

École Normale Supérieure de Cachan
DEA Sémantique, Preuve et Langages
Year 2002-2003

Towards a Common Tactical Language: The Case of Coq and PVS.

An internship by
Florent Kirchner

under guidance of
Gilles Dowek and César Muñoz

in the team
LogiCal (INRIA-FUTURS)

and in cooperation with
Formal Methods Group (NIA – NASA)

“... Transformer en conscience
une expérience aussi large que possible...”.

André Malraux. *L'espoir*

“... To transform into sentience
an experience as large as possible...”.

André Malraux. *Man's Hope*

Abstract

Coq's tacticals and PVS's strategies, though parts of completely separate languages, were at first designed based on more or less the same requirements: to powerfully factorize the functionalities provided by the provers' tacticals. Besides both the Coq and the PVS procedural provers have different characteristics, which might determine the nature of their application. Nevertheless the importance of their interoperability should not be neglected –at least for double-checking purposes. The following work exposes how this interoperability can be achieved at the level of the provers' tacticals; it proposes a detailed small-step semantics of the languages as well as an implementation of a common basis for an unified language of tacticals.

Contents

1	Introduction	1
1.1	Procedural Proof Languages	1
1.2	Coq and PVS	1
1.2.1	Coq	1
1.2.2	PVS	1
1.2.3	De Combinationis...	2
1.3	The LogiCal Project	2
1.4	The Formal Methods Group at NIA	2
2	Logics	3
3	Conventions and Structures	3
3.1	The Proof Context	3
3.2	The Proof Script	5
4	The Semantic Framework	6
5	Coq's Tacticals	7
5.1	Syntax	7
5.2	Semantics	7
5.3	Toplevel Definitions	11
5.4	Context	12
5.5	Tactics	12
6	PVS's Strategies	13
6.1	Syntax	13
6.2	Semantics	13
6.3	User-defined Strategies	17
6.4	Context	17
6.5	Tactics	18
7	Implementation	18
7.1	Coq	19
7.2	PVS	20
8	Conclusion and Future Work	24
9	Acknowledgment	24

1 Introduction

1.1 Procedural Proof Languages

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; they tend to simplify the proposition that is being proven, and, wisely chosen, lead to a proposition trivial enough to be recognized as true by the prover. The instructions submitted to the proof engine can be regrouped into a list called *proof script*.

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 size makes any proof reading or maintenance operation tedious if not impossible. Most procedural provers introduce combinators in their proof language in order to compose elementary proof tactics: generally they will be referred to as *tacticals*. For instance, in Coq, the infix binary tactical “;” combines two tactics by applying them in a sequence, and the unary tactical “Repeat” applies iteratively its argument.

The following section will present Coq and PVS which are examples of procedural provers and will be at the center of our study. Other provers such as Isabelle and NuPr1 also implement tacticals, they have not been included in this work but a similar reasoning could probably apply.

1.2 Coq and PVS

1.2.1 Coq

Coq [1] is derived from the first procedural theorem prover called LCF [7] which was developed at the University of Edinburgh at the end of the 70’s. The Coq project was initiated by Thierry Coquand and Gérard Huet in 1984, the first distribution (v. 4.1.0) was made public in 1989, and in July 2003 the current release is numbered 7.3.1. This prover, written in Caml, implements a higher-order type theory through a typed lambda-calculus named *Calculus of Inductive Constructions*: such a logical framework, fundamentally intuitionistic, allows the representation of the proofs as lambda-terms (using the Curry-Howard isomorphism). As a consequence of the choice of this intuitionism, the proofs can be harder than with classical logic, resulting in a less automated prover than PVS or HOL.

As we have seen previously, Coq as a procedural prover allows the user to control the execution of the proof through the use of tactics and tacticals. It is also possible to enrich the proof language with new tactics or tacticals, either by writing Caml code, that is by modifying the prover’s source code, which is a powerful yet complicated procedure. Or one can directly use the tacticals to define new commands, in which case no specific knowledge of the details of the implementation of Coq is needed; however with this option the possibilities are more limited and might be insufficient to code a highly potent functionality. It is interesting to note that those two options are complementary and not competing, and thus to highlight the essential distinction between the prover’s proof language and the language used in its implementation.

1.2.2 PVS

The *Prototype Verification System*, created in 1992 by Sam Owre, Natarajan Shankar and John Rushby [15], is partially inspired from LCF, and is written in Common Lisp. Unlike Coq though, its source code is not open. PVS implements a variation of the theory of simple types at higher order, and adds features such as structural induction, dependant types and explicit subtyping. This latter addition makes the type verification undecidable, thus forcing the prover to generate type correctness conditions (TCC) which must be discharged by hand. The use of classical logic allows for quite a powerful automation, widely used by PVS; however proofs can not be expressed as lambda-terms as in Coq: the proof representation is purely an intern one and cannot be directly exported. Interestingly enough, PVS has a quite unique way to display a proof: since it cannot directly export it, it uses a tactic tree which is isomorphic to the proof tree. Each tactic in the tree represents the logical rule applied in the corresponding position in the proof tree.

The aforementioned concept of tactics and tacticals is present in PVS, although the distinction between them is shallower. The word *strategies* designates tacticals, but also any user-defined command whether it

is a tactic or a tactical (i.e., a combinator of tactics). In the rest of this study we will consider that the words tacticals and strategies are synonyms, thereby neglecting the fact that a strategy might as well be a user-defined tactic¹.

1.2.3 De Combinationis...

The previous short descriptions highlight the similarities and differences between the provers. The choices made in the implementation, hence the inherent logics, qualify them for very different applications, but then again not necessarily disconnected. For example one could want to verify a specification for a flight system, in which some parts would be quite conceptual and that the user would like to prove quickly using a very automated tool; whereas other parts would benefit from the extraction functionality provided by the use of intuitionistic logic to automatically generate certified code.

It would be interesting to be able to establish some interoperability between provers with such complementary features. This idea can be developed in a few ways, the pursuit of a common basis of tacticals –and more generally of a common proof language– being one. Another could aim at the representation of proofs and try to present an unified structure for this usage. Actually, the only exchanges between PVS and Coq consisted in the sharing of conceptual ideas, which in the end resulted in quite different implementations [9], [5]. Concerning other provers, Caldwell and Cowles developed a representation of Nuprl proof objects in ACL2, enabling Nuprl proofs to be checked in ACL2 [2]. In all cases the differences in the implementation of the provers may steer, restrict or impede any attempt to unify any part of their structure.

Thus, after discussing the possibility of a common language of tacticals, we will propose a semantics of the existing sets of tacticals. Based on these two results, we will expose an implementation which aims at levelling the functionalities of these languages, setting the very basis for an unified representation of tacticals in Coq and in PVS.

