Noname manuscript No.
(will be inserted by the editor)

Declarative Representation of Proof Terms

Claudio Sacerdoti Coen

Received: date / Accepted: date

Abstract We present a declarative language inspired by the pseudo-natural language previously used in Matita for the explanation of proof terms. We show how to compile the language to proof terms and how to automatically generate declarative scripts from proof terms. Then we investigate the relationship between the two translations, identifying the amount of proof structure preserved by compilation and re-generation of declarative scripts.

Keywords declarative language · proof terms · translation · generation

Mathematics Subject Classification (2000) 68W30 · 03B35 · 03B40

1 Introduction

In modern interactive theorem provers, proofs are likely to have several alternative representation inside the system. For instance, in Figure 1 we show the case of a system based on Curry-Howard implementation techniques: proofs could be input by the user in either a declarative or a procedural proof language; then the script could be interpreted and executed yielding a proof tree; from the proof tree we can generate a proof term; from the proof term, the proof tree or the initial script we can generate a description of the proof in a pseudo-natural language; finally, from the proof term, the proof tree or a declarative script we can generate a content level description of the proof, for instance in the OMDoc + MathML content language. For instance, the Coq proof assistant [1] has had in the past or still has all these representations but the last one; our Matita interactive theorem prover [2] also has all these representations but proof trees.

Partially supported by the Strategic Project DAMA (Dimostrazione Assistita per la Matematica e l'Apprendimento) of the University of Bologna.

C. Sacerdoti Coen
Dipartimento di Scienze dell'Informazione, via Mura Anteo Zamboni n. 7, 40127, Bologna (IT)
Tel.: +39-051-2094973
Fax: +39-051-2094510
E-mail: sacerdot@cs.unibo.it

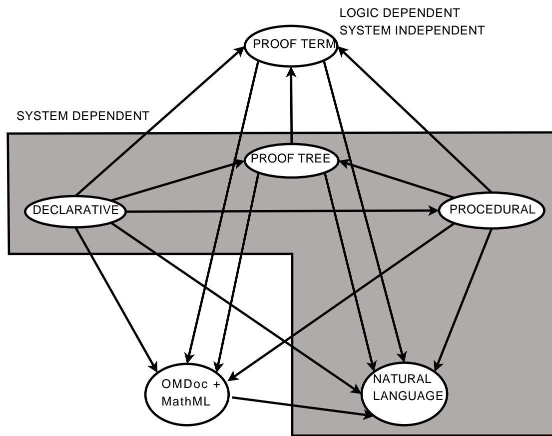

Fig. 1 Proof representations in Curry-Howard based interactive theorem provers. Arrows pointing upwards are compilation/interpretation processes. Arrows pointing downwards are serialisation/pretty-printing processes. The horizontal arrow between procedural and declarative scripts is a reconstruction of a declarative script from the effects of the procedural script on the state. The difference between the procedural language and the natural language is that the former is executable, while the latter is not.

It is then natural to investigate the translations between the different representations, wondering how much proof structure can be preserved in the translations. In [3] we started this study by observing that $\bar{\lambda}\mu\tilde{\mu}$-proof-terms are essentially isomorphic to the pseudo-natural language we proposed in the HELM and MoWGLI projects. In [4] we extended the result to OMDoc documents. At the same time we started investigating the possibility of giving an executable semantics to the grammatical constructions of our pseudo-language, obtaining the declarative language described in this paper. The language, which embeds an unelaborated justification sub-language, is currently in use in the Matita proof assistant.

In this paper we investigate the mutual translation between declarative scripts in this language and proof terms. We use λ-terms for a sub-language of the Calculus of (Co)Inductive Constructions (CIC) to keep the presentation simple but close to the actual implementation in Matita, which is not based on $\bar{\lambda}\mu\tilde{\mu}$-proof-terms.

Our main result is that the two translations preserve the proof structure and behave as inverse functions on declarative scripts generated by proof terms. Compilation and re-generation of a user-provided declarative script results in a script where the original proof steps and their order are preserved, and additional steps are added to make explicit all the justifications previously proved automatically. Misuses of declarative statements are also corrected by the process.

Translation of procedural scripts to declarative scripts can now be achieved for free by compiling procedural scripts to proof terms before generating the declarative scripts. In this case the proof structure is preserved only if it is preserved (by the semantics of tactic compilation) during the first translation.

In the companion paper [5] Ferruccio Guidi investigates the translation between proof terms of CIC and a subset of the procedural language of Matita. Several attempts at capturing the effects of procedural commands with declarative ones have been proposed in the past and are currently in use in the PhoX interactive theorem

prover [6]. Thus the picture about the different translations is now getting almost complete, up to the fact that the papers presented do not agree on the intermediate language used by most translations, which is the proof terms language.

An immediate application of this investigation, also explored in [5], is the possibility to take a proof script from a proof assistant (say Coq), compile it to proof terms, transmit them to another proof assistant (say Matita) based on the same logic and rebuild from them either a declarative or a procedural proof script that is easier to manipulate and to be evolved. A preliminary experiment in this sense is also presented in the already cited paper.

