Free-style Theorem Proving*

David Delahaye**

Programming Logic Group
Chalmers University of Technology***

Abstract. We propose a new proof language based on well-known existing styles such as procedural and declarative styles but also using terms as proofs, a specific feature of theorem provers based on the Curry-Howard isomorphism. We show that these three styles are really appropriate for specific domains and how it can be worth combining them to benefit from their advantages in every kind of proof. Thus, we present, in the context of the Coq proof system, a language, called $\mathcal{L}_{pdt}$, which is intended to make a fusion between these three styles and which allows the user to be much more free in the way of building his/her proofs. We provide also a formal semantics of $\mathcal{L}_{pdt}$ for the Calculus of Inductive Constructions, as well as an implementation with a prototype for Coq, which can already run some relevant examples.

1 Introduction

In theorem provers, we can generally distinguish between two kinds of languages: a proof[1] language, which corresponds to basic or more elaborated primitives and a tactic language, which allows the user to write his/her own proof schemes. In this paper, we focus only on the first kind of language and we do not deal with the power of automation of proof systems. Here, we are interested in expressing proofs by considering different kinds of styles, finding the "good" criteria and thinking about combining these criteria according to the proof.

Proof languages for theorem provers have been extensively studied and, currently, two styles have gained prominence: the procedural style and the declarative style. John Harrison makes a complete comparison between these two styles in [5], where the declarative style is rather enhanced but also moderated. Indeed, the author shows how it is difficult to make a clear division between these two styles and this can be seen in the Mizar mode [6] he developed for HOL. In the same way, Markus Wenzel has implemented a generic declarative layer, called Isar [11], on top of Isabelle. From a more declarative-dedicated viewpoint, there

* This work has been realized within the LogiCal project (INRIA-Rocquencourt, France).

** delahaye@cs.chalmers.se, http://www.cs.chalmers.se/~delahaye/.

*** Chalmers University of Technology, Department of Computing Science, S-412 96 Gothenburg, Sweden.

1 The word "proof" may be open to several interpretations. Here, we mostly use "proof" in the sense of a script to be presented to a machine for checking.

are the work of Don Syme with his theorem prover Declare [8] and Vincent Zammit's thesis [12]. Closer to natural language, there is also the Natural package of Yann Coscoy [2] in Coq, where it is possible to produce *natural* proofs from proof terms and to write *natural* proof scripts directly. Along the same lines, in Alfa [1] (successor of ALF), we can produce *natural* proofs in several languages.

A third kind of proof language, quite specific to the logic used by the proof system, could be called "term language". This is a language which uses the coding of proofs into terms and propositions into types. This semantic (Brouwer-Heyting-Kolmogorov) deals only with intuitionistic logics (without excluded middle) and significantly modifies the way proofs are checked. In such a context, the logic is seen as a type system and checking if a proof (term) corresponds to a proposition (type) is only typechecking (Curry-Howard's isomorphism). The first versions of Coq used that kind of language before having a procedural style even if it is always possible to give terms as proofs. In Lego or Nuprl, we have exactly the same situation. Currently, only Alfa uses a direct manipulation of terms to build proofs.

The main idea in this paper is to establish what the contributions of these three kinds of languages are, especially in which frameworks, and to make a fusion which could be interesting in practice. Before presenting our proposition, we consider a short example of a proof to allow the reader to have some idea of these three proving styles. It also illustrates the various features of these languages and shows the domains (in terms of the kinds of proofs) in which they are suitable. Next, we give the syntax of the language we suggest in the context of Coq [9], and which has been called $\mathcal{L}_{pdt}$ ("pdt" are initials denoting the fusion between the three worlds: "p" is for procedural, "d" is for declarative and "t" is for term), as well as some ideas regarding the formal semantics that has been designed. This last point is also a major originality of this work and as far as the author is aware, ALF is the only system, which has a formally described proof language [7]. However, this semantics deals only with proof terms and, here, we go further trying to give also, in the same context, a formal semantics to procedural parts, as well as to declarative features. From this semantics, a prototype has been carried out and, finally, we consider some examples of use, which show how $\mathcal{L}_{pdt}$ can be appropriate, but also what kind of improvements can be expected.

2 Proof examples

The example we chose to test the three kinds of languages is to show the decidability[2] of equality on the natural numbers. It can be expressed as follows:

$$\forall n, m \in \mathbb{N}. n = m \vee \neg(n = m)$$

[2] This proposition is called "decidability of equality" because we will give an intuitionistic proof in every case. Thus, we can always *realize* the proof of this lemma toward a program which, given two natural numbers, answers "yes" if the two numbers are equal and "no" otherwise.

Informally, the previous proposition can be shown in the following way:

1. We make an induction on n.
   The basis case gives us $0 = m \vee \neg(0 = m)$ to be shown:
   (a) We reason by case on m.
   First, we must show that $0 = 0 \vee \neg(0 = 0)$, which is trivial because we know that $0 = 0$.
   (b) Next, we must show that $0 = m + 1 \vee \neg(0 = m + 1)$, which is trivial because we know that $\neg(0 = m + 1)$.
2. For the inductive case, we suppose that we have $n = m \vee \neg(n = m)$ (H) and we must show that $n + 1 = m \vee \neg(n + 1 = m)$:
   (a) We reason by case on m.
   First, we must show that $n+1 = 0 \vee \neg(n+1 = 0)$, which is trivial because we know that $\neg(n + 1 = 0)$.
   (b) Next, we must show that $n + 1 = m + 1 \vee \neg(n + 1 = m + 1)$. Thus, we reason by case with respect to the inductive hypothesis H:
      i. Either $n = m$: we can conclude because we can deduce that $n+1 = m+1$.
      ii. Or $\neg(n = m)$: we can conclude in the same way because we can deduce that $\neg(n + 1 = m + 1)$.