1.3 The LogiCal Project

The LogiCal team deals with the construction, development and maintenance of proof processing systems. Using a computerized system to process mathematical proofs allows its user to reach a high degree of certainty, that a proof does not contain any error. Such an insurance is particularly valuable when dealing with large proofs, for instance proofs involving polynomials formed with several hundreds of terms. Also, this participates to the quest of a new form of rigour in mathematical writing : the point where nothing is left implicit and where the reader can be replaced by a program.

The main line of research in this work is the development of the system Coq. This system has a large community of industrial and academic users. However, the LogiCal team believes that the development of a system cannot be separated from a more applied research on the uses of this system in particular domains (such as geometry, proofs of imperative or object-oriented programs, proofs of protocols, ...) and a more fundamental research on the formalization of mathematics (on the representation of proofs, on the integration of a programming language in a mathematical formalism, on the notion of bound variable, ...). This work positions itself in this latter field, by trying to understand and formalize certain aspects of the proof languages.

1.4 The Formal Methods Group at NIA

The Formal Methods research program at the National Institute of Aerospace (NIA) aims at developing and applying mathematical techniques and tools for the specification, analysis, and verification of digital systems that are of interest to NASA. Application areas include flight guidance systems, integrated modular avionics, airborne information systems, and other hardware and software systems which are critical to aviation safety. Current research efforts focus on the verification of Air Traffic Management Systems.

NIA's Formal Methods program supports NASA Langley's Formal Methods Group by developing and applying state-of-the-art formal technologies.

¹This simplification clarifies the following presentation and does not affect our semantics' relevancy, since it only deals with tacticals. Nevertheless, the reduction rules for user-defined strategies that are exposed in section 6.3 are compatible with user-defined tactics.

Neither NIA nor NASA Langley has sponsored the development of any general-purpose theorem prover. However, the technology transfer projects have led to significant improvements in the Prototype Verification System (PVS) theorem prover through the development and maintenance of domain-specific mathematical libraries. This work contributes to a better understanding of PVS's proof language, and the implementations presented add to the prover's development.

2 Logics

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:

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

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 $A_1, \dots, A_n \vdash B_1, \dots, B_m$, are commonly named *sequents*. Notice that PVS uses numbers to designate formulas within a proposition, numbers which are heavily used to control the application of the tactics. Similarly, the Curry-Howard isomorphism enables Coq to use the term identifiers to designate their related types (which can be seen as propositions). As a consequence the proof scripts are loaded with these alphanumeric references, making them particularly vulnerable to changes in the formula handling and numbering algorithms, thus decreasing their robustness to system updates and proof maintenance.

The implementation discrepancies exposed in section 1.2 between the two provers have an impact on the way logics are dealt with in these provers. As we have seen, Coq implements an intuitionistic logic whereas PVS uses classical logic. An immediate consequence of this difference is that PVS will discharge any proposition such as $\vdash P \vee \neg P$, whilst Coq will ask for a proof of either P or $\neg P$ to be provided to discharge this goal. Also, the use of intuitionistic logics implies that in Coq's sequents, there is only one consequent formula ($m = 1$): they are of the form $A_1, \dots, A_n \vdash B_1$. Thus a rule like (`flatten`) in PVS, has no direct counterpart in Coq. Moreover this difference in the shape of the sequents may interfere with attempts to provide an unified representation of both proof languages: if those languages tamper with quite different objects, how can interoperability be achieved?

Luckily, unlike tactics, tacticals in both languages are completely independent of the cast of the sequents, as we will expose in section 3.2. This liberty and the fact that a number of simple tactics also obey this rule will allow us to propose a complete semantics for these subsets of the languages, using unified conventions, structures, framework. In this work we focus on the tacticals and not so much on the tactics. Hopefully, a quite large subset of the proof languages can be extracted and expressed in a common way.

3 Conventions and Structures

3.1 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. 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 possibly 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 formulas². 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.
- \perp_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 apply³).
- And \emptyset 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., 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 \emptyset .
 - $\tau.setProgress(b)$: 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.

²The 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.

³The 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.

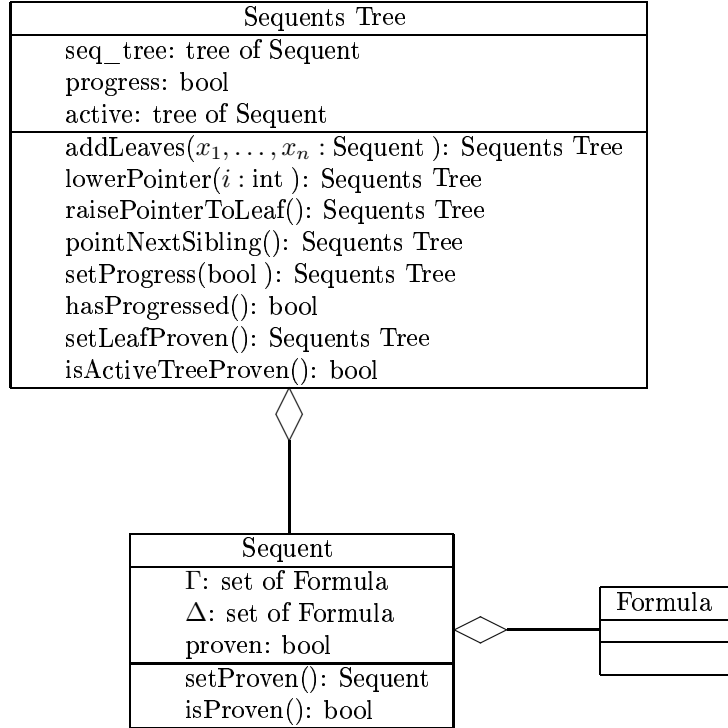


Figure 1: Proof context objects

3.2 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 original distinction between tactics and tacticals within the proof language needs to be clarified, 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

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

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:

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

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.

4 The Semantic Framework

The notion of *small step* or *reduction* semantics was introduced by Plotkin [13] 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 [17], 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 instantiations of this functional definition are of course system specific, and will be exposed in sections 5 and 6.

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:

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

Notice that since we perform a head reduction, the first argument of the tactical needs to be a value of the semantics.

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.

The reduction rules for the tacticals follow. A short example follows each reduction rule.

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:

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

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

$$\begin{aligned} (\text{Fun } x \rightarrow e)(v) / \tau &\xrightarrow{\epsilon} e[x \leftarrow v] / \tau . \\ (\text{Rec } f \ x \rightarrow e)(v) / \tau &\xrightarrow{\epsilon} e[x \leftarrow v][f \leftarrow (\text{Rec } f \ x \rightarrow e)] / \tau . \end{aligned}$$

Usage: $(\text{Fun } x \rightarrow (x; \text{Intro}))(\text{Elim})$ returns the tactical *Elim* ; *Intro* which is then applied to the current goal.