The requirement for the translations investigated in this paper are presented in Section 2. Then in Section 3 we present the syntax and the informal semantics of our declarative proof language. Compared with other state of the art declarative languages such as Isar [7] and Mizar [8], a minor attention has been given to the (sub-)language for justification of proof steps. Right now justifications can be completely omitted (and provided by automation) or they can be hints to automation — like the set of hypotheses to be used or parameters to prune the search space — or they are proof terms.

In Section 4 we show the small steps operational semantics of the language which, scripts being sequences of statements, is naturally unstructured in the spirit of [9]. The semantics of a statement is a function from partial proof terms to partial proof terms, i.e. a procedural tactic. Thus the semantics of a declarative script is a compilation to proof terms mediated by tactics in the spirit of [10].

In Section 5 we show the inverse of compilation, i.e. the automatic generation of a declarative script from a proof term. We prove that the two translations form a retraction pair and that their composition is idempotent.

2 Requirements

In this paper we explain how to translate declarative scripts into proof terms and back. By going through proof terms, procedural scripts can also be translated to declarative scripts. Before addressing the details of the translations, we consider here their informal requirements. We classify the requirements according to two interesting scenarios we would like to address.

Re-generation of declarative scripts from declarative scripts (via proof terms). In this scenario a declarative script is executed obtaining a proof term that is then translated back to a declarative script. The composed translation should preserve the structure of the user provided text, but can make more details explicit. For instance, it can interpolate a proof step between two user provided proof steps or it can add an omitted justification for a proof step. The translation must also reach a fix-point in one iteration. The latter requirement is a consequence of the following stronger requirement: the proof term generated executing the obtained declarative script should be exactly the same proof term used to generate the declarative script. In other terms, the composed translation should not alter the proof term in any way and can only reveal hidden details.

Re-generation of declarative scripts from procedural scripts (via proof terms). In this scenario a procedural script is executed obtaining a proof term that is then translated

back to a declarative script. Ideally the two scripts should be equally easy to modify and maintain. Moreover, the "structure" of the procedural script (if any) should be preserved. Pedantic details or unnecessary complex sub-proofs that are not explicit in the procedural proof should be hidden in the declarative one. This last requirement is not really a constraint on the declarative language, but on the implementation of the tactics of the proof assistant [11].

Some of the requirements, in particular the preservation of the structure of the user provided text, seem quite difficult to obtain. In [3] we claimed that the latter requirement is likely to be impossible to fulfil when proof terms are Curry-Howard isomorphic to natural deduction proof trees, i.e. when proof terms are simple λ-terms. On the contrary, we expect to be able to fulfil the requirements if proof terms are Curry-Howard isomorphic to sequent calculus. This is the case, for instance, for the $\bar{\lambda}\mu\tilde{\mu}$-terms [12] we investigated as proof terms in [3,4]. In particular, automatic structure preserving generation of Mizar/Isar procedural scripts from $\bar{\lambda}\mu\tilde{\mu}$-terms have been attempted in the Fellowship theorem prover [13] (joint work with Florent Kirchner, for more informations see [14]).

Matita proof terms are λ-terms of CIC. The calculus is so rich that several of the required constructs of the $\bar{\lambda}\mu\tilde{\mu}$-calculus are somehow available. Thus we expect to be able to fulfil at least partially the requirements just presented. Even in case of failure it is interesting to understand exactly how close we can get.

In the present paper we restrict ourselves to a fragment of CIC, although the implementation in Matita considers the whole calculus. The fragment is an extension of the first order fragment of CIC where we also keep explicit type conversions, local definitions and local proofs. We will present the proof terms for this fragment in Section 6.

3 The declarative language

The syntax of the declarative language we propose is an adaptation of the syntax of the pseudo natural language already generated by Matita and studied in [15]. It is also a super-set of the language proposed in [3] and studied also in [4]. The sub-language for justifications is unelaborated. Thus currently a justification is either provided as a proof term or it is omitted and recovered by automation. In the latter case, it is possible to specify hints to the automation, like the only lemmas and hypotheses to be used. A comparison with other declarative languages, which would be out of scope for this paper, is currently planned.

We have explicit statements that deal with conversion, a feature of the logical framework of Matita that is not available in first and higher order logics. Two formulae are convertible when they can be reduced by computation to a common value. For instance, $2 * 2$ is convertible with $3 + 1$. Since conversion is a decidable property (in a confluent and strongly normalisable calculus), conversion and reduction steps are not recorded in the proof term (e.g. as rewriting steps). However, since conversion steps are not always obvious to the reader, it is sometimes necessary to make them explicit in the declarative language. Thus the need for the additional statements. In Isar the same steps would be represented by (chains of) rewriting steps since Isabelle's meta-logic does not have any conversion rule, only a primitive notion of equality and rewriting rules.

