A Declarative Language For The Coq Proof Assistant.[⋆]

Pierre Corbineau

pierre.corbineau@cs.ru.nl
Institute for Computing and Information Science
Radboud University Nijmegen, Postbus 9010
6500GL Nijmegen, The Netherlands

Abstract. This paper presents a new proof language for the Coq proof assistant. This language uses the declarative style. It aims at providing a simple, natural and robust alternative to the existing $\mathcal{L}_{tac}$ tactic language. We give the syntax of our language, an informal description of its commands and its operational semantics. We explain how this language can be used to implement formal proof sketches. Finally, we present some extra features we wish to implement in the future.

1 Introduction

1.1 Motivations

An interactive proof assistant can be described as a state machine that is guided by the user from the 'statement ϕ to be proved' state to the 'QED' state. The system ensures that the state transitions (also known as *proof steps* in this context) are sound. The user's guidance is required because automated theorem proving in any reasonable logics is undecidable in theory and difficult in practice. This guidance can be provided either through some kind of text input or using a graphical interface and a pointing device. In this paper, we will concentrate on on the former method.

The ML language developed for the LCF theorem prover [GMW79] was a seminal work in this domain. The ML language was a fully-blown programming language with specific functions called *tactics* to modify the *proof state*. The tool itself consisted of an interpreter for the ML language. Thus a formal proof was merely a computer program. With this in mind, think of a person reading somebody else's formal proof, or even one of his/her own proofs but after a couple of months. Similarly to what happens with source code, this person will have a lot of trouble understanding what is going on with the proof unless he/she has a very good memory or the proof is thoroughly documented. Of course, running the proof through the prover and looking at the output might help a little.

This example illustrates a big inconvenience which still affects man popular proof languages used nowadays: they lack readability. Most proofs written are

⋆ This work was partially funded by NWO Bricks/Focus Project 642.000.501

actually write-only or rather write- and execute-only, since what the user is interested when re-running the proof in is not really the input, but rather the output of the proof assistant, i.e. The sequence of proof states from stating the theorem to 'QED'.

The idea behind declarative style proofs is to actually base the proof language on this sequence of proof states. This is indeed the very feature that makes the distinction between procedural proof languages (like ML tactics in LCF) and declarative proof languages. On one hand, procedural languages emphasize proof methods (application of theorems, rewriting, proof by induction...), at the expense of a loss of precision on intermediate proof states: the intermediate states depend on the implementation on tactics instead of a formal semantics. On the other hand, declarative languages emphasize proof states but are less precise about the logical justification of the gap between one state and the next one.

1.2 Related work

The first proof assistant to implement a declarative style proof language was the Mizar system, whose modern versions date back to the early 1990's. The Mizar system is a batch style proof assistant: it compiles whole files and writes error messages in the body of the input text, so it is not exactly interactive, but its proof language has been an inspiration for all later designs [WW02].

Another important source in this subject is Lamport's *How to write a proof* [Lam95] which takes the angle of the mathematician and provides a very simple system for proof notations, aimed at making proof verification as simple as possible.

The first interactive theorem prover to actually provide a declarative proof language has been Isabelle [Law94], with the Isar (Intelligible Semi-Automated Reasoning) language [Wen99], designed by Markus Wenzel. This language has been widely adopted by Isabelle users since.

John Harrison has also developed a declarative proof language for the HOL88 theorem prover [Har96]. Freek Wiedijk also developed a light declarative solution [Wie01] for John Harrison's own prover HOL Light [Har06].

For the Coq proof assistant [Coq07], Mariusz Giero and Freek Wiedijk have built a set of tactics called the MMode [GW04] to provide an experimental mathematical mode which give a declarative flavor to Coq.

Recently, Claudio Sacerdoti-Coen added a declarative language to the Matita proof assistant [Coe06].

1.3 A new language for the Coq proof assistant

The Coq proof assistant is a Type Theory-based interactive proof assistant developed at INRIA. It has a strong user base both in the field of software and hardware verification and in the field of formalized mathematics. It also has the reputation of being a tool whose procedural proof language $\mathcal{L}_{tac}$ has a very steep learning curve both at the beginner and advanced level.

Coq has been evolving quite extensively during the last decade, and the evolution has made it necessary to regularly update existing proofs to maintain compatibility with most recent versions of the tool.

Coq also has a documentation generation tool to do hyper-linked rendering of files containing proofs, but most proofs are written in a style that makes them hard to understand (even with colored keywords) unless you can actually run them, as was said earlier for other procedural proof languages.

Building on previous experience from Mariusz Giero, we have built a stable mainstream declarative proof language for Coq. This language was built to have the following characteristics:

readable The designed language should use clear English words to make proof reading a feasible exercise.

natural We want the language to use a structure similar to the ones used in textbook mathematics (e.g. for case analysis), not a bare sequence of meaningless commands.

maintainable The new language should make it easy to upgrade the prover itself: errors in the proof should not propagate.

stand-alone The proof script should contain enough explicit information to be able to retrace the proof path without running Coq.