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

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

Usage: *Let* $x = \text{Fresh}$ *In* *Pose* $x := "1 > 0"$ Creates a new fresh identifier and uses it to add a new definition to the proof context.

Term matching This tactical matches a Coq term with a series of patterns, and returns the appropriate expression, properly instantiated.

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 $i \in \{1, \dots, n\}$, $\sigma_{p_i \leftarrow t}$ is the substitution resulting from the matching of t by p_i (undefined if p_i does not match t ; matching by $_$ always succeeds and yields the empty substitution).

The reduction rule then is:

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

Usage: *Match trm With [(0)] -> Fail* | $_$ -> *Idtac* is a proof script that fails if a term *trm* is the integer number zero.

Context matching This tactical matches the current goal with a series of patterns, and returns the appropriate expression, properly instantiated. 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:

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

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

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

Usage: *Match Context With [|- "0 < 1"] -> Apply Rlt_R0_R1* scans the consequent of the current goal and triggers the application of the appropriate lemma if it finds the specific inequality $0 < 1$.

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.

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

Usage: *Intro* ; *Elim* applies the *Intro* tactic to the current goal and then the tactic *Elim* to all the subgoals generated.

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, $\overline{\text{;}}\tau$, to allow potential backtracking. If n does not correspond to the number of generated subgoals, then an error of level 0 is raised.

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

and

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

Usage: *Cut A* ; [*Ring* ; *Assumption*] first applies the *modus ponens* inference rule, and then does AC rewriting on the first generated subgoal and the *Assumption* tactical on the second generated subgoal.

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

$$e_1 \text{ Orelse } e_2 / \tau \xrightarrow{\epsilon} e_1 \overline{\text{Orelse}}_{\tau} e_2 / \tau$$

with

$$\begin{aligned} v_1 \overline{\text{Orelse}}_{\tau} e_2 / \tau' &\xrightarrow{\epsilon} e_2 / \tau' && \text{if } (v_1 \% \tau') = \perp_n \\ &&& \text{or } \neg(v_1 \% \tau'). \text{hasProgressed()} , \\ v_1 \overline{\text{Orelse}}_{\tau} e_2 / \tau' &\xrightarrow{\epsilon} v_1 / \tau' && \text{if } (v_1 \% \tau') \neq \perp_n \\ &&& \text{and } (v_1 \% \tau'). \text{hasProgressed()} . \end{aligned}$$

Usage: *Tauto Orelse Ring Orelse Field* solves the current goal whenever it is a trivial proposition, a polynomial equalities or a fractional equalities.

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

$$\text{Progress } e / \tau \xrightarrow{\epsilon} \overline{\text{Progress}}_{\tau} e / \tau$$

with

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

Usage: the proof script *Progress Apply plus_sym* tries to apply to the current goal the plus symmetry rule, and fails if it does not modify the current goal.

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 $\overline{\text{Do}}_e$, to store the original argument and allow its re-evaluation at each iteration.

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

with

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

Usage: *Do 5 Rewrite plus_assoc_l* rewrites five times every subterm of the current goal with the plus left associativity rule.

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

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

Usage: *Repeat Rewrite plus_assoc_r* rewrites every subterm of the current goal with the plus right associativity rule, until no more change occurs.

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

$$\text{Try } e / \tau \xrightarrow{\epsilon} \overline{\text{Try}}_{\tau} e / \tau$$

with

$$\begin{aligned} \overline{\text{Try}}_{\tau} v / \tau' &\xrightarrow{\epsilon} \text{Idtac} / \tau && \text{if } (v\% \tau') = \perp_0 \\ \overline{\text{Try}}_{\tau} v / \tau' &\xrightarrow{\epsilon} [\text{Fail } (n-1)] / \tau && \text{if } \exists n > 0 (v\% \tau') = \perp_n \\ \overline{\text{Try}}_{\tau} v / \tau' &\xrightarrow{\epsilon} v / \tau' && \text{if } \forall n \geq 0 (v\% \tau') \neq \perp_n . \end{aligned}$$

Usage: Try Rewrite `plus_is_0` rewrites the current goal with the lemma stating that the sum of two natural numbers is zero only if both numbers are zero. If the rewrite rule doesn't apply then the failure is caught.

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

$$\text{First } [e_1|e_2|\dots|e_n] / \tau \xrightarrow{\epsilon} \overline{\text{First}}_{\tau} [e_1|e_2|\dots|e_n] / \tau$$

with

$$\begin{aligned} \overline{\text{First}}_{\tau} [] / \tau' &\xrightarrow{\epsilon} (\text{Fail } 0) / \tau \\ \overline{\text{First}}_{\tau} [v_1|e_2|\dots|e_n] / \tau' &\xrightarrow{\epsilon} v_1 / \tau' && \text{if } \forall n \geq 0 (v_1 \% \tau') \neq \perp_n \\ \overline{\text{First}}_{\tau} [v_1|e_2|\dots|e_n] / \tau' &\xrightarrow{\epsilon} \text{First } [e_2|\dots|e_n] / \tau && \text{if } \exists n \geq 0 (v_1 \% \tau') = \perp_n . \end{aligned}$$

Usage: The proof script `First[Rewrite plus_assoc_l | Rewrite plus_assoc_r]` applies the plus left associativity rule, if it does not affect the current goal then the right version is tried. If no rewrite rule modify the current goal the proof script fails.

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

$$\text{Solve } [e_1|e_2|\dots|e_n] / \tau \xrightarrow{\epsilon} \overline{\text{Solve}}_{\tau} [e_1|e_2|\dots|e_n] / \tau$$

with

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

Usage: The proof script `Solve[Rewrite plus_assoc_l | Rewrite plus_assoc_r]` applies the plus left associativity rule, if it does not solve the current goal then the right version is tried. If no rewrite rule solve the current goal the proof script fails.

5.3 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)`.

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

The declaration of new commands simply writes:

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

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:

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

5.4 Context

The evaluation context is defined as:

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

5.5 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.

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

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

$$\begin{aligned}
\text{Assumption}\%_\tau. \Gamma, (x : A) \vdash A' \quad &= \quad \tau.\text{setLeafProven}().\text{setProgress}(\text{true}) , \\
&\text{with } A \text{ and } A' \text{ unifiable.}
\end{aligned}$$

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

The identity was introduced in [5] as a tactical, but it behaves as a tactic:

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

The same holds for the error command:

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

6 PVS's Strategies

PVS tactics and strategies are thoroughly described in [12] and [10], but as far as we know, there is no published small-step semantics of the strategy language.