Table 1 Syntax

assume id **:** type [**that is equivalent to** type]
suppose prop [(id)] [**that is equivalent to** prop]
let id **:=** term
just **we proved** prop (id) [**that is equivalent to** prop]
just **we proved** prop [**that is equivalent to** prop] **done**
just **done**
just **let** id **:** type **such that** prop (id)
just **we have** prop (id) **and** prop (id)
we need to prove prop [(id)] [**or equivalently** prop]
we proceed by [**cases**|**induction**] **on** term **to prove** prop
case id [(id:type)|(id:prop)]*
by induction hypothesis we know prop (id) [**that is equivalent to** prop]
the thesis becomes prop [**or equivalently** prop]
conclude term rel term just [**done**]
obtain id term rel term just [**done**]
rel term just [**done**]

Justifications:
using proof_term
[control_param] [**by** proof_term$_1$ [, ..., proof_term$_n$]]

Non terminals:

id	identifiers	term	inhabitants of data types
type	data types	rel	transitive relations (e.g. $=, \leq, <$)
prop	propositions	proof_term	proof terms, e.g. an identifier
control_param	for automation		

Table 2 An example of declarative script

we need to prove $\forall R, S : \mathbb{N} \to \mathbb{N}.\langle R, S\rangle$ is a retraction pair $\Rightarrow \forall n : \mathbb{N}.S(n) = (S \circ R \circ S)(n)$
assume $R : \mathbb{N} \to \mathbb{N}$
assume $S : \mathbb{N} \to \mathbb{N}$
suppose $\langle R, S\rangle$ is a retraction pair (H1) **that is equivalent to** $\forall m : \mathbb{N}.S(R(m)) = m$
assume $n : \mathbb{N}$
let $t := S(n)$
by H1 **we proved** $t = S(t)(H3)$ **that is equivalent to** $S(n) = (S \circ R \circ S)(n)$
by H3
done

We now present informally the semantics of the proposed language statements, whose syntax is summarised in Table 1. Table 2 and Table 3 show two examples of declarative scripts where most commands are used.

assume *id* **:** *type1* [**that is equivalent to** *type2*]
Introduces in the context a new generic but fixed term *id* whose type is *type1*. If specified, *type2* must be convertible to *type1*. In this case *id* will be used later on with type *type2*, but in the conclusion of the proof *type1* will be used.

suppose *prop1* [(*id*)] [**that is equivalent to** *prop2*]
Introduces in the context the hypothesis *prop1* labelled by *id*. If the proposition *prop2* is specified, it must be convertible with *prop1*. In this case *id* will stand later on for the hypothesis *prop2*, but in the conclusion of the proof *prop1* will be used.

let *id* **:=** *term*
Introduces in the context a new local definition.

Table 3 Another example of declarative script

we need to prove $\forall n : \mathbb{N}. n + n = n * 2$
  assume $n : \mathbb{N}$
  we proceed by induction on n **to prove** $n + n = n * 2$
    case O
      the thesis becomes $0 + 0 = 0 * 2$ **or equivalently** $0 = 0$
    done
    case S (m : $\mathbb{N}$)
      by induction hypothesis we know $m + m = m * 2$ (IH)
      the thesis becomes $S(m) + S(m) = S(m) * 2$
        or equivalently $S(m + S(m)) = 2 + m * 2$
      conclude
          $S(m + S(m))$
        $= S(S(m + m))$ **by** plus_n_Sm
        $= S(S(m * 2))$ **by** IH
        $= 2 + m * 2$
    done

just **we proved** *prop1* **(***id***)** [**that is equivalent to** *prop2*]
Concludes the proposition *prop1* by means of the justification *just*. The (proof of the) proposition is labelled by *id* for further reference. If *prop2* is specified, it must be convertible with *prop1*. In this case *id* will stand for a proof of the proposition *prop2*.

just **we proved** *prop1* [**that is equivalent to** *prop2*] **done**
Similar to the previous statement. However, the conclusion *prop1* (or *prop2* if specified and convertible with *prop1*) is the current thesis. Thus this statement ends the innermost sub-proof.

just **done**
Similar to the previous statement. However, the conclusion, equal to the current thesis, is not repeated.

just **let** *id1* **:** *type* **such that** *prop* **(***id2***)**
Concludes the proposition $\exists id1 : type$ s.t. *prop* by means of the justification *just*. Exist-elimination is immediately performed yielding the new generic but fixed term *id1* of type *type* and the new hypothesis *prop* labelled by *id2*.

just **we have** *prop1* **(***id1***) and** *prop2* **(***id2***)**
Concludes the proposition $prop1 \wedge prop2$ by means of the justification *just*. And-elimination is immediately performed yielding the new hypotheses *prop1* and *prop2* labelled respectively by *id1* and *id2*.

we need to prove *prop1* [**(***id***)**] [**or equivalently** *prop2*]
If *id* is omitted, it repeats the current thesis *prop1*. Moreover, if *prop2* is specified and convertible with *prop1*, it replaces the current thesis with *prop2*. Otherwise, if *id* is specified, it starts a nested sub-proof of *prop1* that will be labelled by *id*. If *prop2* is specified and convertible with *prop1*, the thesis of the nested sub-proof is *prop2*, but *id* will label *prop1*.

we proceed by [**cases**|**induction**] **on** *term* **to prove** *prop*
case *id1* [**(***id2***:***type2***)**|**(***id2***:***prop***)**]*
by induction hypothesis we know *prop1* **(***id***)** [**that is equiv. to** *prop2*]
the thesis becomes *prop1* [**or equivalently** *prop2*]
This set of statements are used for proofs by structural induction or by case analysis. The initial statement must be followed by a proof for each case. Each proof must be started by the **case** *id* statement, where *id* is the label of the case (i.e. the name of

the inductive constructor the case refers to). The list of arguments that follows *id* binds the local non inductive assumptions for the case. The inductive assumptions are postponed and introduced by the next statement in the set. Only proofs by inductions have inductive assumptions. The last statement in the set, **the thesis becomes**, is used to state explicitly what is the current thesis for each proof.