2.1 Procedural proof

As a procedural style, we decided to choose the Coq proof system [9]. A possible proof script for the previous proposition is the following:

```
Lemma eq_nat:(n,m:nat)n=m\/~n=m.
Proof.
  Induction n.
  Intro;Case m;[Left;Auto|Right;Auto].
  Intros;Case m;[Right;Auto|Intros].
  Case (H n1);[Intro;Left;Auto|Intro;Right;Auto].
Save.
```

Briefly, we can describe the instructions involved: `Intro`[s] introduces products in the local context (hypotheses), `Induction` applies induction schemes, `Case` makes reasoning by case, `Left`/`Right` applies the first/second constructor of an inductive type with two constructors and `Auto` is a resolution tactic which mainly tries to apply lemmas of a data base.

2.2 Declarative style

For the declarative proof, we used Mizar [10], which seems to be very representative of this proving style[3]. The proof is partially described in the following script (the complete proof is a little too long, so we have only detailed the basis case of the first induction):

[3] Indeed, the Mizar project developed a very impressive library essentially in mathematics. So, we can easily consider the stability of this language with respect to its very widespread use.

```
theorem
  eq_nat: n = m or not (n = m)
  proof
    A1: for m holds 0 = m or not (0 = m)
    proof
      A2: 0 = 0 or not (0 = 0)
      proof
        A3: 0 = 0;
        hence thesis by A3;
      end;
      A4: (0 = m0 or not (0 = m0)) implies
          (0 = m0 + 1 or not (0 = m0 + 1))
      proof
        assume 0 = m0 or not (0 = m0);
        A5: not (0 = m0 + 1) by NAT_1:21;
        hence thesis by A5;
      end;
      for k holds 0 = k or not (0 = k) from Ind(A2,A4);
      hence thesis;
    end;
    A6: (for m holds (n0 = m or not (n0 = m))) implies
        (for m holds ((n0 + 1) = m or not ((n0 + 1) = m)))
    ...
```

where `by`/`from` is used to perform a more or less powerful resolution, `hence thesis`, to conclude the current proof (between `proof` and `end`) and `assume`, to suppose propositions.

2.3 Term refinement

The third solution to solve the previous proposition is to give the proof term more or less directly. This can be done in a very easy and practical way by the prover Alfa [1], which is a huge graphic interface to Agda (successor of ALF). Figure 1 shows a part of the proof term which has been built gradually (filling placeholders). As for the Mizar proof, the complete proof is a bit too large, so, in the same way, we have only printed the basis case of the first induction. The rest is behind the "..." managed by Alfa itself, which allows parts of proofs to be hidden. Even if the explicit pretty-print allows well-known constructions (λ-abstractions, universal quantifiers, ...) to be recognized, some comments can be made to understand this proof. The universal quantifier $\forall$ has a special coding and to be introduced/eliminated, specific constants $\forall$`I`/$\forall$`E` are to be used. $\vee$`I1`/$\vee$`I2` allow the side of a $\vee$-proposition to be chosen. Finally, *natrec* is the recursor (proof term) of the induction scheme on natural numbers, *eqZero* is the proof that $0 == 0$ and *nat_ discr1* is the lemma $\forall a \in Nat. \neg(0 == a + 1)$.

2.4 Comparison

The procedural script in Coq shows that it is quite difficult to read procedural proofs. The user quickly becomes lost and does not know what the state of the

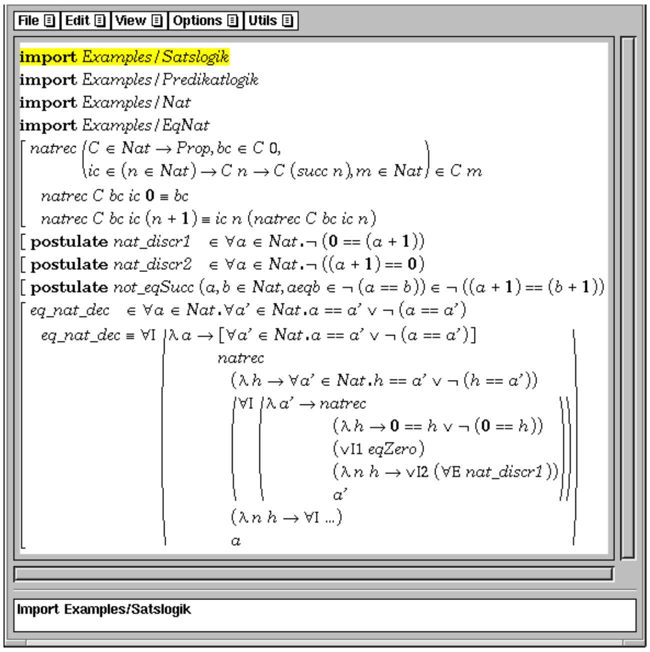

Fig. 1. Proof of the decidability of equality on natural numbers in Alfa.

proof is. Without the processing of Coq, the user cannot understand the proof and this is enforced by, for example, the apparition of free variables (as `H` or `n1` here) which are only bound during the checking of the proof. So, that kind of script cannot be written in a batch way but only and very easily in interaction with Coq. Thus, procedural proofs are rather backward-oriented because there is not much information provided by user's instructions and everything depends on the current goal to be proved. As concerns maintainability, obviously, we can imagine that these scripts are quite sensitive to system changes (naming conventions for example).

The Mizar script shows a very readable and so understandable proof. The declarations of auxiliary states allows the user to know what is proved and how to conclude. The proof can be written in batch mode and an interactive loop is not required at all to build this kind of proof. Indeed, the state of the proof may be *far* from the initial goal to be proved and we know how to prove it only at the end when we know how to make the link between this initial goal and the auxiliary lemmas we have proved to succeed, so that, declarative scripts are rather forward-oriented. A direct inconvenience of this good readability is that it is rather verbose and the user may well find it tedious to repeat many propositions (even with copy-paste). These many (forced) repetitions make declarative scripts fragile with respect to specification changes (permutations of arguments in a definition, for example) and we cannot imagine specific edition tools which could deal with every kind of change.

The proof in Alfa is rather readable. Of course, it requires a little practice to read terms as proofs but it is quite full of information especially when you reveal type annotations. In the same way, it is easy to write such scripts[4], both in an interactive proof loop as well as in batch mode, because, when building terms, we can ignore type annotations, which are filled by the proof system with respect to the goal. Thus, that kind of style is completely backward-oriented. Regarding robustness, system changes does not interfere because we do not give instructions to build a proof term but a proof term directly. But, as declarative scripts, specification changes may involve many changes in the term even without type signatures.