6.1 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 aforementioned.

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 (\Downarrow):

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

And

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

6.2 Semantics

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

$v ::= p$.

The reduction rules for the tacticals follow.

Break \Downarrow marks the end of the command. It triggers the evaluation of the tactics and does the final proof context parameter reset:

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

Lisp conditional A lisp argument t is evaluated to determine whether the first or the second tactic argument is applied. Along with the (following) `let`, these are the only strategies that allow common lisp code within the proof language. Such a lisp code can be used for instance for computation purposes, or to access specific values of the PVS system [15].

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

Usage: `(if ** (flatten -) (split +))`: if `**` (the list of formulas in the antecedent of the current goal) is not empty then apply disjunctive simplification to the antecedent, else apply conjunctive splitting to the consequent.

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 .

$$\begin{aligned} (\text{let } ((x_1 t_1) \dots (x_n t_n)) e) / \tau &\xrightarrow{\epsilon} \\ e[x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n] / \tau &. \end{aligned}$$

Usage: `(let ((form-num car ***)) (lift-if form-num))` fetches the first formula in the consequent of the current goal, and lifts the branching structure in this formula.

Backtracking This strategy combines a branching facility triggered by the progress condition, with an error catching functionality. It applies v_1 to the current goal, if 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 sequential tactical `then` is simply defined as `(then v1 v2) = (try v1 v2 v2)`.

$$(\text{try } e_1 e_2 e_3) / \tau \xrightarrow{\epsilon} (\overline{\text{try}}_\tau e_1 e_2 e_3) / \tau$$

with

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

and

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

and the semantics of the nullary operator $\overline{\text{bt}}$ (backtrack):

$$\begin{aligned} (\overline{\text{try}}_\tau \overline{\text{bt}}) / \tau' &\xrightarrow{\epsilon} \overline{\text{bt}} / \tau \\ \overline{\text{bt}} / \tau &\xrightarrow{\epsilon} (\text{skip}) / \tau . \end{aligned}$$

Usage: `(try (flatten) (prop) (split))` applies `(flatten)` to the current goal; if it generates subgoals then the propositional simplification tactic is applied, else `(split)` is.

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.

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

with

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

Usage: `(repeat (do-rewrite))` repeatedly applies rewrite steps along the main proof branch until no progress results from it.

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

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

with

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

Usage: `(repeat* (do-rewrite))` repeatedly applies rewrite steps along all branches until no progress results from it.

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 .

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

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

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

and

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

and

$$\begin{array}{l} \overline{(\text{spread}_{\tau}^{v_0, e_1, \dots, e_n} v_n)} / \tau' \xrightarrow{\epsilon} \\ \overline{(\text{spread}_{\tau}^{v_0, e_1, \dots, e_{n-1}} v_0, e_1, \dots, e_{n-1})} / \tau \quad \text{if } \tau' = \emptyset , \end{array}$$

and

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

and

$$\begin{array}{l} \overline{(\text{spread}_{\tau}^{v_0, e_1, \dots, e_n} v_n)} / \tau' \xrightarrow{\epsilon} \\ \overline{(\text{spread}_{\tau}^{v_0, e_1, \dots, e_n, (\text{skip})} v_0, e_1, \dots, e_n, (\text{skip}))} / \tau \\ \quad \text{if } (v_n \% \tau').\text{pointNextSibling}() \neq \emptyset , \end{array}$$

Usage: `(spread (flatten) ((ground) (assert) (lift-if)))` applies the disjunctive simplification step to the current goal, then apply `(ground)` to the first generated subgoal, `(assert)` to the second and `(lift-if)` to the third.

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

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

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

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

Usage: `(branch (flatten) ((ground) (assert)))` applies the disjunctive simplification step to the current goal, then apply `(ground)` to the first generated subgoal, and `(assert)` to the rest of the subgoals.

N-ary backtracking A combination of the `try` and the `branch` strategies, `try-branch` applies e_0 to the current goal, and in case it generated subgoals it applies each of the e_i to one of the subgoals. Else it applies e' . As for `try`, this strategy catches any failure that would arise from the application of any of the e_i and backtracks.

First we store the original proof context:

$$(\text{try-branch } e_0 (e_1 \dots e_n) e') / \tau \xrightarrow{\epsilon} \overline{(\text{try-branch}_{\tau} e_0 (e_1 \dots e_n) e')} / \tau$$

and then we go for the first stage:

$$\begin{aligned} \overline{(\text{try-branch}_{\tau} v_0 (e_1 \dots e_n) e')} / \tau' &\xrightarrow{\epsilon} \overline{(\text{try-branch}_{\tau, (v_0 \% \tau')}^{e_1, \dots, e_n} e_1 \dots e_n)} / (v_0 \% \tau') \\ &\quad \text{if } (v_0 \% \tau') . \text{hasProgressed}() \\ &\quad \text{and } \forall n \geq 0 (v_0 \% \tau') \neq \perp_n \\ &\quad \text{and } \neg(v_0 \% \tau') . \text{isActiveTreeProven}() \\ \overline{(\text{try-branch}_{\tau} v_0 (e_1 \dots e_n) e')} / \tau' &\xrightarrow{\epsilon} e' / \tau \quad \text{if } \neg(v_0 \% \tau') . \text{hasProgressed}() \\ \overline{(\text{try-branch}_{\tau} v_0 (e_1 \dots e_n) e')} / \tau' &\xrightarrow{\epsilon} v_0 / \tau' \quad \text{if } \exists n \geq 0 (v_0 \% \tau') = \perp_n \\ &\quad \text{if } (v_0 \% \tau') . \text{isActiveTreeProven}() . \end{aligned}$$

The second stage corresponds to an error-catching version of “`branch`” strategy. Let l be the list $e_1 \dots e_n$:

$$\begin{aligned} &\overline{(\text{try-branch}_{\tau, \tau'}^l v_1 e_2 \dots e_n)} / \tau'' \xrightarrow{\epsilon} \\ &\overline{(\text{try-branch}_{\tau'}^l e_2 \dots e_n)} / (v_1 \% \tau'') . \text{pointNextSibling}() \\ &\quad \text{if } \forall n \geq 0 (v_1 \% \tau'') \neq \perp_n , \end{aligned}$$

and the error case:

$$\begin{aligned} &\overline{(\text{try-branch}_{\tau, \tau'}^l v_1 e_2 \dots e_n)} / \tau'' \xrightarrow{\epsilon} \overline{\text{bt}} / \tau \\ &\quad \text{if } \exists n \geq 0 (v_1 \% \tau'') = \perp_n . \end{aligned}$$