conclude *term1 rel term2 just* [**done**]
obtain *id term rel term just* [**done**]
rel term just [**done**]

This set of statements are used for chains of (in)equalities. A chain is started by either the first or the second command in the set. All the remaining steps in the chain are made by the third command. In all commands *rel* must be a transitive relation. Chains with mixed relations are possible as soon as the different relations enjoy generalised transitivity (e.g. $x \leq y \wedge y < z \Rightarrow x < z$). Every step in the chain must have a justification *just*. The end of the chain is marked by **done**. In every step but the first one the left hand side of the inequation is the right hand side of the previous step.

If the first step of the chain is a **conclude** statement, then the chain must prove the current thesis, and the last step of the chain ends the innermost sub-proof. Otherwise, if the first step of the chain is a **conclude** statement, the chain only proves a local lemma that is labelled by *id* in the rest of the innermost sub-proof. For instance, in the following example H is in scope after **done** and labels the fact $(x+y)^2 = x^2 + 2xy + y^2$:

obtain H

$$\begin{aligned}(x+y)^2 &= (x+y)(x+y) \\ &= x(x+y) + y(x+y) \text{ by } \textit{distributivity} \\ &= x^2 + xy + yx + y^2 \quad \text{by } \textit{distributivity} \\ &= x^2 + 2xy + y^2\end{aligned}$$

done

Justifications of the form "**using** t" represent the direct application of the proof term t. When t has the form "$(H\ E_1\ \ldots\ E_n\ H_1\ \ldots\ H_m)$" we say that the justification "**using** t" is *simple*. All other justifications are invocations of automation. In particular, the user can specify some control parameters to drive automation and he can specify the list of proof terms to be used (usually hypotheses and lemmas in the library or their instantiation). When the list is omitted, the system tries all hypotheses and all the lemmas in the files that have been explicitly required so far by the user. However, there is a control parameter to extend the search space with all the lemmas already proved in the distributed mathematical library of the system. Finally, there is the possibility to open an interactive interface [16] to show and prune the search space and, more generally, to drive the proof search process.

4 Formal semantics

The semantics of each statement of Table 1 is a function from a partial proof term to a partial proof term. Intuitively, a partial proof term is a proof term with linear placeholders for missing sub-proofs and non-linear placeholders for missing sub-expressions. Each placeholder must be replaced with a proof term or an expression, of the appropriate type, closed in the logical context of the placeholder. The logical context of

Table 4 Proof term syntax

Types	
$T ::= T \to T$	function space
$\mid$ nat	basic type
$\mid \ldots$	other type constructors
Propositions	
$P ::= P \Rightarrow P$	logical implication
$\mid \forall x : T.P$	universal quantification
$\mid \exists x : T.P$	existential quantification
$\mid P \wedge P$	conjunction
$\mid E = E$	equality
$\mid F(E_1, \ldots, E_n)$	n-ary predicate
Expressions (inhabitants of types)	
$E ::= x$	bound variable ranging over expressions
$\mid k$	constants
$\mid E(E_1, \ldots, E_n)$	n-ary application
$\mid ?$	expression placeholder
Proof terms (inhabitants of propositions)	
$t ::= \lambda x : T.t$	local assumption (for an universal quantification)
$\mid \Lambda H : P.t$	local supposition (for a logical implication)
$\mid \text{let } x := E \text{ in } t$	local definition
$\mid \text{Let } H : P := t \text{ in } t$	logical cut
$\mid (t : P \equiv P)$	explicit type conversion
$\mid (H\ E_1\ \ldots\ E_n\ H_1 \ldots\ H_m)$	application of a bound variable ranging over proof terms to 0 or more arguments
$\mid (c\ E_1\ \ldots\ E_n\ a_1 \ldots\ a_m)$	application of a constant to 0 or more arguments;
$a ::= (H\ E_1\ \ldots\ E_n\ H_1 \ldots\ H_m)$	provided that the type of the application is not a logical implication or an universal quantification, i.e. the argument is in head long $\beta\eta$ normal form
$\mid (c\ E_1\ \ldots\ E_n\ a_1 \ldots\ a_m)$	provided that the argument is in head long $\beta\eta$ normal form
$\mid \lambda x : T.a$	local assumption (for an universal quantification)
$\mid \Lambda H : P.a$	local supposition (for a logical implication)
$c ::= \text{and_elim}_{P,P,P}$	conjunction elimination
$\mid$ ex_elim	existential elimination
$\mid \text{nat_ind}_P$	induction over Peano natural numbers
$\mid \text{nat_cases}_P$	case analysis over Peano natural numbers
$\mid$ eq_transitive	transitivity of equality
$\mid \ldots$	other constants

the placeholder is the ordered set of hypothesis, definitions and declarations collected navigating the proof term from the root to the placeholder. A partial proof term is complete (i.e. it represents a completed proof) when it is placeholder-free. When a proof is started, it is represented by the partial proof term made of just one term placeholder.

