The Proof Monad[☆]

Florent Kirchner[a,*], César Muñoz[b]

[a] *LIX, INRIA & SRI International, Menlo Park CA 94025, USA*
[b] *NASA Langley Research Center, Hampton, Virginia, USA*

Abstract

A formalism for expressing the operational semantics of proof languages used in procedural theorem provers is proposed. It is argued that this formalism provides an elegant way to describe the computational features of proof languages, such as side effects, exception handling, and backtracking. The formalism, called *proof monads*, finds its roots in category theory, and in particular satisfies the monad laws. It is shown that the framework's monadic operators are related to fundamental tactics and strategies in procedural theorem provers. Finally, the paper illustrates how proof monads can be used to implement semantically clean control structure mechanisms in actual proof languages.

Keywords: proof languages, deductive strategies, category theory, monadic structures

1. Introduction

Proof script languages of interactive theorem provers such as ACL2 [16], Coq [5], HOL [12], or PVS [32] are similar to interpreted programing languages in that they consist of a sequence of instructions that are typically processed in a read-eval-print loop. In the case of proof languages, instructions describe proof steps that gradually construct a *proof object*. Two types of instructions are traditionally distinguished in procedural theorem provers: *tactics*, which are the elementary deduction rules of the prover's logic; and *strategies*, which are tactic combinators that provide the control structure mechanism of the language. The proof engine, which plays the role of a virtual machine, reacts to the application of these instructions by modifying the state of the proof object, e.g., opening new goals in the proof tree, closing some branches, changing the focus to a different branch, etc.

[☆] This work was supported by the National Aeronautics and Space Administration at Langley Research Center under the Research Cooperative Agreement No. NCC-1-02043 awarded to the National Institute of Aerospace.

[*] Corresponding author
Email addresses: florent.kirchner@lix.polytechnique.fr (Florent Kirchner), cesar.a.munoz@nasa.gov (César Muñoz)

The last few years have seen an increasing interest on the development of proof languages for theorem provers and their formal semantics. Most notably, Sacerdoti et al. [8], Delahaye [10], and Martin et al. [26] propose various control structure languages, and use diverse semantic frameworks to formalize them. These approaches abstract away the human-prover interaction of procedural theorem provers, and focus on the representation of the proof object. This paper proposes a different approach to the semantics of proof languages that considers not only the proof object but also the outcome of the interaction with the theorem prover. This approach yields a formalism that is arguably closer in spirit to the read-eval-print loop model of interactive theorem provers.

Take for instance λ-terms, which are well-known to represent complete proof trees in constructive logic through the Curry-de Bruijn-Howard isomorphism. The encoding of incomplete proof trees requires non-trivial extensions to the λ-calculus with open terms, for example via metavariables and explicit substitutions [25, 31, 28, 15]. Unfortunately, open terms are not well-suited to capture the non-logical information that is required to express the semantics of the interactive proof construction process.

The limitations of the usual proof representation is illustrated by an attempt to formalize the semantics of one of PVS's most feature-rich strategies: `try` [2]. Informally, the script `(try` $\mathtt{t}_1$ $\mathtt{t}_2$ $\mathtt{t}_3$`)` applies its first argument $\mathtt{t}_1$ to the goal, and if it generates subgoals, it applies $\mathtt{t}_2$ to the subgoals; otherwise, it applies $\mathtt{t}_3$. Furthermore, if $\mathtt{t}_2$ fails, for example, because $\mathtt{t}_2$ = `(fail)`, then it initiates a backtracking sequence, which is propagated until it is evaluated as the first member of another `try` construct, in which case it evaluates its third argument. The semantics of `try` is given in terms of five different types of state information: *failure*, *success*, *skip*, *subgoals*, *backtrack*. Using $|.|$ as a semantic evaluator, the behavior of `try` can be expressed as follows:

$$
|(\mathtt{try}\ \mathtt{t}_1\ \mathtt{t}_2\ \mathtt{t}_3)| = \left|\begin{array}{lll}
|\mathtt{t}_3| & \text{if} & |\mathtt{t}_1| \in \{\mathit{skip}, \mathit{backtrack}\} \\
|\mathtt{t}_1| & \text{if} & |\mathtt{t}_1| \in \{\mathit{failure}, \mathit{success}\} \\
\mathit{backtrack} & \text{if} & |\mathtt{t}_1| = \mathit{subgoals}, \\
 & & |\mathtt{t}_2| \in \{\mathit{failure}, \mathit{backtrack}\} \\
\mathit{subgoals} & \text{if} & |\mathtt{t}_1| = \mathit{subgoals}, \\
 & & |\mathtt{t}_2| \in \{\mathit{skip}, \mathit{subgoals}\} \\
\mathit{success} & \text{if} & |\mathtt{t}_1| = \mathit{subgoals}, \\
 & & |\mathtt{t}_2| = \mathit{success}\ ,
\end{array}\right.
$$

where

$$
\begin{aligned}
|\mathtt{(skip)}| &= \mathit{skip} \\
|\mathtt{(fail)}| &= \mathit{failure}\ .
\end{aligned}
$$

The information required to deal with the semantics of `try` is not part of the actual proof tree. In fact, in this explanation of the behavior of `try`, the proof tree is not even mentioned. This hints at an approach to express the interactive nature of proof script languages: the notion of proof object can be extended to include non-logical information such as a pointer to the subgoals that are

currently open and feedback information for the user about the outcome of the proof steps that have been applied. Yet the question remains, whether there exists a consistent mathematical framework to construct these objects and what are their properties.

Monads are elements of the theory of categories introduced in computer science to deal with non-functional constructs in purely functional programming languages. The idea behind monads is to bundle extra information into the objects manipulated by functions. For instance, a function f mapping an element a to an element b could be extended to a function f' on pairs that also increases a counter x: $f'(a, x) = (b, x + 1)$. In this simplistic example, the "bundling" that enables the counting side-effect is done by using a pair of values. While this example illustrates the case of imperative side-effects, monads generalize this approach to any type of side-effects, such as exceptions, input-output, continuations, and non-determinism [39].

In this paper, it is shown how adding a monadic structure to a generic proof object provides a mathematical structure for expressing the formal semantics of non-trivial procedural proof languages. Section 2 reviews the core structure of proofs and defines a representation of proof trees that does not rely on any particular data structure. Section 3 provides a primer on category theory, before defining proof monads and their mathematical properties. Section 4 illustrates the use of proof monads by defining the semantics of a sample, parametric proof language. Section 5 discusses the concrete implementation of proof monads in two theorem provers: PVS and FSP [20]. Finally, Section 6 discusses the choices that were made and covers related work.

2. Proof Trees

Given a formal language $\mathcal{L}$ and an entailment relation $\vdash$ defined by a set of deduction rules [4], a sequent is written $\Gamma \vdash \Delta$, where Γ, Δ are sets of formulas of $\mathcal{L}$. A proof is generally organized in a tree structure, called a *proof tree*, in which nodes are labelled with sequents. A *complete proof tree* is a tree of sequents where the leaves are axioms of $\mathcal{L}$ and the connections between children and parent nodes are justified by the $\vdash$-deduction rules. Therefore, the root of a complete proof tree is a theorem in $\mathcal{L}$. In procedural theorem proving, a complete proof tree is gradually constructed from an *incomplete* tree, i.e., a tree where the leaves may not be axioms but open goals that have to be proven in order to declare the root formula a theorem. Henceforth, the term *proof tree* refers to both complete and incomplete proof trees and will be denoted by lower case letters $x, y, \ldots$

Definition 1 (Proof tree). A proof tree is a set of sequents of $\mathcal{L}$-formulas, organized in a tree structure, where instances of $\vdash$-deduction rules constitute the links between nodes. As usual, the nodes of a tree can be referenced by their *position*. The set of *positions* of a tree x is denoted $\mathcal{P}_x$. If $\alpha \in \mathcal{P}_x$, then $x|_\alpha$ denotes the subtree of x such that the node at position α is the root.