The Mizar language has been an important source of inspiration in this work but additional considerations had to be taken into account because the Calculus of Inductive Constructions (CIC) is mush richer than Mizar's (essentially) first-order Set Theory.

One of the main issue is that of proof genericity: Coq proofs use a lot of inductive objects for lots of different applications (logical connectives, records, natural numbers, algebraic data-types, inductive relations. . .). Rather than enforcing the use of the most common inductive definitions, we want to be as generic as possible in the support we give for reasoning with these objects.

Finally, we want to give an extended support for proofs by induction by allowing multi-level induction proofs, using a very natural syntax to specify the different cases in the proof. The implementation was part of the official release 8.1 version of Coq.

1.4 Outline

We first describe some core features of our language, such as forward and backward steps, justifications, and partial conclusions. Then we give a formal syntax and a quick reference of the commands of our language, as well as an operational semantics. We go on by explaining how our language is indeed an implementation of the *formal proof sketches* [Wie03] concept, and we define the notion of well-formed proof. We finally give some perspective for future work.

2 Informal description

2.1 An introductory example

To give a sample of the declarative language, we provide here the proof of a simple lemma about Peano numbers: the `double` function is defined by $\texttt{double}\, x = x{+}x$ and the $\texttt{div}_2$ functions by:

$$\begin{cases} \texttt{div}_2\, 0 = 0 \\ \texttt{div}_2\, 1 = 0 \\ \texttt{div}_2\, (S\, (S\, x)) = S\, (\texttt{div}_2\, x) \end{cases}$$

The natural numbers are defined by means of an inductive type `nat` with two constructors `0` and `S` (successor function. The lemma states that $\texttt{div}_2$ is the left inverse of `double`. We first give a proof of the lemma using the usual tactic language:

```
Lemma double_div2: forall n, div2 (double n) = n.
intro n.
induction n.
reflexivity.
unfold double in *|-*.
simpl.
rewrite <- plus_n_Sm.
rewrite IHn.
reflexivity.
Qed.
```

Now, we give the same proof using the new declarative language:

```
Lemma double_div2: forall n, div2 (double n) = n.
proof.
  let n:nat.
  per induction on n.
    suppose it is 0.
     reconsider thesis as (0=0).
     thus thesis.
    suppose it is (S m) and Hrec:thesis for m.
      have (div2 (double (S m))
            = div2 (S (S (double m)))).
           ~= (S (div2 (double m))).
      thus ~= (S m) by Hrec.
  end induction.
end proof.
Qed.
```

The proof consists of a simple induction on the natural number `n`. The first case is done by conversion (computation) to $0 = 0$ and the second case by computation and by rewriting the induction hypothesis. Of course, you could have guessed that by simply reading the declarative proof.

2.2 Forward and backward proofs

The notions of declarative and procedural proofs are often confused with the notions of forward and backward proof. We believe those two notions are mostly orthogonal: the distinction between declarative and procedural is that declarative proofs mention explicitly the intermediate proof steps, while procedural proofs explain what method is used to go to the next state without mentioning it, whereas the distinction between forward and backward proof is merely a notion of whether the proof is build bottom-up by putting together smaller proofs, or top-down by cutting a big proof obligation into smaller ones.

A possible reason for the confusion between declarative style and backwards proofs is that most declarative languages rely on the core command

$$\texttt{have}\ h : \phi\ justification$$

which introduces a new hypothesis h that asserts ϕ, after having proved it using *justification*. This kind of command is the essence of forward proofs: it builds a new object — a proof of ϕ — from objects that already exists and are somehow mentioned in the justification.

In order to show that you can also use backwards steps in a declarative script, our language contains a command

$$\texttt{suffices}\ \ H_1\ \texttt{and}\ \ldots\ \texttt{and}\ H_n\ \texttt{to show}\ G\ justification$$

which acts as the dual of the `have` construction: it allows to replace the current statement to prove by sufficient conditions with stronger statements (as explained by the justification). For example, you can use this command to generalize your thesis before stating a proof by induction.

2.3 Justifications

When using a command to deduce a new statement from existing ones, or to prove that some statements suffice to prove a part of the conclusion, you need the proof assistant to fill in the gap. The justification is a hint to tell the proof assistant which proof objects are to be used in the process, and how they shall be used.

In our language, justifications are of the form

$$\texttt{by}\ \pi_1, \ldots, \pi_n\ \texttt{using t}$$

the π_k are proof objects which can be hypotheses names (variables) but also more complex terms like application of a general result $H : (\forall x : \texttt{nat}, P\,x)$ to a specific object n : `nat` to get a proof `(H n)` : $P\,n$. The expression `t` is an optional tactic expression that will be used to prove the validity of the step.

The meaning of the π_k objects is that only they and their dependencies — if H is specified and H has type $P\,x$ then x is also implicitly added — will be in the local context in which the new statement is to be proved. If they are omitted then the new statement should be either a tautology or provable without any local hypothesis by the means of the provided tactic. If the user types `by *`, all local hypotheses can be used. The use of `by *` should be transient because it makes the proof more brittle: the justification depends too much on what happened before.