Term placeholders occur only linearly in a partial proof term and, since our language has no dependent types, a term placeholder never occurs in an expression. Thus, we are not obliged to explicitly introduce term placeholders in the formal syntax: instead we can just represent a partial proof term as a function from a tuple of proof terms to

Table 5 Proof term typing rules (standard well-formed conditions on expressions, contexts, and types omitted)

Proof term typing rules (also valid for arguments).

$$\frac{\begin{array}{c}\Gamma \vdash t : \forall x_1 : T_1. \dots . \forall x_n : T_n . P_1 \Rightarrow \dots \Rightarrow P_m \Rightarrow P \\ \Gamma \vdash E_i : T_i\{E_1/x_1 \; ; \; \dots \; ; \; E_{i-1}/x_{i-1}\} \;\; \forall i \in \{1, \dots, n\} \\ \Gamma \vdash t_i : P_i\{E_1/x_1 \; ; \; \dots \; ; \; E_n/x_n\} \;\; \forall i \in \{1, \dots, m\}\end{array}}{\Gamma \vdash (t \; E_1 \; \dots \; E_n \; t_1 \; \dots \; t_m) : P}$$

$$\frac{\Gamma, x : T \vdash t : P}{\Gamma \vdash \lambda x : T.t : \forall x : T.P} \qquad \frac{\Gamma, H : P_1 \vdash t : P_2}{\Gamma \vdash \Lambda H : P_1.t : P_1 \Rightarrow P_2} \qquad \frac{\Gamma \vdash t : P_1 \qquad \Gamma \vdash P_1 \equiv P_2}{\Gamma \vdash (t : P_1 \equiv P_2) : P_2}$$

$$\frac{\Gamma, x := E \vdash t : P}{\Gamma \vdash \text{let } x := E \text{ in } t : P\{E/x\}} \qquad \frac{\Gamma \vdash t_1 : P_1 \qquad \Gamma, H : P_1 \vdash t_2 : P_2}{\Gamma \vdash \text{Let } H : P_1 := t_1 \text{ in } t_2 : P_2}$$

We now show the formal semantics of our language in terms of compilation of a declarative script to a proof term. In Tables 4 and 5 we show the syntax and typing rules for the proof terms we will use to encode first order logic natural deduction trees. We only show the inference rules for proof terms, omitting all the conditions about the well-formedness of contexts, types and propositions occurring in the inference rules, since they are quite standard and not relevant to the present work. Moreover we restrict induction and case analysis to natural numbers and we only consider chains of equalities over natural numbers.
In the definition of the proof term syntax we use the non standard notion of arguments in head long $\beta\eta$-normal form, defined as follows: an argument is in head long $\beta\eta$-normal form when either it is not an application or its type is neither a logical implication nor a universal quantification. Note that this definition is purely based on the syntax and a restricted set of typing rules. In particular, we do not need to assume any notion of $\beta\eta$-reduction on proof terms. Nevertheless, we call it head long $\beta\eta$-normal form since, in presence of $\beta\eta$-reduction rules on proof terms, it restricts terms to be in head β-normal form and to be be non-recursively η-expanded.
We also assume at least the following constant schemes (that are always supposed to be applied to arguments in head long $\beta\eta$ normal form):

$$\begin{array}{ll}\text{and_elim}_{P_1,P_2,P_3} : & P_1 \wedge P_2 \Rightarrow (P_1 \Rightarrow P_2 \Rightarrow P_3) \Rightarrow P_3 \\ \text{ex_elim}_{T,P_1,P_2} : & (\exists x : T.P_1) \Rightarrow (\forall x : T.P_1 \Rightarrow P_2) \Rightarrow P_2 \\ \text{nat_ind}_P : & \forall n : \text{nat}.P(0) \Rightarrow (\forall m : \text{nat}.P(m) \rightarrow P(S(m))) \Rightarrow P(n) \\ \text{nat_cases}_P : & \forall n : \text{nat}.P(0) \Rightarrow (\forall m : \text{nat}.P(m)) \Rightarrow P(n) \\ \text{eq_transitive} : & \forall x, y, z : \text{nat}.x = y \Rightarrow y = z \Rightarrow x = z\end{array}$$

a proof term. Every formal parameter in the tuple corresponds to a term placeholder. On the other hand, we have introduced in Table 4 the explicit syntax “?” for expression placeholders. From now on, if not stated otherwise, a placeholder is always an expression placeholder. Moreover, the sequent (i.e. the pair context/type) associated to a proof term placeholder will be called *goal*.

Formally, we represent a partial proof term as a triple (Σ, Σ', Π). Σ is an ordered list of sequents $\Gamma \vdash P$ providing context and type for the proof term placeholders occurring in the partial proof. Σ' does the same for expression placeholders. Π, the actual partial proof term, is a function from “fillings” for both kinds of placeholders to placeholder-free proof terms.

partial_proof :=
  (context $*$ proposition) list $*$
  (context $*$ type) list $*$
  (proof_term list $*$ expression list $\rightarrow$ proof_term)

We denote the empty list with $[]$, the concatenation of two lists with $l_1@l_2$ and the insertion of an element at the beginning of a list with $x :: l$. With $(l,l') \mapsto t$ we denote an anonymous function from pairs of lists to terms. In particular, (l,l') is a pattern that binds both l and l' in t. With $C[l,l']$ we represent a proof term having all the proof-terms in l and all the expressions in l' as sub-terms. Finally, π_3 is the third projection of a tuple.