Furthermore, the set of *open goals* of x, denoted $\mathcal{O}_x$, is the set of all positions $\alpha \in \mathcal{P}_x$ such that $x|_\alpha$ is an incomplete leaf of x.

The concrete representation of proof trees varies among different theorem provers. For instance, PVS uses a collection of tactic calls, whereas Coq and Alf [25] use variants of open λ-terms. Definition 1 considers an abstract representation of proof trees, which can be instantiated by different datatypes. For instance, the mapping between λ-terms and proof trees follows the Curry-de Bruijn-Howard isomorphism. Moreover, the process carried out by recursive command calls can be abstracted as the construction of a proof tree. Mappings between Coq and PVS tactics and enhanced λ-terms have been proposed in [18].

Since proof development in an interactive theorem prover is a sequential activity, an order relation between the open goals of a proof tree is also needed. This ordering can be chosen arbitrarily, but typically mirrors the various tree traversal algorithms. From now on, it is assumed that there is a total order $\prec$ on $\mathcal{O}_x$.

Procedural theorem provers limit the scope of a proof step application to a subset of open goals. The goals in this subset are called *current goals*, and they usually change after the application of a proof step. Although open goals are often scattered throughout the proof tree, the current goals are usually localized in a branch of the tree. In particular, the current goals are determined by the position of their nearest common ancestor. This position is called the *current index* and the children of the node in the current index are assumed to be either complete subtrees or open goals.

Definition 2 (Indexed proof tree). An *indexed proof tree* $x[\alpha]$ is a proof tree x parametrized by a position $\alpha \in \mathcal{P}_x$. All open goals at position $\beta \in \mathcal{O}_{x|_\alpha}$ are called *current goals*. The type τ is the type of indexed proof trees.

Remark that using indices makes for a very compact representation of current goals. Most modern proof assistants are directly compatible with this abstraction. Others (e.g., [3]) require more fine-grained control over the selection of current goals, which can be achieved by directly indexing the leaves of the proofs and using more complex proof instructions to manipulate them.

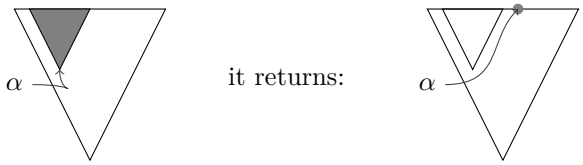
Finally, the following operations are defined on indexed proof trees.

Definition 3 (Leaf). The partial function $\downarrow_i$ of type $\tau \to \tau$ maps an indexed proof tree $x[\alpha]$ to the indexed proof tree $x[\beta]$ such that β is the i-th element of $\mathcal{O}_{x_\alpha}$ (under $\prec$), when it exists.

Definition 4 (Sibling). The partial function $\rightsquigarrow$ of type $\tau \to \tau$ maps an indexed proof tree $x[\alpha]$ to the proof tree $x[\beta]$ such that β is the minimum element of $\mathcal{O}_x \setminus \mathcal{O}_{x|_\alpha}$ (under $\prec$), when it exists.

These next graphics illustrate the semantics of these functions. The big triangle represents a proof, of which the small triangle designates a subproof. The grayed area denotes the subtree whose open goals are the current goals. The function $\downarrow_i$ provides a way to displace the current index to a particular subgoal within the set of current goals. For instance, $\downarrow_1$ transforms the indexed proof tree:

On the other hand, the function $\rightsquigarrow$ would displace the index to the next open subgoal outside the set of current goals. For instance, given the indexed proof tree:



3. Categories, Monads, and Proof Monads

Adding *computational effects* such as side-effects, exceptions or input/output to pure functional languages is an important step towards the usability of these languages. While some languages such as OCaml or Scheme augment their semantics with adhoc constructions and rules, others such as Haskell or Gallina translate these features into pure functional constructs. One simple idea is to augment the objects that functions manipulate with computational objects such as a memory state, an exception stack, or an input/output socket. Moggi [29, 30] proposed the mathematical structure of *monads* to achieve this.

3.1. Categories

Monads are defined as a special kind of categories. For completeness, this section briefly reviews the theory of categories, using Buronni's graphical presentation [6]. For topical introductions to the subject, see [23, 1].

Definition 5 (Graph). A graph is defined as a quadruplet (G_0, G_1, s, t), where G_0 is the set of *objects*, G_1 is the set of *morphisms* of the graph, and s and t are two applications such that:

$$G_1 \underset{t}{\overset{s}{\rightrightarrows}} G_0$$

For any *morphism* f, the *objects* $x = sf$ and $y = tf$ are called respectively the *source* and the *target* of f, and we note $f : x \to y$.

Note that objects and morphisms are also called *nodes* and *arrows*, respectively. In reference to algebra, the objects sf and tf are also called the *domain* and *co-domain* of f, respectively.

Definition 6 (Category). A category is defined as a triple $(G, \mathrm{id}, \circ)$, where $G = (G_0, G_1, s, t)$ is a graph and id and $\circ$ are morphism constructors, called respectively *identity* and *composition*:

$$G_0 \xrightarrow{\mathrm{id}} G_1 \qquad\qquad G_1 \times G_1 \xrightarrow{\circ} G_1$$

The elements of a category verify:

- For any morphisms $f, g \in G_1 \times G_1$ such that $tf = sg$,

$$s(f \circ g) = sf \qquad\qquad t(f \circ g) = tg \ .$$

- For any object x of G_0,

$$s(\mathrm{id}\ x) = t(\mathrm{id}\ x) = x \ .$$

- For any morphisms f, g, h of $G_1 \times G_1 \times G_1$ such that $tf = sg$ and $tg = sh$,

$$(f \circ g) \circ h = f \circ (g \circ h) \ .$$

- For any morphism $f : x \to y$ of G_1

$$f \circ (\mathrm{id}\ y) = (\mathrm{id}\ x) \circ f = f \ .$$

Definition 7 (Functor). Functors are morphisms between categories. A functor $F : C \to C'$ between two categories is defined by a transformation $h : G \to G'$ between their corresponding graphs such that:

- For any object x of G,

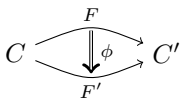
$$h(\mathrm{id}\ x) = \mathrm{id}(hx) \ .$$

- For any morphisms f, g of G such that $tf = sg$,

$$h(f \circ g) = (hf) \circ (hg) \ .$$

Composition between functors is defined by the composition between their corresponding graph transformations.