We enrich the semantics of the nullary operator $\overline{\text{bt}}$:

$$\begin{aligned} &\overline{(\text{try}_{\tau} \overline{\text{bt}})} / \tau' \xrightarrow{\epsilon} \overline{\text{bt}} / \tau \\ &\overline{(\text{try-branch}_{\tau'}^l \overline{\text{bt}} e_2 \dots e_n)} / \tau' \xrightarrow{\epsilon} \overline{\text{bt}} / \tau \\ &\overline{\text{bt}} / \tau \xrightarrow{\epsilon} (\text{skip}) / \tau . \end{aligned}$$

finally for the behaviour of the “branch” part:

$$\begin{aligned} \overline{(\text{try-branch}_{\tau, \tau}^{e_1, \dots, e_n} v_n)} / \tau'' &\xrightarrow{\epsilon} \overline{(\text{try-branch}_{\tau, \tau}^{e_1, \dots, e_{n-1}} e_1, \dots, e_{n-1})} / \tau' \\ &\quad \text{if } \tau'' = \emptyset \\ &\quad \text{or } (v_n \% \tau''). \text{pointNextSibling}() \neq \emptyset , \end{aligned}$$

and

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

and

$$\begin{aligned} \overline{(\text{try-branch}_{\tau, \tau}^{e_1, \dots, e_n} v_n)} / \tau'' &\xrightarrow{\epsilon} \overline{(\text{try-branch}_{\tau, \tau}^{e_1, \dots, e_n, e_n} e_1, \dots, e_n, e_n)} / \tau' \\ &\quad \text{if } (v_n \% \tau''). \text{pointNextSibling}() \neq \emptyset . \end{aligned}$$

Usage: `(try-branch (flatten) ((ground) (assert)) (split))` applies `(flatten)` to the current goal. If it generated subgoals it applies `(ground)` to the first of these subgoals and `(assert)` to the rest of the subgoals. Else it applies `(split)`.

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

$$\begin{aligned} (\text{Black } v) / \tau &\xrightarrow{\epsilon} (\text{skip}) / \tau \quad \text{if } \exists n \geq 0 (v \% \tau) = \perp_n \\ (\text{Black } v) / \tau &\xrightarrow{\epsilon} v / \tau \quad \text{if } \forall n \geq 0 (v \% \tau) \neq \perp_n , \end{aligned}$$

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

6.4 Context

$$\begin{aligned} E ::= & \quad [] \\ & \quad | E \mathbf{!} \\ & \quad | \overline{\text{try}_{\tau} E} e_2 e_3 \\ & \quad | \overline{\text{try}_{\tau} E} \end{aligned}$$

$$\begin{array}{l}
| (\text{spread } E (e'_1 \dots e'_n)) \\
| \overline{\text{spread}_\tau^{v_0, e_1, \dots, e_n} E e_i \dots e_n} \\
| (\text{branch } E (e'_1 \dots e'_n)) \\
| \overline{\text{branch}_\tau^{v_0, e_1, \dots, e_n} E e_i \dots e_n} \\
| (\overline{\text{try-branch}_\tau E (e'_1 \dots e'_n) e_2}) \\
| \overline{\text{try-branch}_\tau^{v_0, e_1, \dots, e_n} E e_i \dots e_n} \\
| (\text{Glass } E) \\
| (\text{Black } E) .
\end{array}$$

6.5 Tactics

The same conventions will be used as for Coq's tactics. Note that PVS does not use the error level: \perp_0 is the only error possible.

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

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

$$\begin{array}{l}
(\text{flatten})\%_\tau. \Gamma, A \wedge B \vdash C = \\
\tau.\text{addLeaves}(\Gamma, A, B \vdash C).\text{setProgress}(\text{true}) .
\end{array}$$

$$\begin{array}{l}
(\text{propax})\%_\tau. \Gamma, A \vdash B = \tau.\text{leafProven}().\text{setProgress}(\text{true}) \\
\text{if } A \text{ and } B \text{ are syntactically} \\
\text{equal.}
\end{array}$$

$$(\text{beta})\%_\tau. \Gamma \vdash (\lambda x : t)(u) = \tau.\text{addLeaves}(\Gamma \vdash t[x \leftarrow u]).\text{setProgress}(\text{true}) .$$

$$(\text{skip})\%_\tau = \tau .$$

$$(\text{fail})\%_\tau = \perp_0 .$$

$$\begin{array}{l}
(\text{skolem} * (\text{"a"}))\%_\tau. \Gamma, (\exists x : A) \vdash B = \\
\tau.\text{addLeaves}(\Gamma, A[x \leftarrow a] \vdash B).\text{setProgress}(\text{true}) .
\end{array}$$

$$\begin{array}{l}
(\text{skolem} * (\text{"a"}))\%_\tau. \Gamma \vdash (\forall x : A) = \\
\tau.\text{addLeaves}(\Gamma \vdash A[x \leftarrow a]).\text{setProgress}(\text{true}) .
\end{array}$$

7 Implementation

The previous semantics, by allowing the strategies and tacticals to be expressed in a common framework, has highlighted the differences in the two languages. The aim for this section is to present the work we did to level up the functionalities of the proof languages, allowing a common basis to be established. The following lines expose the semantics of the newly implemented strategies and tacticals, in our framework. In both cases, the languages of tacticals were used to implement the new functionalities.

7.1 Coq

The following tacticals are at the state of prototypes and are not publicly released yet.

Case-testing This tactical tests the application of its first argument. Depending on whether it progresses or fails, it selects another of its arguments to apply.

$$\text{Case-Test } e_1 e_2 e_3 e_4 / \tau \xrightarrow{\epsilon} \overline{\text{Case-Test}}_{\tau} e_1 e_2 e_3 e_4 / \tau .$$

with:

$$\begin{aligned} \overline{\text{Case-Test}}_{\tau} v_1 e_2 e_3 e_4 / \tau' &\xrightarrow{\epsilon} e_2 / \tau && \text{if } \exists n \geq 0 (v_1 \% \tau) = \perp_n \\ \overline{\text{Case-Test}}_{\tau} v_1 e_2 e_3 e_4 / \tau' &\xrightarrow{\epsilon} e_3 / \tau && \text{if } \neg(v_1 \% \tau). \text{hasProgressed}() \\ \overline{\text{Case-Test}}_{\tau} v_1 e_2 e_3 e_4 / \tau' &\xrightarrow{\epsilon} e_4 / \tau' && \text{if } (v_1 \% \tau). \text{hasProgressed}() \\ &&& \text{and } \forall n \geq 0 (v_1 \% \tau) \neq \perp_n . \end{aligned}$$