If specified, the tactic `t` will then be applied to the modified context and if necessary, the remaining subgoals will be treated with the `assumption` tactic, which looks for a hypothesis convertible with the conclusion. If `using t` is omitted then a default automation tactic is used. This tactic is a combination of `auto`, `congruencecongruence` and `firstorder` [Cor03].

When writing a sequence of deduction steps, it often happens that a statement is only used in the next step. In that case the statement might be anonymous. By using the `then` keyword instead of `have`, this anonymous statement will be added to the list of proof objects to be used in that particular justification.

If a justification fails, the proof assistant issues a warning

```
Warning: insufficient justification.
```

rather than an error. This allows the user to write proofs from the outside in by filling the gaps, rather than linearly from start to end. In the CoqIDE interface, this warning is emphasized by coloring the corresponding command with an orange background. This way, a user reading a proof script will immediately identify where work still needs to be done.

2.4 Partial conclusion, split thesis

The idea of partial conclusions is that within a deduction chain, some steps are actually sub-formulae of the current conclusion , so they can be used to remove that sub-formula from that conclusion. For example, A is a partial conclusion for $A \wedge B$, it is also a partial conclusion for $A \vee B$. In the latter case, choosing to prove A implies a choice in the proof we want to make, by proving A rather than B. In our language, the user can type the command

$$\texttt{thus}\ G\ justification$$

to provide a partial conclusion G whose validity is proved using *justification*. If G is not proved, the usual warning is issued, but if G is not a sub-formula of the conclusion then an error occurs: the user is trying to prove the wrong formula.

More precisely, the notion of partial conclusion is a consequence of the definition of logical connectives by inductive types. We will look for partial conclusions in the possible sub-terms of proof terms based on inductive type constructors.

thesis	partial conclusion	remaining conclusions
$A \wedge B$	A	$?_1 : B$
$A \wedge B$	B	$?_1 : A$
$A \wedge (B \wedge C)$	A	$?_1 : B \wedge C$
$(A \wedge B) \wedge C$	A	$?_1 : B, ?_2 : C$
$A \vee B$	A	-
$A \vee B$	B	-
$(A \wedge B) \vee (C \wedge D)$	C	$?_1 : D$
$\exists x : \texttt{nat}, P\,x$	$2 : \texttt{nat}$	$?_1 : P\,2$
$\exists x : \texttt{nat}, P\,x$	$P\,2$	-
$\exists x : \texttt{nat}, \exists y : \texttt{nat}, (P\,y \wedge R\,x\,y)$	$P\,2$	$?_1 : \texttt{nat}, ?_2 : R\,?_1\,2$

Fig. 1. the partial conclusion mechanism

In the case of the conjunction $A \wedge B$, if we have a proof π of A, using the pairing constructor `conj` we can build a proof $\texttt{conj}\,\pi\,?_1$, where $?_1$ is a place-holder for the remaining item to be proved (i.e. B). This way, we can introduce the notion of remaining conclusions (see Fig. 1) . The plural here is because, unfortunately, it might be possible that partial conclusions cause the thesis to split (i.e. several place-holders are needed in the partial proof). It might even happen that a part of the split thesis depends on another: keep in mind that the existential quantifier is represented by a dependent pair. Finally, when using $P2$ as a partial conclusion for $\exists x : \texttt{nat}, Px$, even though a placeholder for `nat` should remain, this placeholder has to be filled by 2 because of typing constraints.

Using this mechanism, we can allow the user to build the proof piece by piece, by providing partial conclusions, and at each step replacing the part of the thesis by the remaining parts to be proved. Parts of the thesis are given numbers and their type can be referred to by using `thesis[`n`]` for the type of placeholder $?_n$. The keyword `thesis` alone refers to the type of the unique placeholder when there is only one. When there are more than one place-holders, the thesis is said to be split.

A split thesis will induce some constraints on what the user can do, e.g. the user cannot perform a proof by cases or introduce a hypothesis. Therefore in some cases is is preferable to avoid those situation by proving all the conclusion at once.

3 Syntax and semantics

3.1 Syntax

Figure 2 gives the complete formal syntax of the declarative language. the un-bound non-terminals are *id* for identifiers, *num* for natural numbers, *term* and *type* for terms and types of the Calculus of Inductive Constructions, *pattern* refers to a pattern for matching against inductive objects.

```
instruction   ::= proof
                | assume statement [and statement]* [and (we have)-clause]?
                | (let, be)-clause
                | (given)-clause
                | (consider)-clause from term
                | [have|then|thus|hence] statement justification
                | thus? [~ =|= ~] [id:]? term justification
                | suffices statement [and statement]*
                  to show statement justification
                | [claim|focus on] statement
                | take term
                | define id [var[,var]*]? as term
                | reconsider [id|thesis[[num]]?] as type
                | per [cases|induction] on term
                | per cases of type justification
                | suppose [id[,id]* and]? is is pattern
                  [such that statement [and statement]*[and (we have)-clause]?]?
                | end [proof|claim|focus|cases|induction]
                | escape
                | return

α, β-clause   ::= α var[,var]* [β such that statement [and statement]*
                  [and α, β-clause]?]?

statement     ::= [id:]? type
                | thesis
                | thesis[num]
                | thesis for id

var           ::= id[:type]?

justification ::= [by [*|term[,term]*]]? [using tactic]?
```