Following on the previous observations, we can easily tell where these styles are useful and also when it is interesting to use them. We will use procedural proofs for small proofs, known to be trivial and realized interactively in a backward mode, for which, we are not interested in the formal details. These proofs must be seen as black boxes. Declarative style will be used for more complex proofs we want to build more in a forward style (as in a natural proof in mathematics), rather in a batch way and very precisely[5], i.e. with much information for the reader. Finally, proof terms will be also for complex proofs, but backward-oriented, built either interactively or in batch mode (this is adapted for both methods), and for which we can choose the level of detail (put all type signatures, some of them or none). Thus, we can notice that the three styles correspond to specific needs and it could be a good idea to amalgamate them to benefit from their advantages in every kind of proof.

3 Presentation of $\mathcal{L}_{pdt}$

As said previously, $\mathcal{L}_{pdt}$ was developed in the context of the Coq proof system and there are several reasons for that choice. First, in this theorem prover, the procedural part comes for free because the proof language is purely procedural. But, the main reason is certainly that Coq uses the Curry-Howard isomorphism to code proofs into terms and the adaptation to make Coq accept terms as proofs will be rather easy[6]. Only declarative features are not quite natural for insertion in Coq but we will see how they can be simulated on a procedural proof

[4] We do not consider here the useful graphical interface which allows placeholders to be filled very easily (especially for beginners).

[5] However, this depends on the level of automation (that we do not want to consider here). Too great, an automation can strictly break readability even if we ask decision procedures to give details of the proofs they built because automated proofs are not really minimal and thus as clear as "human"-built proofs.

[6] We can already give complete terms (`Exact` tactic) or partial terms (`[E]Apply`, `Refine` tactics) to prove lemmas, but no means is provided to build these terms gradually or to have a total control on the placeholders we want to fill, so most of these tactics are only used internally.

machine. Although the choice for suitable systems might seem rather restricted (procedural systems based on Curry-Howard's isomorphism), such an extension could be also possible and quite appropriate for systems like Lego or Nuprl.

3.1 Definition

The syntax of $\mathcal{L}_{pdt}$ is shown in figure 2, presented in a BNF-like style and where <*proof*> is the start entry. <*ident*> and <*int*> are respectively the entries for identifiers and integers. <*tac*> includes all the current tactic language of Coq. Here, we have deliberately simplified the <*term*> entry. In particular, the `Cases` construction has been reduced compared to the real version and the `Fix`/`CoFix` forms has been removed (we will not use them directly in our examples and we will consider recursors just as constants in the way of Martin-Löf's type theory).

3.2 Semantics

As said previously, one of the major novelties of this work is to try to give a formal semantics to a proof language, here $\mathcal{L}_{pdt}$, not only for proof terms as for ALF in [7], but also for procedural and declarative parts. We chose to build a natural semantics (big steps), which is very concrete and so very close to a possible implementation. This semantics is correct in the context of the Calculus of Inductive Constructions (CIC) but without universes and to deal with the complete theory with universes, we only have to verify the universes constraints at the end with a specific procedure we did not formalize (this allows us to skip universe constraints from the evaluation rules and to keep *light* rules which are easier to read). Thus, in this context, to give a value to a proof script will consist in giving a term whose type is convertible to the lemma to be proved.

Preliminaries In figure 2, we call *proof script* or simply *script*, every expression of the <*proof*> entry. A *sentence* is an expression of the <*proof-sen*> entry. We call *term*, every expression of the <*term*> entry and *pure term*, every expression of the <*term*> entry, the <*proc*> and <*decl*> entries excluded. We call *procedural part*, every expression of the <*proc*> entry and declarative part, every expression of the <*decl*> entry. Finally, a tactic is an expression of the <*tac*> entry.

A *local environment* or *context* is an ordered list of hypotheses. An hypothesis is of the form $(x : T)$, where x is an identifier and T, a pure term called type of x. A *global environment* is an ordered list of hypotheses and definitions. A definition may be inductive or not. A non-inductive definition is of the form $c := t : T$, where c is an identifier called constant, t, a pure term called body of c and T, a pure term called type of c or t. An inductive definition is of the form $\mathsf{Ind}[\Gamma_p](\Gamma_d := \Gamma_c)$, where Γ_p, Γ_d and Γ_c are contexts corresponding to the parameters, the defined inductive types and the constructors. A *goal* is made of a global environment Δ, a local environment Γ and a pure term T (type).

```
<proof>        ::= (<proof-sen>)+

<proof-sen>    ::= <proof-top>.

<proof-top>    ::= Let [[?<int>]] <ident> : <term>
                |  Let <let-clauses>
                |  ?<int> [: <term>]:= <term>
                |  Proof
                |  Qed | Save

<let-clauses>  ::= [[?<int>]] <ident> [: <term>] := <term>
                   (And [[?<int>]] <ident> [: <term>] := <term>)*

<term>         ::= [<binders>]<term>
                |  <term> -> <term>
                |  (<ident>(, <ident>)* : <term>)<term>
                |  ((<term>)*)
                |  [<<term>>] Cases <term> of (<rules>)* end
                |  Set | Prop | Type
                |  <ident>
                |  ? | ?<int>
                |  <proc>
                |  <decl>

<binders>      ::= <ident> (,<ident>)* [: <term>](; <binders>)*

<rules>        ::= [|] <pattern> => <term> (| <pattern> => <term>)*

<pattern>      ::= <ident> | ((<ident>)+)

<proc>         ::= <by <tac>>

<tac>          ::= Intro | Induction <ident> | ...

<decl>         ::= Let <let-clauses> In <term>
```

Fig. 2. Syntax of $\mathcal{L}_{pdt}$.

```
<term-eval> ::= [<binders>]<term-eval>
             |  <term-eval> -> <term-eval>
             |  (<ident>(, <ident>)* : <term-eval>)<term-eval>
             |  ((<term-eval>)*)
             |  [<<term-eval>>] Cases <term-eval> of (<rules>)* end
             |  Set | Prop | Type
             |  <ident>
             |  ?i<int> | ?<int>
```

Fig. 3. Evaluated terms.