Usage: *Case-Test (Apply plus_sym) (Apply simpl_plus_l) (EApply plus_sym)* tries to apply to the current goal the plus symmetry rule. If it does not modify the current goal then it tries to apply another rule; if it fails it uses another tactic that introduces existential variables to properly define the uninstantiated variable of the rule.

pTry Corresponds to the PVS “try” strategy.

$$\text{pTry } e_1 e_2 e_3 / \tau \xrightarrow{\epsilon} \overline{\text{pTry}}_{\tau} e_1 e_2 e_3 / \tau$$

with

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

and

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

Usage: (*pTry (Cut A) Tauto Intro* applies *modus ponens* to the current goal; if it generates subgoals then the propositional simplification tactic is applied, else *Intro* is.

pTry-Branch Corresponds to the PVS “try-branch” strategy.

$$\text{pTry-Branch } e_0 [e_1 | \dots | e_n] e' / \tau \xrightarrow{\epsilon} \overline{\text{pTry-Branch}}_{\tau} e_0 [e_1 | \dots | e_n] e' / \tau$$

and

$$\begin{aligned} \overline{\text{pTry-Branch}}_{\tau} v_0 [e_1 | \dots | e_n] e' / \tau' &\xrightarrow{\epsilon} \overline{\text{pTry-Branch}}_{\tau, (v_0 \% \tau')}^{e_1, \dots, e_n} e_1 \dots e_n / (v_0 \% \tau') \\ &&& \text{if } (v_0 \% \tau'). \text{hasProgressed}() \\ &&& \text{and } \forall n \geq 0 (v_0 \% \tau') \neq \perp_n \\ &&& \text{and } \neg(v_0 \% \tau'). \text{isActiveTreeProven}() \\ \overline{\text{pTry-Branch}}_{\tau} v_0 [e_1 | \dots | e_n] e' / \tau' &\xrightarrow{\epsilon} e' / \tau && \text{if } \neg(v_0 \% \tau'). \text{hasProgressed}() \\ \overline{\text{pTry-Branch}}_{\tau} v_0 [e_1 | \dots | e_n] e' / \tau' &\xrightarrow{\epsilon} v_0 / \tau' && \text{if } \exists n \geq 0 (v_0 \% \tau') = \perp_n \\ &&& \text{if } (v_0 \% \tau'). \text{isActiveTreeProven}() . \end{aligned}$$

Let l be the list $e_1 \dots e_n$:

$$\frac{\overline{(\text{pTry-Branch}_{\tau, \tau}^l, v_1 e_2 \dots e_n) / \tau''} \xrightarrow{\epsilon}}{\overline{(\text{pTry-Branch}_{\tau}^l e_2 \dots e_n) / (v_1 \% \tau'')}} \cdot \text{pointNextSibling()} \quad \text{if } \forall n \geq 0 (v_1 \% \tau'') \neq \perp_n ,$$

and the error case:

$$\overline{(\text{pTry-Branch}_{\tau, \tau}^l, v_1 e_2 \dots e_n) / \tau''} \xrightarrow{\epsilon} \text{Idtac} / \tau \quad \text{if } \exists n \geq 0 (v_1 \% \tau'') = \perp_n .$$

Finally for the behaviour of the “branch” part:

$$\overline{(\text{pTry-Branch}_{\tau, \tau}^{e_1, \dots, e_n} v_n) / \tau''} \xrightarrow{\epsilon} \overline{(\text{pTry-Branch}_{\tau, \tau}^{e_1, \dots, e_{n-1}} e_1, \dots, e_{n-1}) / \tau'} \quad \begin{array}{l} \text{if } \tau'' = \emptyset \\ \text{or } (v_n \% \tau'') \cdot \text{pointNextSibling()} \neq \emptyset , \end{array}$$

and

$$\overline{(\text{pTry-Branch}_{\tau, \tau}^l, v_n) / \tau''} \xrightarrow{\epsilon} (\text{skip}) / (v_n \% \tau'') \cdot \text{lowerPointer}(1) \quad \begin{array}{l} \text{if } \tau'' \neq \emptyset \\ \text{and } (v_n \% \tau'') \cdot \text{pointNextSibling()} = \emptyset , \end{array}$$

and

$$\overline{(\text{pTry-Branch}_{\tau, \tau}^{e_1, \dots, e_n} v_n) / \tau''} \xrightarrow{\epsilon} \overline{(\text{pTry-Branch}_{\tau, \tau}^{e_1, \dots, e_n, e_n} e_1, \dots, e_n, e_n) / \tau'} \quad \text{if } (v_n \% \tau'') \cdot \text{pointNextSibling()} \neq \emptyset .$$

Usage: (*pTry-Branch (Cut A) ((ground) Assumption) Intro*) applies modus ponens to the current goal. If it generated subgoals it applies *ground* to the first of these subgoals and *Assumption* to the rest of the subgoals. Else it applies *Intro*.

7.2 PVS

These strategies are regrouped into an package called “Practicals”, broadcasted to the PVS users list and available online [8].

Term matching The match strategy matches a PVS term with a set of patterns. On the first match found it returns the corresponding step, properly instantiated. The implementation of the matching functionalities rely on Ben DiVito’s package for PVS [16].

$$(\text{match } t (p_i \rightarrow e_i)_{i=1}^n) / \tau \xrightarrow{\epsilon} \bigoplus_{i=1}^n \sigma_{p_i \leftarrow t} e_i / \tau .$$

Usage:

`(match (! -2 R) (("%1 / %2") -> (rewrite-lemma "ndiv_lt" -2 ("%1" "x" "%2" "b"))))`
 matches the right-hand side of formula -2 with a dividing pattern, if there is a match then it rewrites this formula with lemma `ndiv_lt` (stating that the result of the euclidian division of x by b is lower or equal to the corresponding real division).

Context matching The screen strategy matches the proof context against a set of patterns. The order of the patterns is not decisive; $\#n$ designates the formula that was matched by the n th pattern. When a match is found, it applies the corresponding step, properly instantiated. If the $\backslash-$ symbol is omitted, the pattern will be checked against consequent and antecedent formulas.

Here a simplified rule is presented, the complete rule being a simple (but space consuming) extension.

$$\frac{(\text{screen } (hp_i \vdash p_i \rightarrow e_i)_{i=1}^n) / \tau. (\dots A_j \dots \vdash B) \xrightarrow{\epsilon}}{\bigoplus_{i=1}^n \sigma_{hp_i \leftarrow A_j} \sigma'_{p_i \leftarrow B} e_i / \tau} .$$

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

$$\frac{(\text{screen } (hp_i \vdash p_i \rightarrow e_i)_{i=1}^n) / \tau. (\dots A_j \dots \vdash B) \xrightarrow{\epsilon}}{(\text{screen } (hp_i \vdash p_i \rightarrow e_i)_{i=1}^n) / \tau. (\dots A_{j-1} \dots \vdash B) .}$$

Usage:

`(screen ("%1 * %1 = 0" \- "%2 > %3" -> (rewrite "sqrt_1" "#1")) (" -> (grind)))` matches the current goal; if the antecedent contains a formula with a squared term equated to 0 and the consequent contains a formula with a “greater than” symbol, then it rewrites the formula matched by the first pattern with a lemma concerning null squares. If no match is found satisfying these conditions, it applies `(grind)`.