Fig. 2. Syntax for the declarative language

	simple	with previous step	opens sub-proof	iterated equality
intermediate step	have	then	claim	~=/=~
conclusive step	thus	hence	focus on	thus ~=/thus =~

Fig. 3. Synthetic classification of forward steps

3.2 Commands description

`proof.`
` ...`
`end proof.` This is the outermost block of any declarative proof. If several subgoals existed when the `proof` command occurred, only the first one is proved in the declarative proof. If the proof is not complete when encountering `end proof`, then the proof is closed all the same, but with a warning, and `Qed` or `Defined` to save the proof will fail.

`have` h`:`ϕ *justification*`.`
`then` h`:`ϕ *justification*`.` This command adds a new hypothesis h of type ϕ in the context. If the justification fails, a warning is issued but the hypothesis is still added to the context. The `then` variant adds the previous fact to the list of objects used in the justification.

`thus` h`:`ϕ *justification*`.`
`hence` h`:`ϕ *justification*`.` These commands behave respectively like `have` and `then` but the proof of ϕ is used as a partial conclusion. This can end the proof or remove part of the proof obligations. These commands fail if ϕ is not a sub-formula of the thesis.

`claim` $h : \phi$.
` ...`
`end claim.` This block contains a proof of ϕ which will be named h after `end claim`. If the subproof is not complete when encountering `end claim`, then the subproof is still closed, but with a warning, and `Qed` or `Defined` to save the proof later will fail.

`focus on` ϕ.
` ...`
`end focus.` This block is similar to the `claim` block, except that it leads to a partial conclusion. In a way, `focus` is to `claim` what `thus` is to `have`. This comes handy when the thesis is split and one of its parts is an implication or a universal quantification: the `focus` block will allow to use local hypotheses.

`(thus) ~=` t *justification*.
`(thus) =~` t *justification*. These commands can only be used if the last step was an equality $l = r$. t should be a term of the same type as l and r. If `~=` is used then the *justification* will be used to prove $l = t$ and the new statement will be $l = t$. Otherwise, the *justification* will be used to prove $t = r$ and the new statement will be $t = r$. When present, the `thus` keyword will trigger a conclusion step.

`suffices` $H : \Phi$ `to show` Ψ *justification*`.` This command allows to replace a part of the thesis by a sufficient condition, e.g. to strengthen it before starting an proof by induction. In the thesis, Ψ is then replaced by Φ. The justification should prove Ψ using Φ.

`assume G:`Ψ `...and we have x such that H:`Φ`.`
`let x be such that H:`Φ`.` Those commands are two different flavors for the introduction of hypothesis. They expect the thesis not to be split, and of the shape $\Pi_i x_i : Ti.Gi$. It expects the T_i to be

convertible with the provided hypotheses statements. This command is well-formed only if the missing types can be inferred.

`given` x `such that` H`:`Φ`.`
`consider` x `such that` H`:`Φ `from` G`.` `given` is similar to `let`, except that this command works up to elimination of tuples and dependent tuples such as conjunctions and existential quantifiers. Here the thesis could be $\exists x.\Phi' \rightarrow \Psi$ with Φ' convertible to Φ. The `consider` command takes an explicit object `G` to destruct instead of using an introduction rule.

`define` f $(x : T)$ `as` t`.` This command allows to defines objects locally. If parameters are given, a function (λ-abstraction) is defined.

`reconsider thesis as` T`.`
`reconsider` H `as` T`.` These commands allows to replace the statement of a conclusion or a hypothesis with a convertible one, and fails if the provided statement is not convertible.

`take` t`.` This command allows to do a partial conclusion using an explicit proof object. This is especially useful when proving an existential statement: it allows to specify the existential witness.

`per cases on` t`.`
— `of` F *justification*`.`
`suppose` $x : H$`.`
 `...`
`suppose` $x' : H'$`.`
 `...`
`end cases.` This introduces a proof per cases on a disjunctive proof object t or a proof of the statement F derived from *justification*. The `per cases` command must immediately be followed by a `suppose` command which will introduce the first case. Further `suppose` commands or `end cases` can be typed even if the previous case is not complete. In that case a warning is issued. This block of commands cannot be used when the thesis is split. If t occurs in the thesis, you should use `suppose it is` instead of `suppose`.

`per induction on` t`.`
— `cases` —
`suppose it is` *patt* `and` $x : H$`.`
 `...`
`suppose it is` *patt*$'$ `and` $x' : H'$`.`
 `...`
`end cases.` This introduces a proof per dependent cases or by induction. When doing the proof, t is substituted with *patt* in the thesis. *patt* must be a pattern for a value of the same type as t. It may contain arbitrary sub-patterns and `as` statements to bind names to sub-patterns. Those name aliases are necessary to apply the induction hypothesis at multiple levels. If you are doing a proof by induction, you may use the *thesisfor* construction in the `suppose it is` command to refer to induction hypothesis. You may also write induction hypotheses explicitly.