Definition 8 (Natural transformation). Natural transformations are morphisms between functors. Given two categories C, C' and two functors $F, F' : C \to C'$, a natural transformation $\phi : F \to F'$ is a family of morphisms on C' indexed by the objects of C



such that:

- For any object x of C,

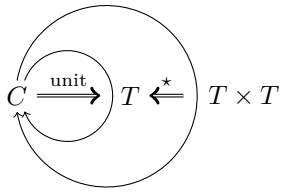
$$s\phi_x = Fx \qquad\qquad t\phi_x = F'x \ .$$

- For any morphism $f : x \to y$ of C,

$$(Ff) \circ \phi_y = \phi_x \circ (F'f) \ .$$

3.2. Monads

Definition 9 (Monad). A monad is a quadruplet $(C, T, \text{unit}, \star)$ where $C = (G, \text{id}, \circ)$ is a category, $T : C \to C$ is a functor, and $\text{unit} : \text{id} \to T$ and $\star : T \to (\text{id} \to T) \to T$ are natural transformations



with the following properties:

- For any object x of C and morphism $f : C \to TC$,

$$\text{unit}\, x \star f = fx \ .$$

- For any object m of TC,

$$m \star \text{unit} = m \ .$$

- For any object m of TC and morphisms $f, g : C \to TC$,

$$m \star \lambda x.((fx) \star g) = (m \star f) \star g \ .$$

The natural transformations unit and $\star$ are commonly referred to as *monadic operators*.

In order to represent computational effects with monads, one has to encode programs as mappings from a category of *values* to the monad of *computations* defined over this category. Thus, assuming that the types A and B are objects from the category of values, and $(T, \text{unit}, \star)$ is a monad, then a program is represented as a morphism $A \to TB$. The monadic operator $\text{unit} : A \to TA$ initializes the system by turning a value into its trivial computation counterpart, and the operator $\star : TA \to (A \to TB) \to TB$ allows a program of type $A \to TB$ to be applied to a computation of type TA.

Of course, there are as many choices for T as there are computational effects and combinations of these. The following example illustrates the use of monads to describe the computational effects of exceptions in a programming language.

Example 1 (The exception monad). Let $\mathcal{P}$ be a purely functional programming language and $C_{\mathcal{P}}$ the category of its values. Exceptions can be added to $\mathcal{P}$ by defining a monad $(C_{\mathcal{P}}, T, \text{unit}, \star)$ such that for any program p,

$$Tp = \left| \begin{array}{l} \mathsf{raise}\, e \\ \mathsf{return}\, p \end{array} \right.$$

where raise and return are two new datatypes of $\mathcal{P}$. Note that the category of program values $C_{\mathcal{P}}$ is extended to encompass these datatypes. The natural transformations unit and $\star$ are defined as:

$$\begin{array}{rl} \text{unit} = & \lambda a.\mathsf{return}\, a \\ \star = & \lambda m.\lambda f.\text{match } m \text{ with} \\ & \left| \begin{array}{lcl} \mathsf{raise}\, e & \mapsto & \mathsf{raise}\, e \\ \mathsf{return}\, a & \mapsto & f a \end{array} \right. \end{array}$$

3.3. Proof Monads

Definition 10 (Proof monad). Assume $\mathcal{L}$ is a formal language and $\vdash$ is an entailment relation operating on formulas of $\mathcal{L}$. Assume τ is the type of indexed proofs in this formalism. Let C_τ be the category of proof values and $(\mathcal{M}, \text{unit}, \star)$ be a monad over C_τ such that for all indexed proofs $x[\alpha]$:

$$\mathcal{M}\, x[\alpha] = \left| \begin{array}{l} x[\mathsf{success}] \\ x[\mathsf{subgoals}\ b\ \alpha] \\ \mathsf{exception}\ s \end{array} \right.$$

where b is a Boolean value, and s is an arbitrary symbol. Let match be a destructor of this datatype. The monadic operators are defined as follows:

$$\begin{array}{rl} \text{unit} = & \lambda x[\alpha].x[\mathsf{subgoals}\ \texttt{false}\ \alpha] \\ \star = & \lambda m.\lambda f.\text{match } m \text{ with} \\ & \left| \begin{array}{l} x[\mathsf{subgoals}\ b\ \alpha] \mapsto \text{match } f\, x[\alpha] \text{ with} \\ \qquad \left| \begin{array}{l} y[\mathsf{subgoals}\ b'\ \beta] \mapsto y[\mathsf{subgoals}\ (b \oplus b')\ \beta] \\ * \mapsto f\, x[\alpha] \end{array} \right. \\ * \mapsto m \end{array} \right. \end{array}$$

where $\oplus$ stands for Boolean disjunction.

Note that the proof monad extends at the same time the type of indices, with the special symbols success and subgoals, and the type of indexed proof trees, with an exception proof-tree state exception. The symbols success, subgoals, and exception constitute the feedback information of the proof system, i.e., the outcome of a proof instruction:

- subgoals b indicates whether or not subgoals have been generated by the instruction. By convention subgoals `false` means that no subgoals were generated;

- success indicates that the instruction has discharged (proven) all the current goals;
- exception s indicates that the instruction has raised an exception, labelled s. Exceptions are raised by the user (through the use of appropriate tactics) or by the proof engine (if a tactic fails to apply correctly).

As for the monadic operators unit and $\star$, the former can be understood as a neutral instruction that does not modify the proof tree, while the latter composes the outcome of an instruction with the application of a second.

Also note that, in order for this development to match the monadic requirements, it is assumed that a categorical interpretation of $(\mathcal{L}, \vdash)$ exists. Such an interpretation, called the Lambek-Lawvere isomorphism [21, 24], was first introduced in the scope of intuitionistic propositional logic, and was later generalized to systems of increasing complexity [22, 33, 34, 35, 38, 14].

Propositions 1 and 2 ensure that the monadic operators of proof monads are correctly defined.

Proposition 1. *The operators* $\star$ *and* unit *satisfy the left and right unit properties:*

$$\forall x[\alpha] : \tau, \forall f : \tau \to \mathcal{M}\,\tau, (\text{unit } x[\alpha]) \star f = f\,x[\alpha] \tag{1}$$

$$\forall m : \mathcal{M}\,\tau, m \star \text{unit} = m \tag{2}$$

Proof. The proofs of these two properties are easy. Formula (1) is proven by case analysis on the outcome of f. Formula (2) is proven by case analysis on m. Both proofs use the fact that the Boolean disjunction with `false` is the identity.

Proposition 2. *The operator* $\star$ *is associative:*

$$\forall m : \mathcal{M}\,\tau, \forall f_1, f_2 : \tau \to \mathcal{M}\,\tau, m \star \lambda x^{\tau}.((f_1\ x) \star f_2) = (m \star f_1) \star f_2 \tag{3}$$

Proof. The proof is carried by case analysis on m, and on the outcome of f_1 and f_2. The associativity of $\oplus$ concludes the proof.

Additional *map* and *join* operators can also be defined. They are usually seen as a decomposition of the $\star$ operator: $m \star k = \mathit{join}\,(\mathit{map}\,k\,m)$.