Fail `(throw tag)` fails with the name `tag`. Of course this is a tactic.

$$(\text{throw } t) \% \tau = \perp_t$$

Usage: `(throw "fatalError")` throws an error with the name “fatalError”.

Catch `(catch step1 tag step2)`: if `step1` throws an error whose name corresponds to `tag` then run `step2`. If no tag is provided or if the tag is the empty string then the strategy catches any error that `step1` creates. If no `step2` is provided then it is assumed to be a `(skip)`.

$$(\text{catch } e_1 t e_2) / \tau \xrightarrow{\epsilon} (\overline{\text{catch}}_{\tau} e_1 t e_2) / \tau$$

and

$$\begin{aligned} (\overline{\text{catch}}_{\tau} v_1 t e_2) / \tau' &\xrightarrow{\epsilon} e_2 / \tau && \text{if } (v_1 \% \tau') = \perp_t \\ (\overline{\text{catch}}_{\tau} v_1 t e_2) / \tau' &\xrightarrow{\epsilon} (\text{throw } u) / \tau' && \text{if } (v_1 \% \tau') = \perp_u \\ (\overline{\text{catch}}_{\tau} v_1 t e_2) / \tau' &\xrightarrow{\epsilon} v_1 / \tau' && \text{if } \forall n \geq 0 (v \% \tau') \neq \perp_0 . \end{aligned}$$

Usage: `(catch (case "x > 0") "" (split))` applies a case analysis on the current sequent, on the variable `x`. If this fails then it applies the conjunctive splitting rule.

Progressive branching `(try! step1 step2 step3)` is a “strict” try, that does not do error handling. If `step1` generates subgoals, then `step2` is applied to all of them, else `step3` is applied.

$$(\text{try! } e_1 e_2 e_3) / \tau \xrightarrow{\epsilon} (\overline{\text{try!}}_{\tau} e_1 e_2 e_3) / \tau$$

with

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

Usage: `(try! (flatten) (propax) (split))` applies `(flatten)` to the current goal; if it generates subgoals then the `(propax)` tactic is applied, else `(split)` is. If any of these tactics fail, it fails.

Extended Lisp conditional (`when lterm step1 &rest step2...stepN`): if the lisp term *lterm* evaluates in `nil` then this strategy returns (`skip`). Else it applies *step1...stepN* in a sequence along the proof's main branch (the `then@` tactical, derived from `try`, serves that purpose).

$$\begin{aligned} (\text{when } t \ e_1 \ \dots e_2) / \tau &\xrightarrow{\epsilon} (\text{skip}) / \tau \quad \text{if } t = \text{nil} \\ (\text{when } t \ e_1 \ \dots e_2) / \tau &\xrightarrow{\epsilon} (\text{then@ } v_1 \dots v_n) / \tau \quad \text{if } t \neq \text{nil} . \end{aligned}$$

Usage: (`when (done-subgoals *ps*) (skolem!)`) applies variable skolemization to the current goal if there are no proved subgoals in the proof state.

(`when* lterm step1 &rest step2...stepN`) is a slight variant from the previous strategy: if the lisp term *lterm* does not evaluate in `nil` then it applies *step1...stepN* in an all-branches sequence.

$$\begin{aligned} (\text{when } t \ e_1 \ \dots e_2) / \tau &\xrightarrow{\epsilon} (\text{skip}) / \tau \quad \text{if } t = \text{nil} \\ (\text{when } t \ e_1 \ \dots e_2) / \tau &\xrightarrow{\epsilon} (\text{then@ } v_1 \dots v_n) / \tau \quad \text{if } t \neq \text{nil} . \end{aligned}$$

Usage: (`when* (done-subgoals *ps*) (skolem!)`) applies variable skolemization to the current goal if there are no proved subgoals in the proof state.

Conditional iteration (`while lterm step1 &rest step2...stepN`): when the lisp term *lterm* is non-`nil`, this strategy applies repeatedly *step1...stepN* along the main branch of the proof.

$$(\text{while } t \ e_1 \dots e_2) / \tau \xrightarrow{\epsilon} (\text{repeat } (\text{when } t \ e_1 \dots e_n)) / \tau .$$

Usage: (`while (pending-subgoals *ps*) (skolem!)`) applies variable skolemization to the current goal, and then to each of the first subgoal generated, as long as there are unproved subgoals in the proof state and that the proof progresses.

The strategy (`while* lterm step1 &rest step2...stepN`) behaves just as `while`, but possibly repeats *step1...stepN* on all the branches of the proof.

$$(\text{while } t \ e_1 \dots e_2) / \tau \xrightarrow{\epsilon} (\text{repeat* } (\text{when } t \ e_1 \dots e_n)) / \tau .$$

Usage: (`while* (pending-subgoals *ps*) (skolem!)`) applies variable skolemization to the current goal, and then simultaneously to all of the subgoals generated, as long as there are unproved subgoals in the proof state and that the proof progresses.

Iteration (`for lint step`): here *lint* is a lisp integer. This strategy repeats *step*, *lint* times, along the main branch of the proof. If *lint* is negative, the strategy is equivalent to the (`repeat step`) strategy.

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

with

$$\begin{aligned} (\overline{\text{for}}_e 0 \ e) / \tau &\xrightarrow{\epsilon} (\text{skip}) / \tau \\ (\overline{\text{for}}_e k \ v) / \tau &\xrightarrow{\epsilon} (\overline{\text{for}}_e (k-1) \ e) / (v\% \tau). \text{raisePointerToLeaf}() \\ &\quad \text{if } \forall n \geq 0 \ (v\% \tau) \neq \perp_n \\ &\quad \text{and } \neg(v\% \tau). \text{isActiveTreeProven}() \\ (\overline{\text{for}}_e k \ v) / \tau &\xrightarrow{\epsilon} v / \tau \quad \text{if } \exists n \geq 0 \ (v\% \tau) = \perp_n \\ &\quad \text{or } (v\% \tau). \text{isActiveTreeProven}() \\ (\overline{\text{for}}_e k \ v) / \tau &\xrightarrow{\epsilon} (\text{repeat } v) / \tau \quad \text{if } n < 0 . \end{aligned}$$

Usage: (`for 3 (beta)`) applies the β -reduction rule three times, first on the current goal and then on the first of the generated subgoals.

