, e_1, \ldots, e_n, e_n}_{\tau}}\ e_1, \ldots, e_n, e_n\ /\ (v_0 \% \tau)} \quad \text{if } (v_n \% \tau').\text{pointNextSibling}() \neq \varnothing\ ,$$

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 $\mathcal{M}(x) = (Box\ e)$, 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:

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

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

5.4 Context

$$
\begin{aligned}
E ::= {} & [\,] \\
| {} & E\ \P \\
| {} & (\texttt{try}\ E\ e_2\ e_3) \\
| {} & \overline{\text{try}_\tau} E \\
| {} & (\texttt{spread}\ E\ (e'_1 \ldots e'_n)) \\
| {} & \overline{\text{spread}_\tau^{v_0, e_1, \ldots, e_n}} E\ e_i \ldots e_n \\
| {} & (\texttt{branch}\ E\ (e'_1 \ldots e'_n)) \\
| {} & \overline{\text{branch}_\tau^{v_0, e_1, \ldots, e_n}} E\ e_i \ldots e_n \\
| {} & (\texttt{try-branch}\ E\ (e'_1 \ldots e'_n)\ e_2) \\
| {} & \overline{\text{try-branch}_\tau^{v_0, e_1, \ldots, e_n}} E\ e_i \ldots e_n \\
| {} & (\texttt{Glass}\ E) \\
| {} & (\texttt{Black}\ E)\ .
\end{aligned}
$$

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.

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

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

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

$$\begin{aligned}\texttt{(propax)}\%\tau.\ \ \Gamma, A \vdash B\ &= \tau.\,\text{leafProven}().\,\text{setProgess(true)} \\ &\quad \text{if } A \text{ and } B \text{ are syntaxically} \\ &\quad \text{equal.}\end{aligned}$$

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

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

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

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

$$\begin{aligned}&\texttt{(skolem * (``a''))}\%\tau.\ \ \Gamma \vdash (\forall x : A)\ = \\ &\tau.\,\text{addLeaves}\ (\Gamma \vdash A[x \leftarrow\!\shortmid a]).\,\text{setProgess(true)}\ .\end{aligned}$$

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.