The *evaluated terms* are defined by the <*term-eval*> entry of figure 3. The set of evaluated terms is noted $\mathcal{T}_{\mathcal{E}}$. We call *implicit argument*, every term of the form `?`, or every evaluated term of the form $\texttt{?}_{\texttt{i}n}$, where n is an integer. The set of evaluated terms which are implicit arguments is noted $\mathcal{I}_{\mathcal{E}}$. We call *metavariable*, every term or every evaluated term of the form $\texttt{?}n$, where n is an integer. Finally, every term or every evaluated term of the form `Set`, `Prop` or `Type` is a *sort* and the sort set is noted S.

Term semantics This consists in typechecking incomplete[7] terms of CIC (pure terms) and interpreting some other parts (procedural or declarative parts) which give indications regarding the way the corresponding pure term has to be built. Procedural and declarative parts which are in the context of a term are replaced by metavariables and instantiations (of those metavariables) where they occur at the root.

The values are:

- either $(t, \Delta[\Gamma \Vdash T], m, \sigma)$, where $t \in \mathcal{T}_{\mathcal{E}}$, $\Delta[\Gamma \Vdash T]$ is a goal, m is a set of couples metavariable numbers-goals $(i, \Delta[\Gamma_i \Vdash T_i])$, σ is a substitution from $\mathcal{I}_{\mathcal{E}}$ to $\mathcal{T}_{\mathcal{E}}$, s.t. $\nexists j$, $?j \in \Delta[\Gamma \Vdash T]$ and $?j \in \Delta[\Gamma_i \Vdash T_i]$,
- or Error.

The set of those values is noted $\mathcal{V}_{\mathcal{T}}$.

Thus, a value is given by a proof term t, the type of this proof term $\Delta[\Gamma \Vdash T]$, a set of metavariables (bound to their types) m occurring in t and an instantiation of some implicit arguments (coming from the typechecking of t) σ. Moreover, there are two restrictions: there is no metavariable in the type of t ($\Delta[\Gamma \Vdash T]$), as well as in the types of the metavariables (in m).

Two modes of evaluation are possible if we have the type of the term or not: the verification mode and the inference mode. Initially, we have the type of the term (coming from the goal) and we use the verification mode, but some terms (like applications) can only be evaluated in the inference mode.

[7] Indeed, pure terms may contain some metavariables.

Due to lack of space here, we will not be able to give all the evaluation rules (with the corresponding error rules) of the several semantics. The reader can refer to [4][8] for a complete description (figures 5.3–5.16, pages 57–73) and we will just give some examples of rules. In particular, we will not consider the error rules, which can be also found in [4] (appendix A).

Pure terms A pure term t is evaluated, in the inference mode (resp. in the verification mode), into v, with $v \in \mathcal{V}_{\mathcal{T}}$, in the undefined goal $\Delta[\Gamma]$ (resp. in the goal $\Delta[\Gamma \Vdash T]$), iff $(t, \Delta[\Gamma]) \triangleright v$ (resp. $(t, \Delta[\Gamma \Vdash T]) \triangleright v$) holds.

As an example of rule (to derive the relation $\triangleright$), the evaluation of a λ-abstraction *à la Curry* in an undefined goal is the following:

$$\frac{\begin{array}{c}(t, \Delta[\Gamma, (x :?_{in})]) \triangleright (t_1, \Delta[\Gamma_1, (x : T_1) \Vdash T_2], m, \sigma) \\ x \notin \Delta[\Gamma] \qquad \mathsf{new\_impl}(n)\end{array}}{([x :?]t, \Delta[\Gamma]) \triangleright ([x : T_1]t_1, \Delta[\Gamma_1 \Vdash (x : T_1)T_2], m, \sigma)} \; (\lambda_{\mathsf{UCurry}})$$

where $\mathsf{new\_impl}(n)$ ensures that the integer n numbering an implicit argument is unique.

This rule consists in giving a number (n) to the implicit argument corresponding to the type of x, verifying that x is not already in the (global and local) environments and typechecking t.

Declarative parts We extend the relation $\triangleright$ to deal with the declarative parts: a pure term or a declarative part t is evaluated, in the inference mode (resp. in the verification mode), into v, with $v \in \mathcal{V}_{\mathcal{T}}$, in the undefined goal $\Delta[\Gamma]$ (resp. in the goal $\Delta[\Gamma \Vdash T]$), iff $(t, \Delta[\Gamma]) \triangleright v$ (resp. $(t, \Delta[\Gamma \Vdash T]) \triangleright v$) holds.

For instance, a `Let ... In` in an undefined goal is evaluated as follows:

$$\frac{\begin{array}{c}(T_1, \Delta[\Gamma]) \triangleright (T_2, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1) \qquad s \in S \\ (t_1, \Delta[\Gamma_1 \Vdash T_2]) \triangleright (t_3, \Vdash [\Gamma_2 \Vdash T_3], m_2, \sigma_2) \\ (t_2, \Delta[\Gamma_2, (x : T_3)]) \triangleright (t_4, \Delta[\Gamma_3, (x : T_4) \Vdash T_5], m_3, \sigma_3)\end{array}}{\begin{array}{c}(Let\ x : T_1 := t_1\ In\ t_2, \Delta[\Gamma]) \triangleright \\ (([x : T_4]t_4\ t_3\sigma_3), \Delta[\Gamma_3 \Vdash T_5[x \backslash t_3\sigma_3]], (m_1\sigma_2 \cup m_2)\sigma_3 \cup m_3, \sigma_1\sigma_2\sigma_3)\end{array}} \; (\mathsf{ULetIn})$$

In this rule, first, the type of the binding (T_1) is typechecked, next, the body of the binding (t_1) is typechecked with the previous type where some instantiations (of implicit arguments) may have been performed (T_2) and, finally, the body of the `Let ... In` is typechecked adding x as well as the new type of x (T_3) in the local context. We can notice that this rule is sequential (the rule must be read from the top to the bottom) and instead of allowing *parallel* evaluations, which would require a specific mechanism to merge the several constraints, we chose to impose an evaluation order giving the constraints from an evaluation to the next one.

[8] A version of this document can be found at: http://logical.inria.fr/~delahaye/these-delahaye.ps.