`escape.`
` ...`
`return.` This block allows to escape the declarative mode back to the tactic mode. If the thesis is split, then several subgoals are provided, or the command fails if some parts depend on others.

3.3 Operational semantics

The purpose of this section is to give precise details about what happens to the proof state when you type a proof command. The proof state consists of a stack $\mathcal{S}$ that contains open proofs and markers to count open sub-proofs, and each sub-proof is a judgement $\Gamma \vdash \Delta$, where Γ and Δ are lists of types (or propositions) indexed by names for Γ and integers for Δ. The intuition is that we are looking for a proof of the *conjunction* of the types in Δ from the hypotheses in Γ. When Δ is empty, the sub-proof is complete and the user has to close the block. If at some point Δ is supposed to be empty and is not, then the command issues a warning.

A proof script $\mathfrak{S}$ consists of the concatenation of instructions The rules are given as big-step semantics $\mathfrak{S} \Rightarrow_{\mathfrak{T}} \mathcal{S}$ means that when proving theorem $\mathfrak{T}$, we reach state $\mathcal{S}$ when executing the script $\mathfrak{S}$. This means that any script allowing to reach the empty stack $[]$ is a complete proof of the theorem $\mathfrak{T}$.

The $j \triangleright \Gamma \vdash G$ expression means that the justification j is sufficient to solve the problem $\Gamma \vdash G$. If it is not, the command issues a warning. The $\equiv$ relation is the conversion ($\beta\delta\iota\zeta$-equivalence) relation of the Calculus of Inductive Constructions (see [Coq07]).

We write $L \sqsubseteq R$ whenever the L context can be obtained by decomposing tuples in the R context. The $-$ operator computes the remaining parts of the thesis when a sub-formula is already proved. The details of the $-$ are too technical to be written here. We use the traditional λ notation for abstractions and Π for dependent products (either implication or universal quantification, depending on the context).

The distinction between cases_d and cases_{nd} is used to prevent the mixing of `suppose` with `suppose it is`. The coverage condition for case analysis has been omitted for simplicity, as have been the semantics for `escape` and `return`.