The strategy (`for* lint step`) behaves as a `for`, but applies *step* on all the branches of the proof.

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

with

$$\begin{aligned}
(\overline{\text{for}_e} 0 e) / \tau &\xrightarrow{\epsilon} (\text{skip}) / \tau \\
(\overline{\text{for}_e} k v) / \tau &\xrightarrow{\epsilon} (\overline{\text{for}_e} (k-1) e) / (v\% \tau) \\
&\quad \text{if } \forall n \geq 0 (v\% \tau) \neq \perp_n \\
&\quad \text{and } \neg(v\% \tau).\text{isActiveTreeProven}() \\
(\overline{\text{for}_e} k v) / \tau &\xrightarrow{\epsilon} v / \tau \quad \text{if } \exists n \geq 0 (v\% \tau) = \perp_n \\
&\quad \text{or } (v\% \tau).\text{isActiveTreeProven}() \\
(\overline{\text{for}_e} k v) / \tau &\xrightarrow{\epsilon} (\text{repeat } v) / \tau \quad \text{if } n < 0 .
\end{aligned}$$

Usage: `(for* 3 (beta))` applies the β -reduction rule three times, first on the current goal and then on all of the generated subgoals.

First tactic to succeed Applies the first of the steps that does not fail. If no step fulfill such a condition, the strategy fails.

$$(\text{first } (e_1 e_2 \dots e_n)) / \tau \xrightarrow{\epsilon} \overline{\text{First}}_{\tau} [e_1 | e_2 | \dots | e_n] / \tau$$

with, as for the `Coq` operator,

$$\begin{aligned}
\overline{\text{First}}_{\tau} [] / \tau' &\xrightarrow{\epsilon} (\text{Fail } 0) / \tau \\
\overline{\text{First}}_{\tau} [v_1 | e_2 | \dots | e_n] / \tau' &\xrightarrow{\epsilon} v_1 / \tau' \quad \text{if } \forall n \geq 0 (v_1\% \tau') \neq \perp_n \\
\overline{\text{First}}_{\tau} [v_1 | e_2 | \dots | e_n] / \tau' &\xrightarrow{\epsilon} \text{First } [e_2 | \dots | e_n] / \tau \quad \text{if } \exists n \geq 0 (v_1\% \tau') = \perp_n .
\end{aligned}$$

Usage: `(first (case "y > 0") (bddsimp) (skip))` tries to apply the case analysis command to the current goal, if it fails it tries the propositional simplification. If this also fails, it applies the `(skip)` tactic (which cannot fail).

First tactic to solve This strategy selects and applies the first of its arguments that will prove the current goal. If it has no such argument, it fails.

$$(\text{solve } (e_1 e_2 \dots e_n)) / \tau \xrightarrow{\epsilon} \overline{\text{Solve}}_{\tau} [e_1 | e_2 | \dots | e_n] / \tau$$

with, as in `Coq`,

$$\begin{aligned}
\overline{\text{Solve}}_{\tau} [] / \tau' &\xrightarrow{\epsilon} \text{Fail } 0 / \tau \\
\overline{\text{Solve}}_{\tau} [v_1 | e_2 | \dots | e_n] / \tau' &\xrightarrow{\epsilon} v_1 / \tau' \quad \text{if } (v_1\% \tau').\text{isActiveTreeProven}() \\
\overline{\text{Solve}}_{\tau} [v_1 | e_2 | \dots | v_n] / \tau' &\xrightarrow{\epsilon} \text{Solve } [e_2 | \dots | e_n] / \tau \\
&\quad \text{if } \neg(v_1\% \tau').\text{isActiveTreeProven}() .
\end{aligned}$$

Usage: `(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 tries the propositional simplification. If this also fails to completely prove the current goal, it fails.

Not skip... `(piks)` is not a strategy as we defined them since it does not act as a combinator of tactics, but more as a tactic. It does not do anything special, but does not either trigger the "No change on..." reaction of PVS. Basically, `(piks)` is used in strategy writing to deceive the progress-testing strategies.

$$(\text{piks})\% \tau = \tau.\text{setProgress}(\text{true})$$

Usage: `(try (piks) (flatten-disjunct) (skip))` is a simple way to catch any error generated by the application of the controlled disjunctive simplification rule.

8 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} [5], and for PVS by the Prover Guide [15]. Future work will also investigate the possibility 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 NuPrI) in this framework. In addition, since tactics behave as rewriting rules over a proof context, 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 conservatively extendable, and should allow for an efficient and –hopefully– not too complicated continuation.

The implementation effort should be conducted synchronously with the future developments of the semantics, and we will continue to maintain and enhance the existing code.

Finally, this work sets the very basis for an unified representation of PVS's strategies and Coq's tacticals, as well as simple tactics from both sides. On the long term, such a common proof language would allow for proof portability, double-checking, prover-relevancy modularization, i.e., an overall improved flexibility and interoperability.

9 Acknowledgment

I would like to warmly thank the LogiCal and the Formal Methods team and especially Gilles Dowek, Cesar Munoz, Benjamin Werner, Hugo Herbelin and Alfons Geser for their continuous help and support. Many thanks to N. Shankar for clarifying the semantics of PVS's try to us and providing several examples of its use. Thanks to François Pottier and Didier Remy for their help on the small step version of the semantics of Match. This work was supported by ENS de Cachan, INRIA FUTURS and the National Institute of Aerospace (under NASA Cooperative Agreement NCC-1-02043).

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] James L. Caldwell and John Cowles. Representing Nuprl Proof Objects in ACL2: toward a proof checker for Nuprl.
- [3] David Carlisle, Scott Pakin, and Alexander Holt. *The Great, Big List of L^AT_EX Symbols*, February 2001.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *lncs*. sv, 1979.
- [8] Florent Kirchner. Programmation tacticals, 2003. <http://research.nianet.org/fm-at-nia/Practicals/>.
- [9] C. Muñoz and M. Mayero. Real automation in the field. Technical Report NASA/CR-2001-211271 Interim ICASE Report No. 39, ICASE-NASA Langley, dec 2001.

- [10] César Muñoz. Strategies in PVS. Lecture notes, 2002. National Institute of Aerospace.
- [11] Tobias Oetiker. *The Not So Short Introduction to L^AT_EX2_ε*, January 1999.
- [12] 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.
- [13] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.
- [14] François Pottier. Typage et Programmation. Lecture notes, 2002. DEA PSPL.
- [15] 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.
- [16] Ben L. Di Vito. A PVS prover strategy package for common manipulations. Technical Report TM-2002-211647, Langley Research Center, Hampton, VA, April 2002.
- [17] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.