Procedural parts Finally, to deal with all the terms, we have to include the procedural parts in the relation $\triangleright$. To do so, we have to define the evaluation of tactics. This evaluation can be done only in verification mode because most of tactics do not provide a pure term as proof but a method to build a pure term from goals which they are applied to.

A tactic t is evaluated into v, with $v \in \mathcal{V}_\mathcal{T}$, in the goal $\Delta[\Gamma \Vdash T]$, which is noted $(t, \Delta[\Gamma \Vdash T]) \blacktriangleright v$, iff $(t, \Delta[\Gamma \Vdash T]) \blacktriangleright v$ holds.

For example, the `Cut` tactic, which is used to make a cut of a term, is evaluated by means of the following rule:

$$\frac{\begin{array}{c}(T_1, \Delta[\Gamma]) \triangleright (T_3, \Delta[\Gamma_1 \Vdash s], \emptyset, \sigma_1) \qquad s \in S\\ \mathsf{new\_meta}(n_1, n_2)\end{array}}{\begin{array}{c}(\mathtt{Cut}\ T_1, \Delta[\Gamma \Vdash T_2]) \blacktriangleright ((?n_1\ ?n_2), \Delta[\Gamma_1 \Vdash T_2\sigma_1],\\ \{(n_1, \Delta[\Gamma_1 \Vdash (x : T_3)T_2\sigma_1]), (n_2, \Delta[\Gamma_1 \Vdash T_3])\}, \sigma_1)\end{array}}\ (\mathsf{Cut})$$

where $\mathsf{new\_meta}(n_1, n_2, \ldots, n_p)$ ensures that the integers $n_1, n_2, \ldots, n_p$, numbering some metavariables, are unique.

As this rule generates a cut, that is to say an application in the proof term, this produces two subgoals, bound to metavariables $?n_1$ and $?n_2$, and the corresponding proof term is $(?n_1\ ?n_2)$. Once the term to be cut (T_1) evaluated (into T_3), the first goal (bound to $?n_1$), which corresponds to the head function, is a product type ($(x : T_3)T_2\sigma_1$) and the second subgoal (bound ot $?n_2$), which corresponds to the argument, is the result of the previous evaluation (T_3).

Now, using the relation $\blacktriangleright$[9], we can give the complete evaluation of the terms, including the procedural parts: a term t is evaluated, in the inference mode (resp. in the verification mode), into v, with $v \in \mathcal{V}_\mathcal{T}$, in the undefined goal $\Delta[\Gamma]$ (resp. in the goal $\Delta[\Gamma \Vdash T]$), iff $(t, \Delta[\Gamma]) \triangleright v$ (resp. $(t, \Delta[\Gamma \Vdash T]) \triangleright v$) holds.

Sentence semantics A sentence allows us to instantiate a metavariable or to add a lemma in the context of a metavariable. The values are the following:

- $(t, \Delta[\Gamma \Vdash T], m, p, d)$, where t is a pure term, $\Delta[\Gamma \Vdash T]$ is a goal, m is a set of couples metavariable numbers-goals $(i, \Delta[\Gamma_i \Vdash T_i])$, p is a stack of values ($\lfloor\rfloor$ is the empty stack and $\triangleleft$ is the concatenation operator), $d \in \mathcal{D}$ with $\mathcal{D} = \{\mathsf{Lemma}; \mathsf{Let}_i; \mathsf{Lemma}_b; \mathsf{Let}_{i,b}\}$, s.t. $\not\exists j,\ ?j \in \Delta[\Gamma \Vdash T]$ and $?j \in \Delta[\Gamma_i \Vdash T_i]$,
- Error

The set of those values is noted $\mathcal{V}_\mathcal{P}$ and a value will be also called a *state*.

Here, as for the term semantics, a value is given by a proof term t, the type of this proof term ($\Delta[\Gamma \Vdash T]$) and the set of the metavariables (bound to their types) m occurring in t. A difference is that we do not keep an instantiation of the implicit arguments and, in particular, this means that all the implicit arguments involved by a sentence must be solved during the evaluation of this

[9] The relation $\blacktriangleright$ is used by the relation $\triangleright$ during the evaluation of a `<by ... >` expression (rule By in [4]).

sentence (we cannot use the next sentences to do so). Moreover, in a value, we have also a stack of values, which will be used to store the current proof session when opening new proof sessions during the evaluation of `Let`'s which provide no proof term. Finally, in the same way, we add a declaration flag, which deals with the several possible declarations (`Lemma` or `Let`). Initially, this flag is set to Lemma and becomes Lemma_b when `Proof` is evaluated. This is similar for `Let` with the flags Let_i and $\mathsf{Let}_{i,b}$, where we give also a tag (here i) which is the metavariable corresponding to the upcoming proof of the `Let`.

A sentence "p." is evaluated into v, with $v \in \mathcal{V}_{\mathcal{P}}$, in the state (or value) $(t, \Delta[\Gamma \Vdash T], m, p, d)$, iff $(p., t, \Delta[\Gamma \Vdash T], m, p, d) \twoheadrightarrow v$ holds.

As an example, let us consider the evaluation of a `Let` in the current goal:

$$\frac{\begin{array}{c} d = \mathsf{Lemma}_b \text{ ou } d = \mathsf{Let}_{i,b} \\ n = min\{i \mid (?i, b) \in m\} \qquad (?n, \Delta[\Gamma_n \Vdash T_n]) \in m \qquad x \notin \Gamma_n \\ (T, \Delta[\Gamma_n]) \triangleright (T_2, \Delta[\Gamma_1 \Vdash s], \emptyset, \sigma) \qquad \mathsf{new\_meta}(k) \end{array}}{\begin{array}{c} (\mathtt{Let}\ x : T, t, \Delta[\Gamma \Vdash T_1], m, p, d) \Rrightarrow \\ (?1, \Delta[\Gamma_1 \Vdash T_2], \{(?1, \Delta[\Gamma_1 \Vdash T_2])\}, \\ \lfloor(([x : T_2]?n\ ?k), \Delta[\Gamma \Vdash T_1]\sigma, (m - \{(?n, \Delta[\Gamma_n \Vdash T_n])\})\sigma \cup \\ \{(?n, \Delta[\Gamma_n\sigma, (x : T_2) \Vdash T_n\sigma]; (?k, \Delta[\Gamma_1 \Vdash T_2])\}, p\sigma, d)\rfloor \triangleleft p\sigma, \mathsf{Let}_k) \end{array}} \ (\mathsf{LetCur})$$