Definition 11 (Map). The operator *map* lifts a function on proof trees to a function on computations.

$$\begin{aligned}
&\mathit{map} : (\tau \to \tau) \to (\mathcal{M}\,\tau \to \mathcal{M}\,\tau)\\
&\mathit{map} = \lambda f.\lambda m.m \star \lambda x.\text{unit}\,(f\,x)\\
&\qquad = \lambda f.\lambda m.\text{match } m \text{ with}\\
&\qquad\qquad \left|\begin{array}{l} x[\mathsf{subgoals}\ b\ \alpha] \mapsto x[\mathsf{subgoals}\ b\ (f\,\alpha)]\\ * \mapsto m \end{array}\right.
\end{aligned}$$

Definition 12 (Join). The operator *join* flattens two layers of information into one.

$$\begin{aligned} &\mathit{join} : \mathcal{M}\,(\mathcal{M}\,\tau) \to \mathcal{M}\,\tau \\ &\mathit{join} = \lambda m.m \star \lambda x.x \\ &\quad = \lambda m.\text{match } m \text{ with} \\ &\qquad \left| \begin{array}{l} x[\mathsf{subgoals}\ b\ \alpha] \mapsto x[\alpha] \\ * \mapsto m \end{array} \right. \end{aligned}$$

4. Semantics of a Proof Language

This section illustrates the use of proof monads by specifying the semantics of a generic proof language. In this language, instructions are split into a logic-dependent set of tactics T, and a parametrized logic-agnostic set of strategies PRF(T). Let $(\mathcal{M}, \text{unit}, \star)$ be a monad over the category C_τ, defined in accordance with Definition 10.

Consider a logical framework with a minimalistic propositional sequent calculus.

Definition 13 ($\mathcal{L}_{min}$). The syntax of the minimalistic propositional logic $\mathcal{L}_{min}$is defined as follows:

$$A, B = \top \ \mid \ f \ \mid \ A \Rightarrow B$$

where f ranges over a set of propositional variables. Proof terms of this logic are given as Curien and Herbelin's $\bar{\lambda}\mu\tilde{\mu}$-terms [9] according to the following syntax:

$$\begin{aligned} c &= (v \| e) \\ v &= \chi \ \mid \ \ltimes \ \mid \ \lambda\chi^A.v \ \mid \ \mu\omega^A.c \\ e &= \omega \ \mid \ v \cdot e \ \mid \ \tilde{\mu}\chi^A.c \end{aligned}$$

where c denotes a command in the $\bar{\lambda}\mu\tilde{\mu}$-calculus, v denotes a term, and e denotes an environment. Additionally, χ and ω range over sets of term and environment variables, respectively, and $\ltimes$ is a constant that inhabits $\top$. Note that the $\bar{\lambda}\mu\tilde{\mu}$-terms used in this example are by no means central to the topic of this paper: their inclusion is solely meant to illustrate sequent labelling. For the interested reader, more details about the term structure, calculus and proof-term isomorphism are available in [9].

Sequents have the following form:

$$\Gamma; e : A \vdash \Delta \qquad\qquad \Gamma \vdash v : A; \Delta$$

Incomplete proofs are represented using metavariables at the level of proof terms. The sequent calculus is specified in Figure 1, in the form of a labelled rewriting system on (possibly incomplete) proof instantiations and derivations. This particular formalisation of a proof calculus as a rewriting system closely replicates the usual proof instantiation process (e.g., [31]), and is further developed in [17].

$$\Gamma; \Xi_0 : A \vdash \chi : A \xrightarrow{\mathrm{ax}_{\mathcal{L}}\ \chi} \frac{}{\Gamma; \chi : A \vdash \chi : A}$$

$$\Gamma, \chi : A \vdash X_0 : A \xrightarrow{\mathrm{ax}_{\mathcal{R}}\ \chi} \frac{}{\Gamma, \chi : A \vdash \chi : A}$$

$$\Gamma \vdash X_0 : \mathtt{true} \xrightarrow{\mathrm{true}_{\mathcal{R}}} \frac{}{\Gamma \vdash \ltimes : \mathtt{true}}$$

$$\Gamma; \Xi_0 : A \Rightarrow B \vdash C \xrightarrow{\Rightarrow_{\mathcal{L}}} \frac{\Gamma \vdash X_1 : A \qquad \Gamma; \Xi_1 : B \vdash C}{\Gamma; X_1 \cdot \Xi_1 : A \Rightarrow B \vdash C}$$

$$\Gamma \vdash X : A \Rightarrow B \xrightarrow{\Rightarrow_{\mathcal{R}}\ \chi} \frac{\Gamma, \chi : A \vdash X_1 : B}{\Gamma \vdash \lambda\chi^{A}.X_1 : A \Rightarrow B}$$

$$\Gamma; \Xi_0 : A \vdash C \xrightarrow{\mathrm{cut}_{\mathcal{L}}\ \chi\ B} \frac{\Gamma, \chi : A \vdash X_1 : B \qquad \Gamma, \chi : A; \Xi_1 : B \vdash C}{\Gamma; \tilde{\mu}\chi^{A}.(X_1 \| \Xi_1) : A \vdash C}$$

$$\Gamma \vdash X_0 : A \xrightarrow{\mathrm{cut}_{\mathcal{R}}\ \omega\ B} \frac{\Gamma \vdash X_1 : B \qquad \Gamma; \Xi_1 : B \vdash \omega : A}{\Gamma \vdash \mu\omega^{A}.(X_1 \| \Xi_1) : A}$$

Figure 1: Sequent calculus of the minimalistic propositional logic

Unlike Curien and Herbelin's sequents, which can contain commands, their simplified version presented here only includes terms and environments. This is achieved by collating, in Curien and Herbelin's calculus, derivation rules that introduce commands with rules that eliminate them.

Definition 14 (T_{min}). For each rewrite rule in Figure 1, an element t in the set of tactics $T_{min}$is defined. The syntax for each tactic is derived from the corresponding rule label:

$$t = \mathrm{ax}_{\mathcal{L}}\ \chi \mid \mathrm{ax}_{\mathcal{R}}\ \chi \mid \mathrm{true}_{\mathcal{R}} \mid \Rightarrow_{\mathcal{L}} \mid \Rightarrow_{\mathcal{R}}\ \chi \mid \mathrm{cut}_{\mathcal{L}}\ \chi\ B \mid \mathrm{cut}_{\mathcal{R}}\ \chi\ B$$

where χ and B are respectively a proof term variable and a well-formed formula. The application of a tactic to a proof tree, denoted $\langle t, x[\alpha] \rangle$ builds an object in $\mathcal{M}\,\tau$, as described below.

The proof language PRF is a variation of the original proof language of the LCF system, which includes most of the features available in modern theorem provers such as Coq, Isabelle, and PVS.

Definition 15 (PRF). Given a tactic language T, the proof language PRF(T) for the minimalistic logic consists of the following tactics and strategies:

$$\begin{aligned} s &= \mathtt{postpone} \mid \mathtt{idtac} \mid i\ ;\ i \mid [\ i_{\mathrm{list}}\ ] \mid i\ .\ i \mid \square \\ i &= t \mid s \\ i_{\mathrm{list}} &= i, \ldots, i \end{aligned}$$

where $t \in T$. Note that the nonterminal i_{list} denotes a comma-separated list of arbitrary length. The notation $\langle\, , \rangle$ is extended to denote the application of all instructions (tactics and strategies) to proof trees, where $\langle i, x[\alpha] \rangle$ has the type $\mathcal{M}\,\tau$.

The intended semantics of PRF(T) is described as follows (a more formal definition follows in Section 4.1):

- Tactics apply once their corresponding inference rule to *each* of the current goals. In particular, if the current index points to a subtree in the proof, the inferences are applied to all of the open goals in this subtree. The proof index remains unchanged, and the outcome of this application is recorded in the proof monad.
- The tactic `postpone` moves the index to the current goals to its brother. The symbol '.' is a stepwise instruction evaluator, and □ is an end-of-proof delimiter. These three elements will be referred to as *interactive commands*.
- The tactic `idtac` and the symbol ';' are the identity and sequential composition tactic combinators, respectively. The list combinator $[i_1, i_2, \ldots, i_n]$ applies the instruction i_k to the k-th current subgoal. It is assumed that the number of current subgoals is equal to n. Since these combinators are used to build complex proof scripts, they will be referred to as *programming strategies*.