$$\frac{\mathfrak{T} = \{\Gamma \vdash G\}}{\texttt{proof.} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash ?_1 : G);[]} \qquad \frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \emptyset);[]}{\mathfrak{S}\ \texttt{end proof.} \Rightarrow_{\mathfrak{T}} []}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \Delta);\mathcal{S} \quad j \triangleright \Gamma \vdash T \quad \Delta \neq \emptyset}{\mathfrak{S}\ \texttt{have}\ (x : T)\ j\texttt{.} \Rightarrow_{\mathfrak{T}} (\Gamma; x : T \vdash \Delta);\mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma; l : T' \vdash \Delta);\mathcal{S} \quad j, l \triangleright \Gamma \vdash T \quad \Delta \neq \emptyset}{\mathfrak{S}\ \texttt{then}\ (x : T)\ j\texttt{.} \Rightarrow_{\mathfrak{T}} (\Gamma; l : T'; x : T \vdash \Delta);\mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \Delta);\mathcal{S} \quad j \triangleright \Gamma \vdash T \quad \Delta \neq \emptyset}{\mathfrak{S}\ \texttt{thus}\ (x : T)\ j\texttt{.} \Rightarrow_{\mathfrak{T}} (\Gamma; x : T \vdash \Delta - (x : T));\mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma; l : T' \vdash \Delta); \mathcal{S} \quad j, l \triangleright \Gamma \vdash T \quad \Delta \neq \emptyset}{\mathfrak{S}\ \texttt{hence}\ (x : T)\ j\texttt{.} \Rightarrow_{\mathfrak{T}} (\Gamma; l : T'; x : T \vdash \Delta - (x : T)); \mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \Delta); \mathcal{S} \quad j \triangleright \Gamma \vdash r = u \quad \Delta \neq \emptyset}{\mathfrak{S}\ \texttt{\textasciitilde=}\ u\ j\texttt{.} \Rightarrow_{\mathfrak{T}} (\Gamma; e : l = u \vdash \Delta); \mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \Delta); \mathcal{S} \quad j \triangleright \Gamma \vdash u = l \quad \Delta \neq \emptyset}{\mathfrak{S}\ \texttt{=\textasciitilde}\ u\ j\texttt{.} \Rightarrow_{\mathfrak{T}} (\Gamma; e : u = r \vdash \Delta); \mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \Delta); \mathcal{S} \quad j \triangleright \Gamma \vdash r = u \quad \Delta \neq \emptyset}{\mathfrak{S}\ \texttt{thus}\ \texttt{\textasciitilde=}\ u\ j\texttt{.} \Rightarrow_{\mathfrak{T}} (\Gamma; e : l = u \vdash \Delta - (l = u)); \mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \Delta); \mathcal{S} \quad j \triangleright \Gamma \vdash u = l \quad \Delta \neq \emptyset}{\mathfrak{S}\ \texttt{thus}\ \texttt{=\textasciitilde}\ u\ j\texttt{.} \Rightarrow_{\mathfrak{T}} (\Gamma; e : u = r \vdash \Delta - (u = r)); \mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \Delta); \mathcal{S} \quad \Delta \neq \emptyset}{\mathfrak{S}\ \texttt{claim}\ (x : T)\texttt{.} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash ?_1 : T); \mathsf{claim}; (\Gamma; x : T \vdash \Delta); \mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma' \vdash \emptyset); \mathsf{claim}; (\Gamma \vdash \Delta); \mathcal{S}}{\mathfrak{S}\ \texttt{end claim.} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \Delta); \mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \Delta); \mathcal{S} \quad \Delta \neq \emptyset}{\mathfrak{S}\ \texttt{focus on}\ (x : T)\texttt{.} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash ?_1 : T); \mathsf{focus}; (\Gamma; x : T \vdash \Delta - (x : T)); \mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma' \vdash \emptyset); \mathsf{focus}; (\Gamma \vdash \Delta); \mathcal{S}}{\mathfrak{S}\ \texttt{end focus.} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \Delta); \mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \Delta); \mathcal{S} \quad \Gamma \vdash t : T \quad \Delta \neq \emptyset}{\mathfrak{S}\ \texttt{take}\ t\texttt{.} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \Delta - (t : T)); \mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \Delta); \mathcal{S} \quad \Gamma;\ x_1 : T_1, \ldots, x_n : T_n \vdash t : T \quad \Delta \neq \emptyset}{\begin{array}{c}\mathfrak{S}\ \texttt{define}\ f\ (x_1 : T_1) \ldots (x_n : T_n)\ \texttt{as}\ t\texttt{.} \Rightarrow_{\mathfrak{T}} \\ (\Gamma; f :=\ \lambda x_1 : T_1 \ldots \lambda x_n : T_n . t \vdash \Delta); \mathcal{S}\end{array}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash ?_p :\ \Pi x_1 : T'_1 \ldots \Pi x_n : T'_n . G); \mathcal{S} \quad (T_1 \ldots T_n) \equiv (T'_1 \ldots T'_n)}{\mathfrak{S}\ \texttt{assume/let}\ (x_1 : T_1) \ldots (x_n : T_n)\texttt{.} \Rightarrow_{\mathfrak{T}} (\Gamma;\ x_1 : T_1; \ldots; x_n : T_n \vdash ?_1 : G); \mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash ?_p :\ \Pi x_1 : T'_1 \ldots \Pi x_n : T'_n . G); \mathcal{S} \quad (T_1 \ldots T_m) \sqsubseteq (T'_1 \ldots T'_n)}{\mathfrak{S}\ \texttt{given}\ (x_1 : T_1) \ldots (x_m : T_m)\texttt{.} \Rightarrow_{\mathfrak{T}} (\Gamma;\ x_1 : T_1; \ldots; x_m : T_m \vdash ?