In this rule[10], the first line consists only in verifying that we are in the context of a opened proof (introduced by `Lemma` or another toplevel `Let`, and where a `Proof` command has been evaluated). Next, we look for the current goal, which this `Let` is applied to. Our convention is to choose this goal such that it is bound to the lowest metavariable number (n). The type of the `Let` (T) is then evaluated (into T_2) and we generate a new metavariable ($?k$), which will correspond to the proof given to the `Let` (once `Qed` reached, this metavariable will be replaced by the actual proof). The result of the evaluation shows that we obtain a new goal bound to metavariable $?1$ (this will be also when opening a new proof session with `Lemma`). This goal has the context of the current goal (after the evaluation of T, i.e. Γ_1) and the evaluated type introduced by the `Let` (T_2). The current proof state is put in the stack. In this state, as expected, the global proof term is an application $(([x : T_2]?n\ ?k))$. Here, contrary to the previous tactic `Cut`, the type of the cut is introduced automatically (this is an expectable behavior) and this is why the proof term has a λ-abstraction in head position. Finally, the Let_k flag is set to declare that we are in the context of `Let` bound to metavariable $?k$. To accept other `Let` instructions or instantiations, a `Proof` command has to be processed to transform this flag into the flag $\mathsf{Let}_{k,b}$.

Script semantics Finally, the values of the script semantics are:

- $(t, \Delta[\Vdash T])$, where t is a pure term, $\Delta[\Vdash T]$ is a goal, s.t. $\nexists j,\ ?j, ?_{ij} \in t, T$.
- Error

[10] Here, we use the relation $\Rrightarrow$ which is related to the relation $\twoheadrightarrow$ just adding a dot (".") at the end of the sentence (rule Sen in [4]).

The set of those values is noted $\mathcal{V}_{\mathcal{S}}$.

As a script must provide a complete proof, a value is a proof term t and the type of this proof term $((t, \Delta[\Vdash T]))$, where there is no metavariable or implicit argument. The context of the type is empty because we suppose that a script is a proof of a lemma introduced in the toplevel of the global environment (Δ) by the command `Lemma`. Hence, we call *lemma*, every goal $\Delta[\Vdash T]$, where $\Delta[\Vdash T]$ is a goal introduced by `Lemma`.

A script "$p_1.\ p_2.\ \ldots\ p_n.$" of the lemma $\Delta[\Vdash T]$ is evaluated into v, with $v \in \mathcal{V}_{\mathcal{S}}$, iff $(p_1.\ p_2.\ \ldots\ p_n., \Delta[\Vdash T]) \downarrow v$ holds.

This evaluation consists mainly in iterating the rules for sentences.

4 Examples

We have implemented $\mathcal{L}_{pdt}$ for the latest versions V7 of Coq. Currently, this is a prototype which is separate from the official release and all the following examples have been carried out in this prototype.

4.1 Decidability of the equality on natural numbers

Now, we can go back to the example we introduced to compare the three proving styles. Using $\mathcal{L}_{pdt}$, the proof could look like the following script:

```
Lemma eq_nat:(n,m:nat)n=m\/~n=m.
Proof.
  Let b_n:(m:nat)(0)=m\/~(0)=m.
  Proof.
    ?1 := [m:nat]
          <[m:nat](0)=m\/~(0)=m>
          Cases m of
          | O => <by Left;Auto>
          | (S n) => <by Right;Auto>
          end.
  Qed.
  Let i_n:(n:nat)((m:nat)n=m\/~n=m)->(m:nat)(S n)=m\/~(S n)=m.
  Proof.
    ?1 := [n:nat;Hrec;m:nat]
          <[m:nat](S n)=m\/~(S n)=m>
          Cases m of
          | O => <by Right;Auto>
          | (S n0) => ?2
          end.
    ?2 := <[_:n=n0\/~n=n0](S n)=(S n0)\/~(S n)=(S n0)>
          Cases (Hrec n0) of
          | (or_introl _) => <by Left;Auto>
          | (or_intror _) => <by Right;Auto>
          end.
  Qed.
  ?1 := (nat_ind ([n:nat](m:nat)n=m\/~n=m) b_n i_n).
Save.
```

In this proof, we have mixed the three *proof worlds*. The first induction on n is done declaratively with two toplevel Let's for the basis (b_n) and the inductive (i_n) cases. The two cases are used to perform the induction on n, at the end of the script, by an instantiation (of ?1) with a term using nat_ind, the induction scheme on natural numbers (the first argument of nat_ind is the predicate to be proved by induction). The proofs of b_n and i_n are realized by instantiations with terms and trivial parts are settled by small procedural scripts, which are clearly identified (using <by ...>). In particular, reasonings by case are dealt with Cases constructions and the term between <...> is the corresponding predicate (the first λ-abstraction is the type of the term to be destructured and the rest is the type of the branches, i.e. the type of the Cases). Currently, these predicates have to be provided (by the user) because all the Cases used in this script are incomplete (presence of metavariables, even with procedural scripts, see section 3.2) and the typechecker (due to possible dependences) needs to be helped to give a type to metavariables.

More precisely, to solve the basis case (b_n), we use directly an instantiation (of ?1). We *assume* the variable m (with a λ-abstraction [...]) and we perform a reasoning by case (with Cases) on m. If m is equal to O, then this is trivial and we can solve it in a procedural way. We choose the left-hand side case with the tactic Left and we have to prove O=O, which is solved by the tactic Auto. Otherwise, if m is of the form (S n), it is quite similar and we choose the right-hand side case with the tactic Right to prove ~(S n)=O with Auto.

For the inductive case (i_n), we use also directly an instantiation (of ?1). We assume the induction hypothesis Hrec (coming from the induction on n) and we perform another reasoning by case on m. If m is equal to O, this is trivial and we solve it procedurally in the same way as in the basis case. Otherwise, if m is equal to (S n0), this is a bit more complicated and we decide to *delay* the proof using a metavariable (?2). Next, this metavariable is instantiated performing a reasoning by case on Hrec applied to n0. The two cases correspond to either n=n0 or ~n=n0 (or_introl/r is the first/second constructor of \/), which are trivially solved in a procedural way as in the basis case.