We now look at some examples of partial proofs.

Example 1 (Initial partial proof) Let P be a statement (a proposition). In order to prove P, the following initial partial proof is considered: $([\vdash P], [], ([H], []) \mapsto H)$. The list of open goals presented to the user is the singleton list $[\vdash P]$, i.e. the user must prove P in the empty context. The list of placeholders is empty. The function $([H], []) \mapsto H$ must be applied to the singleton list $[t]$ of inhabitants of $[\vdash P]$ and to the empty list of instantiations for $[]$; once applied, it reduces to t which is the closed proof term that inhabits the statement of the theorem. Indeed, t has type P.

Example 2 (Closed partial proof) A closed partial proof is a triple $([], [], ([], []) \mapsto t)$. Since the list of open goals is empty, there is nothing left to prove. Thus the function $([], []) \mapsto t$ must be applied to $([], [])$ to compute the closed proof term t that inhabits the statement of the theorem.

Example 3 (General partial proof) The partial proof $([\vdash 2 > 0;\quad (x : \mathbb{N}; H : x > 0; \vdash x \neq 0)], [], ([H1; H2], []) \mapsto \text{let } x := 2 \text{ in Let } H := H1 \text{ in } H2)$ represents a situation where there are two goals to be proved: the first one requires a proof of $2 > 0$ in the empty context; the second one requires a proof of $x \neq 0$ under the assumption $x : \mathbb{N}$ and the supposition $x > 0$ (inhabited by H). Let t_1 be a closed proof term for the first goal and t_2 a proof term for $x \neq 0$ where x and H may freely occur. The application of the function to $([t_1, t_2], [])$ is the closed proof term let $x := 2$ in Let $H := t_1$ in t_2. Note that the free occurrences of x and H in t_2 are now bound in the final proof term.

Example 4 (Partial proof with a placeholder) The partial proof $([x : \mathbb{N}; H :? > 1 \vdash ? > 0], [x : \mathbb{N} \vdash \mathbb{N}], ([H2], [E]) \mapsto \lambda x : \mathbb{N}.\lambda H1 : E > 1.H2)$ represents a situation where there is one unknown term of type $\mathbb{N}$ which can be instantiated with any term whose only free variables are in $\{x\}$. Moreover, there is one open goal that requires to prove positivity of the unknown term, represented by the placeholder ?, under the assumption $? > 1$. For any instantiation e of the placeholder and any proof term t for the goal instantiated over e, the function returns $\lambda x : \mathbb{N}.\lambda H1 : e > 1.t$ which is a proof term for $\forall x : \mathbb{N}.e > 1 \Rightarrow e > 0$.

Not every element of the partial proof data type actually corresponds to a well formed proof in progress. To define a typing judgement for partial proofs it is sufficient to require: that every sequent in the list (with at most one element) of placeholders declarations is well formed; that every sequent in the list of goals is well typed under the assumption that any occurrence of ? has the declared type and is put in a context compatible with the declared one; that the function expects two lists whose lengths are equal to those of the lists of goals and placeholders; that the function produces only well typed proof terms under the assumption that the i-th element of the list of goals (placeholders) has the declared type; that every occurrence of the i-th element of the list of goals (placeholders) occurs in a context compatible with the declared one. Two contexts are compatible when every variable (assumption) declared/defined in the first context is also declared/defined in the same way in the second context.

We do not give here the formal judgements corresponding to the previous conditions and, similarly, we omit from the paper the typing rules for expressions, types and formulae and the reduction rules for formulae and terms. As a consequence, we omit as well all the meta-theory of the given calculus. We believe that being more rigorous by better fixing the calculus is unnecessary to understand the ideas presented here, that apply to a broad spectrum of calculi. Indeed, only the typing rules for proof terms matter in the remaining of the paper. The implementation we provide in Matita considers the whole CIC without any major difference from what is presented here, but for the complication of having to detect in advance which terms are proof-terms and which terms are expressions. This is achieved with an enhanced version of Coscoy's double type inference algorithm presented in [17].

The semantic function $\mathcal{C}[\![\cdot]\!]$ shown in Table 6 maps statements to partial functions from partial proof terms to partial proof terms. Typically, the head goal is removed from the list of goals in input and zero or more goals are added to the list of goals in output. The new function that builds the final proof term takes in input the proof terms for the newly generated goals and for the goals that are just propagated by the command; then it calls the old function passing for the first argument — the proof term for the removed goal — a proof term built from those for the newly generated goals; the remaining arguments are just propagated. In simpler words, the function is responsible for building an evidence for the removed goal from the evidences for the new goals.