_1 : G); \mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \Delta); \mathcal{S} \quad \Gamma \vdash t : T \quad (T_1 \ldots T_n) \sqsubseteq (T) \quad \Delta \neq \emptyset}{\begin{array}{c}\mathfrak{S}\ \texttt{consider}\ (x_1 : T_1) \ldots (x_n : T_n)\ \texttt{from}\ t\texttt{.} \Rightarrow_{\mathfrak{T}} \\ (\Gamma;\ x_1 : T_1; \ldots; x_n : T_n \vdash \Delta); \mathcal{S}\end{array}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma; x : T' \vdash \Delta); \mathcal{S} \quad T \equiv T' \quad \Delta \neq \emptyset}{\mathfrak{S}\ \texttt{reconsider}\ x\ \texttt{as}\ T\texttt{.} \Rightarrow_{\mathfrak{T}} (\Gamma; x : T \vdash \Delta); \mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \Delta; ?_p : T'); \mathcal{S} \quad T \equiv T' \quad \Delta \neq \emptyset}{\mathfrak{S}\ \texttt{reconsider thesis[}p\texttt{] as } T\texttt{.} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \Delta; ?_p : T); \mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \Delta); \mathcal{S} \quad j \triangleright \Gamma; x_1 : T_1; \ldots; x_n : T_n \vdash T \quad \Delta \neq \emptyset}{\begin{array}{c}\mathfrak{S}\ \texttt{suffices } (x_1 : T_1) \ldots (x_n : T_n) \texttt{ to show } T\ j\texttt{.} \Rightarrow_{\mathfrak{T}} \\ (\Gamma \vdash (\Delta - T) \uplus \{T_1; \ldots; T_n\}); \mathcal{S}\end{array}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash ?_p : G); \mathcal{S} \quad j \triangleright \Gamma \vdash t : T}{\mathfrak{S}\ \texttt{per cases of } T\ j\texttt{.} \Rightarrow_{\mathfrak{T}} \mathsf{cases}_{\mathsf{nd}}(t : T); (\Gamma; x : T \vdash ?_p : G); \mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash ?_p : G); \mathcal{S} \quad \Gamma \vdash t : T}{\mathfrak{S}\ \texttt{per cases on } t\texttt{.} \Rightarrow_{\mathfrak{T}} \mathsf{cases}(t : T); (\Gamma; x : T \vdash ?_p : G); \mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} \mathsf{cases}(t : T); (\Gamma \vdash ?_p : G); \mathcal{S}}{\begin{array}{c}\mathfrak{S}\ \texttt{suppose } (x_1 : T_1) \ldots (x_n : T_n)\texttt{.} \Rightarrow_{\mathfrak{T}} \\ (\Gamma; x_1 : T_1; \ldots; x_n : T_n \vdash ?_p : G); \mathsf{cases}_{\mathsf{nd}}(t : T); (\Gamma \vdash ?_p : G); \mathcal{S}\end{array}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma' \vdash \emptyset); \mathsf{cases}_{\mathsf{nd}}(t : T); (\Gamma \vdash ?_p : G); \mathcal{S}}{\begin{array}{c}\mathfrak{S}\ \texttt{suppose } (x_1 : T_1) \ldots (x_n : T_n)\texttt{.} \Rightarrow_{\mathfrak{T}} \\ (\Gamma; x_1 : T_1; \ldots; x_n : T_n \vdash ?_p : G); \mathsf{cases}_{\mathsf{nd}}(t : T); (\Gamma \vdash ?_p : G); \mathcal{S}\end{array}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} \mathsf{cases}(t : T); (\Gamma \vdash ?_p : G); \mathcal{S}}{\begin{array}{c}\mathfrak{S}\ \texttt{suppose it is } p \texttt{ and } (x_1 : T_1) \ldots (x_n : T_n)\texttt{.} \Rightarrow_{\mathfrak{T}} \\ (\Gamma; x_1 : T_1; \ldots; x_n : T_n \vdash ?_p : G[p/t]); \mathsf{cases}_{\mathsf{d}}(t : T); (\Gamma \vdash ?_p : G); \mathcal{S}\end{array}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma' \vdash \emptyset); \mathsf{cases}_{\mathsf{d}}(t : T); (\Gamma \vdash ?_p : G); \mathcal{S}}{\begin{array}{c}\mathfrak{S}\ \texttt{suppose it is } p \texttt{ and } (x_1 : T_1) \ldots (x_n : T_n)\texttt{.} \Rightarrow_{\mathfrak{T}} \\ (\Gamma; x_1 : T_1; \ldots; x_n : T_n \vdash ?_p : G[p/t]); \mathsf{cases}_{\mathsf{d}}(t : T); (\Gamma \vdash ?_p : G); \mathcal{S}\end{array}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma' \vdash \emptyset); \mathsf{cases}_{\mathsf{d/nd}}(t : T); (\Gamma \vdash ?_p : G); \mathcal{S}}{\mathfrak{S}\ \texttt{end cases.} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \emptyset); \mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash ?_p : G); \mathcal{S} \quad \Gamma \vdash t : T}{\mathfrak{S}\ \texttt{per induction on } t\texttt{.} \Rightarrow_{\mathfrak{T}} \mathsf{induction}(t : T); (\Gamma; x : T \vdash ?_p : G); \mathcal{S}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} \mathsf{induction}(t : T); (\Gamma \vdash ?_p : G); \mathcal{S}}{\begin{array}{c}\mathfrak{S}\ \texttt{suppose it is } p \texttt{ and } (x_1 : T_1) \ldots (x_n : T_n)\texttt{.} \Rightarrow_{\mathfrak{T}} \\ (\Gamma; x_1 : T_1; \ldots; x_n : T_n \vdash ?_p : G[p/t]); \mathsf{induction}(t : T); (\Gamma \vdash ?_p : G); \mathcal{S}\end{array}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma' \vdash \emptyset); \mathsf{induction}(t : T); (\Gamma \vdash ?_p : G); \mathcal{S}}{\begin{array}{c}\mathfrak{S}\ \texttt{suppose it is } p \texttt{ and } (x_1 : T_1) \ldots (x_n : T_n)\texttt{.} \Rightarrow_{\mathfrak{T}} \\ (\Gamma; x_1 : T_1; \ldots; x_n : T_n \vdash ?_p : G[p/t]); \mathsf{induction}(t : T); (\Gamma \vdash ?_p : G); \mathcal{S}\end{array}}$$