As said above, the last part of the script consists in solving the global lemma using an instantiation (of ?1) and performing the induction on n with the induction scheme nat_ind. This scheme takes three arguments: the predicate to be proved by induction, the proof of the basis case and the proof of the inductive case. The two last proofs have been already introduced by two toplevel Let's and they are directly used.

4.2 Stamp problem

As another example, let us consider the stamp problem, coming from the PVS tutorial [3] and asserting that every postage requirement of 8 cents or more can be met solely with stamps of 3 and 5 cents. Formally, this means that every natural number greater or equal to 8 is the sum of some positive multiple of 3 and some positive multiple of 5. This lemma can be expressed and proved (using $\mathcal{L}_{pdt}$) in the following way:

```
Lemma L3_plus_5:(n:nat)(EX t:Z|(EX f:Z|`(inject_nat n)+8=3*t+5*f`)).
Proof.
  Let cb:(EX t:Z|(EX f:Z|`(inject_nat O)+8=3*t+5*f`)).
  Proof.
    ?1 := (choose `1` (choose `1` <by Ring>)).
  Qed.
  Let ci:(n:nat)(EX t:Z|(EX f:Z|`(inject_nat n)+8=3*t+5*f`))->
               (EX t:Z|(EX f:Z|`(inject_nat (S n))+8=3*t+5*f`)).
  Proof.
    ?1 := [n;Hrec](skolem ?2 Hrec).
    ?2 := [x;H:(EX f:Z|`(inject_nat n)+8=3*x+5*f`)](skolem ?3 H).
    ?3 := [x0;H0:`(inject_nat n)+8 = 3*x+5*x0`]?4.
    Let cir:(EX t:Z|(EX f:Z|`3*x+5*x0+1=3*t+5*f`)).
    Proof.
      ?1 := <[_:?](EX t:Z|(EX f:Z|`3*x+5*x0+1=3*t+5*f`))>
            Cases (dec_eq `x0` `0`) of
            | (or_introl H) => ?2
            | (or_intror _) => ?3
            end.
      ?2 := (choose `x-3` (choose `2` <by Rewrite H;Ring>)).
      ?3 := (choose `x+2` (choose `x0-1` <by Ring>)).
    Qed.
    ?4 := <by Rewrite inj_S;Rewrite Zplus_S_n;Unfold Zs;Rewrite H0;
              Exact cir>.
  Qed.
  ?1 := (nat_ind ([n:nat]?) cb ci).
Save.
```

where `Z` is the type of integers, `EX` is the existential quantifier, `inject_nat` is the injection from `nat` to `Z` (this injection is needed because we have set this equality over `Z` in order to be able to use the decision procedure for Abelian rings, i.e. the tactic `Ring`), `choose` is syntactical sugar[11] for `(ex_intro ? ?)` (`ex_intro` is the single constructor of `EX`), `skolem` is syntactical sugar for `(ex_ind ? ? ?)` (`ex_ind` is the induction scheme of `EX`) and `(dec_eq` x_1 x_2`)` is equivalent to x_1=x_2\/~x_1=x_2.

This proof is done by induction (on `n`). The basis case is declared with the name `cb`, as well as the inductive case with the name `ci`. For `cb`, this is trivial, the instantiations are `1` and `1` (given by the term `choose`), since we have `0+8=3*1+5*1` (which is solved by the tactic `Ring`). For the inductive case, in the first instantiation (of `?1`), we introduce the induction hypothesis (`Hrec`), which is skolemized (with `skolem`). Then, in the second instantiation (of `?2`), we give a name for the skolem symbol (`x`) and we introduce the corresponding skolemized hypothesis (`H`), which is skolemized again. In the third instantiation (of `?3`), in the same way, we give another skolem name (`x0`) and we introduce the skolemized hypothesis (`H0`). At that point, we can notice that the conclusion can be transformed (in particular, moving the successor `S` outside `inject_nat`) in order to obtain the left-hand side member of the skolemized hypothesis as a

[11] This is done by means of toplevel syntactic definitions (see [9] for more details).

subterm. So, we carry out a cut (`cir`) of the conclusion after this *trivial step* (which require mainly rewritings). This auxiliary lemma is proved by case analysis (`Cases` term) according to the fact that `x0=0` or not. If `x0=0` (this is the first case, bound to metavariable `?2`), the instantiations are `x-3` and `2`, then `Ring` solves the goal. Otherwise (this is the second case, bound to metavariable `?3`), the instantiations are `x+2` and `x0-1`, then the equality is directly solved by `Ring` again. Once `cir` proved, we can perform, in a procedural way, the rewriting steps in the conclusion (we do not detail all the tactics and lemmas involved, which may require an advanced knowledge of Coq, and the reader only has to know what transformation is carried out by this procuderal part) to obtain the type of `cir` and to be able to conclude with the direct use of `cir` (with the tactic `Exact`). Finally, the induction over `n` is done using the term `nat_ind` directly with the corresponding proofs `cb` and `ci` as arguments.

Regarding proof style, we can notice that rewriting is, *a priori*, compatible with $\mathcal{L}_{pdt}$. The user only has to make a cut of the expression where the rewritings have been done and this cut is then directly used in the procedural part, which carries out the rewritings. This method allows us to use rewriting in a procedural way and in the middle of a proof. This seems to be a good method if rewriting is not heavily used. Otherwise, for example, for proofs which are built exclusively using rewritings (lemmas proved over axiomatizations), this could be a bit rigid and an interesting extension of $\mathcal{L}_{pdt}$ could consist in providing an appropriate syntax for equational reasoning.