In this formalism, proof instructions are considered to be functions from proofs to computations, i.e., objects of type $\tau \to \mathcal{M}\,\tau$. In this sense, the proposed treatment of proof languages is very similar to that of functional programming languages.

Definition 16 (PRF(T_{min})). The set of tactics in Definition 15 is instantiated with T_{min}, which forms the proof language PRF(T_{min}).

4.1. Semantics of Tactics

The formal semantics of tactics is derived from the semantics given in Figure 1.

Definition 17 (Formal semantics of tactics). The result of the evaluation of a tactic t on an indexed proof $x[\alpha]$ is derived from the semantics of its local application to a given sequent, as given in Figure 1, by the following algorithm.

- If there is only one current goal, the sequent inference rule is applied to the goal, with the necessary guards:
  - if the topmost symbol of the current goal does not match the symbol required for the application of the logical inference rule, then the result is the monadic construction $\mathsf{exception}\, s$, where s is a symbol that is associated with, say, the exception "`tactic cannot be applied to current subgoal`";
  - if no conditions are necessary for the application of the inference rule and y is the proof x with the previously current goal closed, then the result is the monadic construction $y[\mathsf{success}]$;

  - if $n > 0$ conditions need to be discharged for the application of the inference rule and y is the proof x extended with the n corresponding subgoals, then the result is the monadic construction subgoals `true` $y[\alpha]$;
  - finally, if a tactic applies but does not modify $x[\alpha]$, then the result is the monadic construction subgoals `false` $x[\alpha]$. This is used mainly to perform bookkeeping tasks, such as formula renaming, some subgoals reordering, etc.

- If there are m current goals. In this case, the tactic t is recursively evaluated on each of the current goals following the proof tree's open goals ordering. Furthermore,
  - if any of the evaluations generates an exception, then return the exception;
  - if the evaluation discharges all current goals, and z is the proof x with its previously current goals closed, then the result is the monadic construction z[success];
  - if, for some goals, subgoals are generated, and z is the proof x extended with the subgoals, then the result is the monadic construction subgoals `true` $z[\alpha]$;
  - if the tactic has not modified any of the current goals, then the result is the monadic construction subgoals `false` $x[\alpha]$.

Example 2. Let $x[\alpha]$ be a proof tree with only one current goal, representing the following formula:

$$\Gamma; \Xi_0 : A \Rightarrow B \vdash C$$

Without loss of generality, we can assume that α represents the position of this open goal in x. Let $z[\beta]$ be a proof tree where no such current goal exists. The semantics of $\Rightarrow_{\mathcal{L}}$ is given by

$$\langle \Rightarrow_{\mathcal{L}}, x[\alpha] \rangle \rightarrow \text{subgoals } \texttt{true } y[\alpha]$$
$$\langle \Rightarrow_{\mathcal{L}}, z[\beta] \rangle \rightarrow \text{exception s}$$

where:

- y is a proof tree exactly as x except in the position α, where $y|_\alpha$ represents the subtree of the following derivation:

$$\frac{\Gamma \vdash X_1 : A \qquad \Gamma; \Xi_1 : B \vdash C}{\Gamma; X_1 \cdot \Xi_1 : A \Rightarrow B \vdash C}$$

  Note that the index α remains unchanged by the tactic application.

- `s` is the symbol that uniquely represents the exception "`tactic cannot be applied to current subgoal`".

$$\langle \texttt{idtac}, x[\alpha]\rangle \longrightarrow \mathrm{unit}\, x[\alpha]$$
$$\langle c_1; c_2, x[\alpha]\rangle \longrightarrow \langle c_1, x[\alpha]\rangle \star c_2$$
$$\langle [c, l], x[\alpha]\rangle \longrightarrow \mathit{fold}_{\alpha,0}\ [c, l]\ \langle c, x[\mathsf{success}]\rangle$$

with

$$\begin{array}{l}
\mathit{fold}_{\alpha,i}\ [c]\ m \longrightarrow \mathrm{match}\ m\ \mathrm{with} \\
\quad \mid x[\mathsf{subgoals}\ b\ \beta] \mapsto \mathrm{match}\ \langle c, x[\downarrow_i \alpha]\rangle\ \mathrm{with} \\
\qquad\quad \mid y[\mathsf{subgoals}\ b'\ \gamma] \mapsto y[\mathsf{subgoals}\ (b \oplus b')\ \alpha] \\
\qquad\quad \mid y[\mathsf{success}] \mapsto y[\mathsf{subgoals}\ b\ \alpha] \\
\qquad\quad \mid \mathsf{exception}\ s \mapsto \mathsf{exception}\ s \\
\quad \mid x[\mathsf{success}] \mapsto \mathrm{match}\ \langle c, x[\downarrow_i \alpha]\rangle\ \mathrm{with} \\
\qquad\quad \mid y[\mathsf{subgoals}\ b\ \gamma] \mapsto y[\mathsf{subgoals}\ b\ \alpha] \\
\qquad\quad \mid * \mapsto m \\
\quad \mid \mathsf{exception}\ s \mapsto \mathsf{exception}\ s
\end{array}$$

and

$$\begin{array}{l}
\mathit{fold}_{\alpha,i}\ [c, l]\ m \longrightarrow \mathrm{match}\ m\ \mathrm{with} \\
\quad \mid x[\mathsf{subgoals}\ b\ \beta] \mapsto \mathrm{match}\ \langle c, x[\downarrow_i \alpha]\rangle\ \mathrm{with} \\
\qquad\quad \mid y[\mathsf{subgoals}\ b'\ \gamma] \mapsto \mathit{fold}_{\alpha,i+1}\ [l]\ y[\mathsf{subgoals}\ (b \oplus b')\ \gamma] \\
\qquad\quad \mid y[\mathsf{success}] \mapsto \mathit{fold}_{\alpha,i+1}\ [l]\ y[\mathsf{subgoals}\ b\ \beta] \\
\qquad\quad \mid \mathsf{exception}\ s \mapsto \mathsf{exception}\ s \\
\quad \mid x[\mathsf{success}] \mapsto \mathit{fold}_{\alpha,i+1}\ [l]\ \langle c, x[\downarrow_i \alpha]\rangle \\
\quad \mid \mathsf{exception}\ s \mapsto \mathsf{exception}\ s
\end{array}$$

Figure 2: Semantics of programming strategies

To conclude the example, these semantics can be extended to deal with any number of current goals: if there are k current goals, then either $\Rightarrow_{\mathcal{L}}$ applies to all of them, and subgoals `true` ... is generated. Otherwise, the exception constructor is used.

4.2. Semantics of Strategies

Definition 18 (Formal semantics of strategies). Figures 2 and 3 complete the definition of the semantics of programming strategies and interactive commands, respectively.