$$\frac{\mathfrak{S} \Rightarrow_{\mathfrak{T}} (\Gamma' \vdash \emptyset); \mathsf{induction}(t : T); (\Gamma \vdash ?_p : G); \mathcal{S}}{\mathfrak{S}\ \texttt{end induction.} \Rightarrow_{\mathfrak{T}} (\Gamma \vdash \emptyset); \mathcal{S}}$$

4 Proof editing

4.1 Well-formedness

If we drop the justification ($j \triangleright \dots$) and completeness ($\Delta = \emptyset$) conditions in our formal semantics, we get a notion of well-formed proofs. Those proofs when run in Coq, are accepted with warnings but cannot be saved since the proof tree contains gaps.

This does not prevent the user from going further with the proof since the user is still able to use the result from the previous step. The smallest well formed proof is: `proof. end proof.`

Introduction steps such as `assume` have additional well-formedness requirements: the introduced hypotheses must match the available ones. The `given` construction allows a looser correspondence. The `reconsider` statements have to give a convertible type.

For proofs by induction, well-formedness requires the patterns to be of the correct type, and induction hypotheses to be build from the correct sub-objects in the pattern.

4.2 Formal proof sketches

We claim that well-formed but incomplete proofs in our language play the role of *formal proof sketches*: they ensure that hypotheses correspond to the current statement and that objects referred to exist and have a correct type. When avoiding the `by *` construction, justifications are preserved when adding extra commands inside the proof. In this sense our language supports incremental proof development.

The only thing that the user might have trouble doing when turning a textbook proof into a proof sketch in our language is ensuring that first-order objects are introduced before a statement refers to them. Once this is done, The user will be able to add new lines within blocks (mostly forward steps).

5 Conclusion

5.1 Further work

Arbitrary relation composition The first extension that is needed for our language is the support for iterated relations other than equality. This is possible as soon as a generalized transitivity lemma of the form $\forall xyz, x\, R_1\, y \rightarrow y\, R_2\, z \rightarrow x\, R_3\, z$ is available.

Better automation There is a need for a more precise and powerful automation for the default justification method, to be able to give better predictions of when a deduction step will be accepted. A specific need would be an extension of equality reasoning to arbitrary equivalence relations (setoids, PERs ...).

Multiple induction The support for induction is already quite powerful (support for deep patterns with multiple `as` bindings), but more can be done if we start considering multiple induction. It might be feasible to detect the induction scheme used (double induction, lexicographic induction ...) to build the corresponding proof on-the-fly.

Translation of procedural proofs The declarative language offers a stable format for the preservation of old proofs over time. Since many Coq proofs in procedural style already exist, it will be necessary to translate them to this new format. The translation can be done in two ways: by generating a declarative proof either from the proof tree, or from the proof term. The latter will be more fine grain but might miss some aspects of the original procedural proof. The former looks more difficult to implement.

5.2 Conclusion

The new declarative language is now widely distributed, though not yet widely used, and we hope that this paper will help new users to discover our language. The implementation is quite stable and the automation, although not very predictable, offers a reasonable compromise between speed and power.

We really hope that this language will be a useful medium to make proof assistant more popular, especially in the mathematicians community and among undergraduate students. We believe that our language provides a helpful implementation of the formal proof sketch concept; this means it could be a language of choice for turning textbook proofs into formal proofs. It could also become a tool of choice for education.

In the context of collaborative proof repositories (using the Wiki paradigm), our language, together with other declarative languages, will fill the gap between the narrow proof assistant community and the general public: we aim at presenting big formal proofs to the public.

References

[Coe06] Claudio Sacerdoti Coen. Automatic generation of declarative scripts. CHAT : Connecting Humans and Type Checkers, December 2006.

[Coq07] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.1*, February 2007.

[Cor03] Pierre Corbineau. First-order reasoning in the calculus of inductive constructions. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES : Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 162–177. Springer, 2003.

[Cor06] Pierre Corbineau. Deciding equaltiy in the constructor theory. In Thorsten Altenkirch and Conor McBride, editors, *TYPES : Types for Proofs and Programs*, Lecture Notes in Computer Science. Springer, 2006.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF.* Number 78 in Lecture Notes in Computer Science. Springer Verlag, 1979.

[GW04] Mariusz Giero and Freek Wiedijk. MMode, a mizar mode for the proof assistant coq. Technical report, ICIS, Radboud Universiteit Nijmegen, 2004.

[Har96] John Harrison. A mizar mode for HOL. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics, 9th International Conference, TPHOLs'96, Turku, Finland, August 26-30, 1996, Proceedings*, volume 1125 of *Lecture Notes in Computer Science*, pages 203–220. Springer, 1996.

[Har06] John Harrison. *The HOL Light manual*, 2006. Version 2.20.

[Lam95] Leslie Lamport. How to write a proof. *American Mathematics Monthly*, 102(7):600–608, 1995.

[Law94] Lawrence Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[Wen99] Markus Wenzel. Isar - A generic interpretative approach to readable formal proof documents. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, volume 1690 of *Lecture Notes in Computer Science*, pages 167–184. Springer, 1999.

[Wie01] Freek Wiedijk. Mizar light for HOL light. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001, Edinburgh, Scotland, UK, September 3-6, 2001, Proceedings*, volume 2152 of *Lecture Notes in Computer Science*, pages 378–394. Springer, 2001.

[Wie03] Freek Wiedijk. Formal proof sketches. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES : Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 378–393. Springer, 2003.

[WW02] Markus Wenzel and Freek Wiedijk. A comparison of mizar and isar. *Journal of Automated Reasoning*, 29(3-4):389–411, 2002.