5 Conclusion

5.1 Summary

In this paper, several points have been achieved:

- We have compared three proof styles, namely, the declarative, the procedural and the term style. This has been done by means of three well-known theorem provers, which are respectively, Coq, Mizar and Alfa. It has appeared that these three styles do not deserve to be opposed but were dedicated to specific kinds of proofs. A merging could allow the user greater freedom in the presentation of his/her proofs.
- In this view, a uniting language, called $\mathcal{L}_{pdt}$, has been designed in the context of the Coq proof system. In particular, the declarative features are available by means of `Let` ... `In`'s and toplevel `Let`'s. The procedural parts are directly inherited from the tactic language of Coq. Finally, the term language uses the intuitionistic Curry-Howard isomorphism in the context of Coq and is based on a language of incomplete proof terms (i.e. with metavariables), which can be refined by some instantiations.
- As a major originality of $\mathcal{L}_{pdt}$, a formal semantics has been given. This semantics consists essentially in building a proof term (always using Curry-Howard's isomorphism) corresponding to a given script. Here, the novelty has been to give also such a semantics for declarative and procedural parts.

– An implementation for versions V7 of Coq has been also realized. Some examples have been described and can be already evaluated in this prototype. In particular, those examples have shown that these three kinds of languages could naturally coexist in the same proof.

5.2 Extensions and future work

Some evolutions or improvements for $\mathcal{L}_{pdt}$ can be expected:

– As seen in the example of the stamp problem (subsection 4.2), $\mathcal{L}_{pdt}$ could be more appropriate to build easily proofs by rewritings. Such proofs can be made in $\mathcal{L}_{pdt}$, but this must be rather sporadic and especially trivial enough in such a way that the corresponding cut can be easily related to the initial goal. Thus, another method remains to be found to deal with proofs which use equational reasoning heavily. An idea could be to add a feature, which allows us to iterate some equalities, as e.g., in the Mizar-mode for HOL.
– $\mathcal{L}_{pdt}$ provides various `Let`'s, which add declarative features to the language (especially the toplevel `Let`'s). However, the way the lemmas introduced by some `Let`'s are combined to prove some other more complex lemmas can be improved in order to get a more declarative behavior. Indeed, as can be seen in the examples we have described (section 4), the auxiliary lemmas, which are declared by some `Let`'s, are simply and directly used. No specific automation is used to combine them and to solve. It would be interesting to add some automations here, because this is also a significative feature of declarative systems[12]. A first step, almost for free, could consist in using naturally the tactic `Auto` and adding systematically the lemmas declared by some `Let`'s in the database of this tactic. A more ambitious step would be to make `Auto` perform automatically some inductions.

5.3 Generalization and discussion

To conclude, we can wonder how such a proof language could be generalized to other logical frameworks and, in particular, how it could be applied to other proof systems. As can be seen, in $\mathcal{L}_{pdt}$, the term part has been clearly emphasized and, in a way, Curry-Howard's isomorphism is *brought to light*. Indeed, here, λ-terms are not only seen as a way of coding (or programming) proofs and giving a computational behavior to proofs, but also as a way of expressing proofs. So, $\mathcal{L}_{pdt}$ can be applied, almost for free, to other systems based on the (intuitionistic) Curry-Howard isomorphism, such as Lego or Nuprl. We can also include Alfa, although Alfa does not provide any tactic language yet (to add such a language to Alfa would certainly be a significative work, but more from a practical point of view than for theoretical reasons). Next, if we want to deal with other systems,

[12] For instance, ACL2 (successor of Nqthm) provides a huge automation and even Mizar, which is less automated, has a non trivial deduction system, which is hidden behind the keyword `by`.

the problem becomes a bit more difficult due to the term part. First, a good idea would be to consider only procedural systems (as said previously, to provide a full tactic language is a quite significative and tedious task), such as PVS or HOL. As those systems are generally based on classical logic, this means we have to consider a classical Curry-Howard isomorphism and we have to design another term language using some λC-calculi or $\lambda\mu$-calculi. Such an extension could be very interesting because as far as the author knows, no classical proof system based on such λ-calculi has been ever designed. Finally, if we deal with pure declarative systems, like Mizar or ACL2, the task is quite harder because we have to build the term part, as well as the procedural part (but again, this must be considered as a problem in practice and not from a theoretical point of view).

References

1. Thierry Coquand, Catarina Coquand, Thomas Hallgren, and Aarne Ranta. The Alfa Home Page, 2001.
   http://www.md.chalmers.se/~hallgren/Alfa/.
2. Yann Coscoy. A Natural Language Explanation for Formal Proofs. In C. Retoré, editor, *Proceedings of Int. Conf. on Logical Aspects of Computational Linguistics (LACL), Nancy*, volume 1328. Springer-Verlag LNCS/LNAI, September 1996.
3. Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A Tutorial Introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, April 1995.
4. David Delahaye. *Conception de langages pour décrire les preuves et les automatisations dans les outils d'aide à la preuve: une étude dans le cadre du système* Coq. PhD thesis, Université Pierre et Marie Curie (Paris 6), Décembre 2001.
5. John Harrison. Proof Style. In Eduardo Giménez and Christine Paulin-Mohring, editors, *Types for Proofs and Programs: International Workshop TYPES'96*, volume 1512 of *LNCS*, pages 154–172, Aussois, France, 1996. Springer-Verlag.
6. John Harrison. A Mizar Mode for HOL. In *J. von Wright, J. Grundy, and J. Harrison, editors, Theorem Proving in Higher Order Logics: TPHOLs '96*, volume 1125 of *LNCS*, pages 203–220, 1996.
7. Lena Magnusson. *The Implementation of* ALF*—a Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution.* PhD thesis, Chalmers University of Technology, 1994.
8. Don Syme. *Declarative Theorem Proving for Operational Semantics.* PhD thesis, University of Cambridge, 1998.
9. The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 7.3.* INRIA-Rocquencourt, May 2002.
   http://coq.inria.fr/doc-eng.html.
10. Andrzej Trybulec. The Mizar-QC/6000 logic information language. In *ALLC Bulletin (Association for Literary and Linguistic Computing)*, volume 6, pages 136–140, 1978.
11. Markus Wenzel. Isar - A Generic Interpretative Approach to Readable Formal Proof Documents. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, volume 1690 of *LNCS*, pages 167–184. Springer-Verlag, 1999.
12. Vincent Zammit. *On the Readability of Machine Checkable Formal Proofs.* PhD thesis, University of Kent, Canterbury, October 1998.