The monadic operators unit and $\star$ are exact denotations for the identity `idtac` and sequential composition ';' combinators. An intermediary function *fold* is used to inductively define the semantics of the list evaluation. The monadic operator *map* is also implicitly used to implement the tactic `postpone`.

$$
\begin{array}{rl}
\langle \texttt{postpone}, x[\alpha]\rangle \rightarrow & \mathsf{subgoals}\ \texttt{false}\ x[\rightsquigarrow \alpha] \\
\langle c_1.c_2, x[\alpha]\rangle \rightarrow & \text{match } \langle c_1, x[\alpha]\rangle \text{ with} \\
& \left|\begin{array}{l} y[\mathsf{subgoals}\ b\ \beta] \mapsto \langle c_2, y[\downarrow_1 \beta]\rangle \\ y[\mathsf{success}] \mapsto \langle c_2, y[\rightsquigarrow \alpha]\rangle \\ \mathsf{exception}\ s \mapsto \mathsf{exception}\ s \end{array}\right. \\
\langle \square, x[\alpha]\rangle \rightarrow & x[\mathsf{success}] \text{ if no open goals remain in } x, \\
& \mathsf{exception}\ \texttt{"the proof is not yet finished"} \text{ else.}
\end{array}
$$

Figure 3: Semantics of interactive commands

$$
\begin{array}{rl}
\langle \texttt{spread}\ l, x[\alpha]\rangle \rightarrow & \text{if length } l > \text{card } \alpha \\
& \text{then } \langle [\text{truncate (card } \alpha) \ l], x[\alpha]\rangle \\
& \text{else if length } l < \text{card } \alpha \\
& \text{then } \langle [\text{extend (card } \alpha) \ l], x[\alpha]\rangle \\
& \text{else } \langle [l], x[\alpha]\rangle \ ,
\end{array}
$$

where

- length l and card α return respectively the length of the list l and the number of open goals in α;
- truncate $n\,l$ and extend $n\,l$ respectively truncate the list l at length n and extend the list l to length n, using its last element as a filler.

Figure 4: A safe list strategy

The list operator does not handle the case where the number of current goals and the number of commands in the list are not equal. In implementations of this strategy where such a mismatch can occur, tests are added to detect and correct these cases. Figure 4 shows the semantics of such a strategy, called `spread`: if there are fewer goals than commands in the list, then the list is truncated; if there are more goals than commands then the last command is applied by default. This treatment of special cases can also be applied to define safe versions of the commands `postpone` and ‘;’.

The proof language presented in this section can easily be extended with exception constructors and destructors, e.g., `throw` and `catch` such as in the upcoming Example 3, progress testers, e.g., Coq’s `orelse`, and more complex strategies. Indeed, many of these language extensions are included in the im-

plementations of this concept discussed in Section 5.

Example 3. An exception mechanism that uses exception to raise, propagate and catch failures can be defined as follows

$$\begin{aligned}\langle \texttt{throw}\ s, x[\alpha]\rangle &\rightarrow \mathsf{exception}\ s\\ \langle \texttt{catch}\ i_1\ i_2, x[\alpha]\rangle &\rightarrow \text{match}\ \langle i_1, x[\alpha]\rangle\ \text{with}\\ &\quad \left|\begin{array}{l}\mathsf{exception}\ s \mapsto \langle i_2, x[\alpha]\rangle\\ m \mapsto m\end{array}\right.\end{aligned}$$

4.3. Characterization of Strategies

The formalism presented here can be used to mathematically characterize the behavior of different kinds of strategies. For instance, programming strategies and interactive commands can be characterized as follows.

Proposition 3 (Proof commands characterization).

1. *Let c be any proof command constituted only of tactics and programming strategies. If $\langle c, x[\alpha]\rangle = \mathsf{subgoals}\ b\ y[\beta]$, then $\alpha = \beta$. In other terms, the programming strategies do not modify the position of the current index.*
2. *Interactive commands are the only constructs of a proof language that can modify the position of the current index in the proof.*

Proof. Proposition 1 is proven by case analysis on strategies. Proposition 2 is a straightforward corollary of Proposition 1. Q.E.D.

Remark that, while interactive commands can modify the position of the current index, they will not necessarily result in such a change. For instance, as a consequence of the semantics of $\rightsquigarrow$, `postpone` will not affect the current index if there is only one open goal in the proof tree.

The characterization effort can be pushed one level of abstraction higher, and reflect on what constitutes a proof strategy language:

- There needs to be a data structure to mirror user interaction: given a formal language, a consequence relation and the associated proof tree representation, it can be argued that the proof monad $(\mathcal{M}, \texttt{idtac}, \star)$ provides such a structure.
- Some constructors and a destructor for the monadic datatype, e.g., `match`, are required to generate and analyse proof feedback.
- The *fold* instruction is required to provide a way to apply distinct instructions to different current goals.
- Finally, there needs to be a programming language λ to build complex strategies out of simple ones—using the monadic datatype destructor if necessary.

The inclusion of the *fold* operator into this list suggests another kind of presentation of the proof monad, one that complicates the underlying categorical foundations but simplifies the characterization of a proof strategy language. Indeed, consider the $n+1$-ary strategy $i_0; [i_1, \ldots, i_n]$ as an implementation of a monadic composition operator.

Conjecture 1. *There exists an extension of the theory of monads, where the operator $\star$ is n-ary on the right. The monad laws for this construction are the left and right unit:*

$$\begin{aligned} \texttt{idtac} \star (i, \ldots, i) &= i \\ i \star (\texttt{idtac}, \ldots, \texttt{idtac}) &= i \end{aligned}$$

and associativity:

$$i \star (i_0^1 \star (i_1^1, \ldots, i_{n_1}^1), \ldots, i_0^m \star (i_1^m, \ldots, i_{n_m}^m)) = i \star (i_0^1, \ldots, i_0^m) \star (i_1^1, \ldots, i_{n_m}^m)$$

Given such an extension, the triple $(\mathcal{M}, \texttt{idtac}, \star)$, where $\star$ is n-ary on the right, is a monad.

Assuming this conjecture holds, the previous list of features for proof languages can be reduced to an n-ary proof monad built over the proof structure, along with its datatype constructors and destructors, accompanied by a programming language of some sort.

5. Implementations

5.1. The Proof Monad in PVS

PVS has an extensive proof language [36]. Initial attempts to formalize its semantics [2], although incomplete, have highlighted its complexity. In particular, it uses two kinds of exceptions, called *failure* and *backtrack*. In [19], the authors have proposed denotational semantics for a large part of this language, based on a pseudo-monadic structure. The theory of the proof monad is a direct extension and simplification of this work, and the results obtained in [19] can be trivially translated into this article's formalism.

The formalization of the proof structure allowed for a better understanding and documentation of PVS's proof language. Based on this formalization, a number of new strategies testing the proof engine feedback have been written to aid the user in controlling the prover's power. For instance, a strategy named `test` was implemented that simulates the application of an instruction to the proof state without modifying it and, depending on the outcome, it applies one of its remaining arguments.

Furthermore, a streamlined proof language was designed based on the proof structure, with the aim of providing instructions with very elementary semantics to the user, and hence a leaner learning curve for new strategy programmers. This language, called PVS$^{\#}$, has been implemented in a PVS package called *Practicals*, and is available to download at:

`shemesh.larc.nasa.gov/people/cam/Practicals`

In PVS$^{\#}$, the *failure/backtrack* exception mechanism was abandoned in favor of a simple exception handling mechanism. Additionally, while PVS's original proof language was largely based on the complex `try` strategy, PVS$^{\#}$'s underlying structure is built on the simpler monadic operators.