The output of the semantic function is a partial function since it may be the case that a command is erroneously applied to a goal that does not have the expected shape. For instance, in order to assume a variable, the goal must be a universal quantification. We appreciate this kind of strictness when the declarative language is used in education to teach logic to first year students, like we are currently doing at the University of Bologna. On the other hand, it is common mathematical practice to be more liberal in this respect. Isar tries to adhere to this practice by allowing users to prove something different from what is stated in the main or local proofs. Lightweight automation is then applied to conclude the original goal. This can be easily accommodated in the proposed framework and we leave it as a future extension.

Since we are not interested in the way automation finds justifications, we assume the existence of a (partial) function $\mathcal{A}[\![\cdot]\!]$ that, given a justification and a proposition, returns a proof term that inhabits the proposition.

$$\mathcal{A}[\![\cdot]\!] : \text{justification} * \text{proposition} \to \text{proof_term}$$

However, we must impose the following requirement: for each proof term t that inhabits P we ask $\mathcal{A}[\![\textbf{using}\ t, P]\!] = t$. Moreover, the grammar of proof terms already constrains automation. For instance, an automatically found proof of the argument of a constant must be in the particular head long $\beta\eta$ normal form of Table 1. If no proof in such form can be found, automation must fail.

$\mathcal{C}[\![\cdot]\!]^\star$ extends the semantics to a list of statements (a declarative script). Given a declarative script $S_1 \ \cdots \ S_n$, the proof term generated executing the script $\mathcal{S}$ from the initial proof state for a proposition P is given by $\mathcal{C}[\![\cdot]\!]_s$ applied to $(\mathcal{S}, P)$. More

rigorously:

$$\mathcal{C}[\![\cdot]\!] : \text{statement} \rightarrow \text{partial_proof} \rightarrow \text{partial_proof}$$

$$\mathcal{C}[\![\cdot]\!]^\star : \text{statement list} \rightarrow \text{partial_proof} \rightarrow \text{partial_proof}$$
$$\mathcal{C}[\![S_1 \cdots S_n]\!]^\star = \mathcal{C}[\![S_n]\!] \circ \cdots \circ \mathcal{C}[\![S_1]\!]$$

$$\mathcal{C}[\![\cdot]\!]_s : \text{statement list} * \text{proposition} \rightarrow \text{proof_term}$$
$$\mathcal{C}[\![S_1 \cdots S_n, P]\!]_s = \pi_3(\mathcal{C}[\![S_1 \ \cdots \ S_n]\!]^\star \ ([\vdash P], [], ([H], []) \mapsto H)) \ ([], [])$$

Example 5 Consider the statement $\forall x : \text{nat}.P(x)$ and a script "**assume** x : nat $\mathcal{S}$" where we suppose that $\mathcal{S}$ produces for the sequent $x : \text{nat} \vdash P(x)$ a proof term t (i.e. that $\mathcal{C}[\![\mathcal{S}]\!]^\star([x : \text{nat} \vdash P(x)], [], \Pi) = ([], [], ([], []) \mapsto \Pi([t], [])))$

We have:

$$\begin{aligned}
&\mathcal{C}[\![\textbf{assume } x : \text{nat } \mathcal{S}, \forall x : \text{nat}.P(x)]\!]_s \\
&= \pi_3((\mathcal{C}[\![\mathcal{S}]\!]^\star \circ \mathcal{C}[\![\textbf{assume } x : T]\!]) \ ([\vdash \forall x : \text{nat}.P(x)], [], (H, []) \mapsto H)) \ ([], []) \\
&= \pi_3(\mathcal{C}[\![\mathcal{S}]\!]^\star([x : \text{nat} \vdash P(x)], [], ([hd], []) \mapsto \lambda x : \text{nat}.hd)) \ ([], []) \\
&= \pi_3(([], [], ([], []) \mapsto \lambda x : \text{nat}.t)) \ ([], []) \\
&= \lambda x : \text{nat}.t
\end{aligned}$$

We immediately notice from the rules in the table that **assume** generates a λ-abstraction, **suppose** a Λ-abstraction and **let** a let ... in definition. Moreover, all commands that prove a sub-result, i.e. **we proved** and **obtain**, generate a logical cut Let ... in. The commands that end with **done** are those that close the head goal without opening new ones. Most commands have an alternative form to also handle conversion, that is explicitly recorded in the proof term. Most of the remaining commands are syntactic sugar for the application of elimination principles or transitivity principles. Finally, notice that some commands, like **the thesis becomes** and **we need to prove** are perfect synonyms meant to be used in different contexts.

The translation of the **case** command is only partial, since the name H of the case is ignored. Thus the user must remember to prove the base case and the inductive case exactly in this order. In a realistic implementation, the system should detect if the two cases are swapped and either complain or, even better, match the second proof term and goal instead of the first one when the inductive case is addressed first. This is easily achievable by labelling each sequent in the list of sequents with a string that can be, for instance, O in the base case and S in the inductive case. We did not do that in the formal semantics to keep it simple.

Only the command **obtain** H $E_1 = E_2$ introduces in Σ' a new expression placeholder ?. The placeholder ? stands for the right hand side of the last expression of the chain. Its instantiation will be known only in the last step of the chain, i.e. in the next $= E_3$ **done** statement. Since equation chains cannot be nested, in a partial proof term there can be at most one placeholder, i.e. Σ' can have at most one element and one placeholder symbol ? is sufficient. A simple extension consists in introducing non-linear, numbered placeholders $?_i$ in order to introduce additional commands that leave some part of a local thesis unspecified and that can be freely nested. The current semantics has already been given with this extension in mind, so that no rules would need to be changed but the typing rules for sequents that have been omitted.