5.2. The Proof Monad in FSP

FSP is a prover intended to be used as an interface to other provers, such as Coq and PVS. Its proof development component carries a thorough implementation of the proof monad as presented in this article as a means to encode the prover-user interaction. In particular, it features the strategy constructs proposed in this paper. These constructs have been used to write proof scripts, in particular in the development of a 150 proofs-long library encoding the theory of real closed fields.

FSP is still under construction, but releases are readily available at:

`www.lix.polytechnique.fr/Labo/Florent.Kirchner/fellowship/trunk`

An extensive documentation of the system and of its proof language can be found in [18].

6. Discussion and Related Work

From the perspective of strategies, the type of tactics in the monadic framework is $\tau \to \mathcal{M}\tau$. This means that a tactic takes a proof tree with potentially many current goals and returns another tree with additional information on the outcome of the application of the tactic. This global interpretation of tactics is lifted from the local one, that maps a current goal to a one-step proof inference with zero or more subgoals. Other avenues were also explored:

- Defining an operator $\star$ of the form:

  $$\mathit{goal\ multiset} \to (\mathit{goal} \to \mathit{goal\ multiset}) \to \mathit{goal\ multiset}$$

  with monadic information attached to multisets. This approach ignores the global semantics of strategies. For instance, strategies that require knowledge of the complete proof tree (such as proof plans) cannot be expressed using this approach. While the semantics of proof plans are beyond the scope of this paper, it seems possible to formalize proof plans using proof monads.

  What is more, the use of goal multisets entails the loss of the structure of the proof tree, an arguably significant asset when reasoning about the semantics of a proof script. This is aggravated by the complexity of the datatypes involved in the treatment of metavariables (see Section 6.1).

- Defining an operator $\star$ of the form:

$$\mathcal{M}\tau \to (\tau_{[1]} \to \mathcal{M}\tau) \to \mathcal{M}\tau$$

  where $\tau_{[1]}$ is a proof tree with just one current goal. In this case the expression of the operator $\star$ is convoluted, having to include a way to loop through the subgoals of its first argument. Instead, the approach presented in this paper trades a more liberal notion of tactic application (one that takes effect on a current tree rather than a single goal) for a streamlined composition operation.

The semantics of tactic application presented in this paper seems to be restrictive. Indeed, it is assumed that applying a tactic to a set of current goals corresponds to iterating the application of the tactic to all goals in order. This assumption on the semantics of tactics introduces some of the traditional limitations of LCF tactics as, for example, a view of the proof tree that is limited to the current goals. Some of these limitations can be overcome with the use of programming strategies, rather than tactics. However, this may not always be possible given both performance requirements, and the overhead involved with such a programming paradigm.

6.1. The case of existential variables

There exists an advanced feature of procedural theorem provers that complicates the semantics of tactics: *existential variables*. In a number of theorem provers, existential variables are used to represent incomplete formulas, and facilitate proof development by deferring some of the guessing work to a later point. As such, they can be shared between different goals, and they can be instantiated as a side-effect of some proof instructions. While this has no impact on strategies, it requires tactics to be considered as acting on a more global structure, instead of local goals. It is possible to deal with this extension in a modular way, for example by dealing with the goals locally and widening the scope as needed, but this significantly complicates the semantics of tactics.

Furthermore, existential variables are treated differently by different theorem provers. In Coq, they are considered separate from the proof tree, and can only be manipulated by specific tactics. This isolates the problem to a few corner cases, that can be dealt with, for instance, by adding a map of these variables to the proof tree structure. In Matita [3], however, they are considered an integral part of the proof tree, and goals can be discharged as a common tactic side-effect. Proof monads can be extended to cover this use by promoting an approach similar to Arnaud Spiwack's, who is currently working on a way to represent dependencies between subgoals in Coq [37]. In this approach, all the goals that contain an existential variable can cross-reference each other, and the definition of a current goal would comes to encompass all its dependent subgoals. A tactic that needs this piece of information can then access the cross-referenced goals.

6.2. Related work

Delahaye [10] made the first attempt at formalizing the semantics of Coq's proof language. He proposed a language called $\mathcal{L}_{pdt}$ and provided its formal big-step semantics. However, these semantics were coupled with the semantics of lower-level tactics, making it difficult to abstract the principles of his design from his implementation of a logical framework. Using the constructs inherited from LCF [13], Delahaye also enriched Coq's proof language with a few powerful programming constructs, and documented this work using informal big-step semantics.

Jojgov [15] used a notion of parametrized metavariables to describe unproved branches of incomplete proofs in the CIC logical framework. He proposed small-step operational semantics for proof languages, but he did not recognize the modularity of the formalisms for tactics and strategies. As a result, the rules in his framework deal with whole proofs. The case of the sequence and identity strategies are only mentioned as an aside and without any reference to a monadic structure, and the semantics are largely focussed on the case of tactics.

Sacerdoti, Tassi, and Zacchiroli [8] recently formalized and implemented an innovative step-by-step evaluator for some of the strategies in the LCF proof language. In doing so, they expressed the small-step semantics of their language and discussed the human-prover interaction. In their work, they treated proofs as collections of lists (context, continuation, open goals, etc.) and they did not make the connection with monadic structures.

Martin and Gibbons [27] in an unpublished note remarked that Angel's [26] proof language had a monadic structure, and generalized their observations to a generic proof language. Their note is based on the same intuition as the one presented in this paper. However their development avoided any description of a proof datatype, only mapping proof language constructs to monadic operators. Furthermore, their proof language was minimal and the extension to some of the tactic combinators presented in this paper is not straightforward.

An interesting comparison can be made with frameworks for combinator parsing [39] that, like the work presented here, propose a sequencing operator. However, unlike proof languages, those frameworks provide support for alternation, i.e., parallel application. The closest feature to alternation in LCF-style theorem provers is the strategy `orelse`, which does not provide the aggregation feature of alternation. Instead, this strategy applies its arguments sequentially until one succeeds. This feature could be used in dealing with proof search strategies, a key features of automated theorem provers, by helping model the construction of the search space.

Rewriting strategies [7] use state information similar to the monadic structure presented in this paper. Moreover, the structure of proof trees is also similar to the structure of *terms* used by rewrite systems. However, rewriting strategy languages do not have interactive features as proof languages of procedural theorem provers do. In this area, the link with rewriting strategies has recently been investigated [17] and the relation to deduction modulo [11] is being examined.

7. Conclusion

This article has presented a mathematical structure designed to express the complex semantics of procedural proof languages, and in particular the human-prover interaction of these languages. It has been shown that a monadic construct, combined with a careful abstraction of the concepts of proof trees and current goals, is expressive enough for this purpose. In this sense this result is in line with those obtained in the theory of programming languages, where the same kind of monadic constructions have been used to add imperative traits to functional languages.

An important feature of this formalism is that it relies on a representation of proof trees that is not tied to any particular logical framework: this ensures that it can be used to formalize all kinds of different proof systems. This feature has been illustrated by using the proof monad to express the semantics of the proof languages in two different proof assistants: PVS, which is based on typed higher-order logic, and FSP, which is based on an untyped first-order logic.

The main contribution of this work is in the field of proof languages design. The proof monad sets a formal framework for this task, where previously none existed. It also helps identify the core constructs of these languages, and suggests a dichotomy between their programming and interactive components.

Acknowledgements

The authors would like to thank François-Régis Sinot for the discussions on monads, and Albert Buronni for his excellent course on category theory at Université de Paris VII. Many thanks to the Formal Methods Group at NASA Langley and SRI International for their early review and comments on this work, and especially to Natarajan Shankar for providing the semantics of PVS's `try` strategy. The anonymous referees have significantly contributed to the final version of this paper, through their precise reading and discerning criticism.

References

[1] J. Adámek, H. Herrlich, G. Strecker, Abstract and Concrete Categories, Pure and Applied Mathematics, John Wiley & Sons, New York, NY, 1990.

[2] M. Archer, B.D. Vito, C. Muñoz, Developing user strategies in PVS: A tutorial, in: Proc. Int. Workshop on Design and Application of Strategies/Tactics in Higher Order Logics, number CP-2003-212448 in NASA Conference Publication, pp. 16–42.

[3] A. Asperti, C.S. Coen, E. Tassi, S. Zacchiroli, Crafting a proof assistant, in: T. Altenkirch, C. McBride (Eds.), Proc. 2006 Int. Workshop on Proofs and Programs, volume 4502 of *LNCS*, Springer, Heidelberg, 2007, pp. 18–32.

[4] A. Avron, Simple consequence relations, Information and Computation 92 (1991) 105–139.

[5] B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, B. Werner, The Coq Proof Assistant Reference Manual, 2006.

[6] A. Buronni, Théorie des catégories, 2004. Informal lectures.

[7] H. Cirstea, C. Kirchner, L. Liquori, Rewrite strategies in the rewriting calculus, in: B. Gramlich, S. Lucas (Eds.), Proc. 3rd Int. Workshop on Reduction Strategies in Rewriting and Programming, volume 86 of *Electronic Notes in Theoretical Computer Science*, Elsevier, Amsterdam, 2003.

[8] C.S. Coen, E. Tassi, S. Zacchiroli, Tinycals: Step by step tacticals, in: Proc. 7th Workshop on User Interfaces for Theorem Provers, volume 174 of *Electronic Notes in Theoretical Computer Science*, Elsevier, Amsterdam, 2007, pp. 125–142.

[9] P.L. Curien, H. Herbelin, The duality of computation, ACM sigplan notices 35 (2000).

[10] D. Delahaye, A tactic language for the system Coq, in: M. Parigot, A. Voronkov (Eds.), Proc. 7th Int. Conf. on Logic for Programming and Automated Reasoning, volume 1955 of *LNCS*, Springer, Heidelberg, 2000, pp. 85–95.

[11] G. Dowek, T. Hardin, C. Kirchner, Theorem proving modulo, Journal of Automated Reasoning 31 (2003) 33–72.

[12] M. Gordon, T. Melham (Eds.), Introduction to HOL: A theorem proving environment for higher order logic, Cambridge Univ. Press, 1993.

[13] M. Gordon, R. Milner, C. Wadsworth, Edinburgh LCF: A Mechanized Logic of Computation, volume 78 of *LNCS*, Springer, Heidelberg, 1979.

[14] B. Jacobs, Categorical Logic and Type Theory, volume 141 of *Studies in Logic and the Foundations of Mathematics*, Elsevier, Amsterdam, 1998.

[15] G. Jojgov, Incomplete proofs and terms and their use in interactive theorem proving, Ph.D. thesis, Technische Universiteit Eindhoven, 2004.

[16] M. Kaufmann, J.S. Moore, ACL2: An industrial strength version of Nqthm, in: Compass'96: Eleventh Annual Conference on Computer Assurance, National Institute of Standards and Technology, Gaithersburg, Maryland, 1996, p. 23.

[17] C. Kirchner, F. Kirchner, H. Kirchner, Strategic computations and deductions, in: Reasoning in Simple Type Theory, volume 17 of *Mathematical Logic and Foundations*, College Publications, 2008.

[18] F. Kirchner, Interoperable proof systems, Ph.D. thesis, École Polytechnique, 2007.

[19] F. Kirchner, C. Muñoz, PVS#: Streamlined tacticals for PVS, in: Workshop on Strategies in Automated Deduction, volume 174/11 of *Electronic Notes in Theoretical Computer Science*, Elsevier, Amsterdam, 2007, pp. 47–58.

[20] F. Kirchner, C. Sacerdoti Coen, The Fellowship proof manager, 2007.

[21] J. Lambek, Deductive systems and categories I: Syntactic calculus and residuated categories, Math. Systems Theory 2 (1968) 287–318.

[22] J. Lambek, P. Scott, Introduction to Higher Order Categorical Logic, volume 7 of *Cambridge Studies in Advanced Mathematics*, Cambridge Univ. Press, Cambridge, 1986.

[23] S.M. Lane, Categories for the Working Mathematician, volume 5 of *Graduate Texts in Mathematics*, Springer, New York, NY, 1971.

[24] F.W. Lawvere, Adjointness in foundations, Dialectica 23 (1969) 281–296.

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

[26] A. Martin, P. Gardiner, J. Woodcock, A tactic calculus, in: J. Allison (Ed.), Formal Aspects of Computing.

[27] A. Martin, J. Gibbons, A monadic interpretation of tactics, 2002.

[28] C. McBride, Dependently Typed Functional Programs and their Proofs, Ph.D. thesis, University of Edinburgh, 1999.

[29] E. Moggi, Computational lambda-calculus and monads, in: Proc. 4th IEEE Symp. Logic in Computer Science, IEEE Comp. Soc. Press, 1989. Superseded by [30].

[30] E. Moggi, Notions of computation and monads, Information and Computation 93 (1991) 55–92.

[31] C. Muñoz, Un calcul de substitutions pour la représentation de preuves partielles en théorie de types, Thèse de doctorat, Université Paris 7, 1997. English version available as INRIA research report RR-3309.

[32] S. Owre, J. Rushby, N. Shankar, PVS: A prototype verification system, in: D. Kapur (Ed.), Proc. 11th Int. Conf. on Automated Deduction, volume 607 of *Lecture Notes in Artificial Intelligence*, Springer, 1992, pp. 748–752.

[33] R. Seely, Hyperdoctrines, natural deduction and the Beck condition, Zeitschrift für mathematische Logik und Grundlagen der Mathematik 29 (1983) 505–542.

[34] R. Seely, Locally cartesian closed categories and type theory, in: Mathematical Proceedings of the Cambridge Philosophical Society, volume 95, Cambridge Philosophical Society, pp. 33–48.

[35] R. Seely, Linear logic, $*$-autonomous categories and cofree coalgebras, in: J. Gray, A. Scedrov (Eds.), Categories in Computer Science and Logic, volume 92 of *Contemporary Mathematics*, American Mathematical Society, Providence, Rhode Island, 1989, pp. 371–382.

[36] N. Shankar, S. Owre, J. Rushby, D. Stringer-Calvert, PVS Prover Guide, Computer Science Laboratory, SRI International, 1999.

[37] A. Spiwack, [Coq-Club] Instantiating existential variables, Email conversation, 2009.

[38] T. Streicher, Semantics of Type Theory, volume 5 of *Progress in Theoretical Computer Science*, Birkhäuser, 1991.

[39] P. Wadler, Monads for functional programming, in: J. Jeuring, E. Meijer (Eds.), Advanced Functional Programming, volume 925 of *LNCS*, Springer, Heidelberg, 1995.