Table 6: Formal semantics

$\mathcal{C}[\![\textbf{assume } x : T]\!](\Gamma \vdash \forall x : T.P :: \Sigma, \Sigma', \Pi) =$
$\quad ((\Gamma\ ;\ x : T \vdash P) :: \Sigma, \Sigma', (hd :: tl, l) \mapsto \Pi((\lambda x : T.hd) :: tl, l))$
$\mathcal{C}[\![\textbf{assume } x : T_1 \textbf{ that is equivalent to } T_2]\!](\Gamma \vdash \forall x : T_1.P :: \Sigma, \Sigma', \Pi) =$
$\quad ((\Gamma\ ;\ x : T_2 \vdash P) :: \Sigma, \Sigma',$
$$\begin{cases} (hd : P' \equiv P) :: tl, l) \mapsto \Pi(((\lambda x : T_2.hd) : (\forall x : T_2.P') \equiv (\forall x : T_1.P)) :: tl, l) \\ (hd :: tl, l) \qquad\qquad \mapsto \Pi(((\lambda x : T_2.hd) : (\forall x : T_2.P) \equiv (\forall x : T_1.P)) :: tl, l) \end{cases}$$

$\mathcal{C}[\![\textbf{suppose } P_1\ (H)]\!](\Gamma \vdash \forall P : P_1.P_2 :: \Sigma, \Sigma', \Pi) =$
$\quad ((\Gamma\ ;\ H : P_1 \vdash P_2) :: \Sigma, \Sigma', (hd :: tl, l) \mapsto \Pi((\Lambda H : P_1.hd) :: tl, l))$
$\mathcal{C}[\![\textbf{suppose } P_1\ (H) \textbf{ that is equivalent to } P_2]\!](\Gamma \vdash \forall H : P_1.P :: \Sigma, \Sigma', \Pi) =$
$\quad ((\Gamma\ ;\ H : P_2 \vdash P) :: \Sigma, \Sigma',$
$$\begin{cases} ((hd : P_3 \equiv P) :: tl, l) \mapsto \Pi(((\Lambda H : P_2.hd) : (P_2 \to P_3) \equiv (P_1 \to P)) :: tl, l) \\ (hd :: tl, l) \qquad\qquad \mapsto \Pi(((\Lambda H : P_2.hd) : (P_2 \to P) \equiv (P_1 \to P)) :: tl, l) \end{cases}$$

$\mathcal{C}[\![\textbf{let } x := E]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) =$
$\quad ((\Gamma\ ;\ x := E \vdash P) :: \Sigma, \Sigma', (hd :: tl, l) \mapsto \Pi((\text{let } x := E \text{ in } hd) :: tl, l))$

$\mathcal{C}[\![j \textbf{ we proved } P_1\ (H)]\!](\Gamma \vdash P_2 :: \Sigma, \Sigma', \Pi) =$
$\quad ((\Gamma\ ;\ H : P_1 \vdash P_2) :: \Sigma, \Sigma', (hd :: tl, l) \mapsto \Pi((\text{Let } H : P_1 := \mathcal{A}[\![j, P_1]\!] \text{ in } hd) :: tl, l))$
$\mathcal{C}[\![j \textbf{ we proved } P_1\ (H) \textbf{ that is equivalent to } P_2]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) =$
$\quad ((\Gamma\ ;\ H : P_2 \vdash P) :: \Sigma, \Sigma',$
$\qquad (hd :: tl, l) \mapsto \Pi((\text{Let } H : P_1 := (\mathcal{A}[\![j, P_1]\!] : P_1 \equiv P_2) \text{ in } hd) :: tl, l))$
$\mathcal{C}[\![j \textbf{ we proved } P \textbf{ done}]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) =$
$\quad (\Sigma, \Sigma', (tl, l) \mapsto \Pi(\mathcal{A}[\![j, P]\!] :: tl, l))$
$\mathcal{C}[\![j \textbf{ we proved } P_1 \textbf{ that is equivalent to } P_2 \textbf{ done}]\!](\Gamma \vdash P_2 :: \Sigma, \Sigma', \Pi) =$
$\quad (\Sigma, \Sigma', (tl, l) \mapsto \Pi((\mathcal{A}[\![j, P_1]\!] : P_1 \equiv P_2)) :: tl, l))$
$\mathcal{C}[\![j \textbf{ done}]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) = (\Sigma, \Sigma', (tl, l) \mapsto \Pi(\mathcal{A}[\![j, P]\!] :: tl, l))$

$\mathcal{C}[\![j \textbf{ let } x : T \textbf{ such that } P_1\ (H)]\!](\Gamma \vdash P_2 :: \Sigma, \Sigma', \Pi) =$
$\quad ((\Gamma\ ;\ x : T\ ;\ H : P_1 \vdash P_2) :: \Sigma, \Sigma',$
$\qquad (hd :: tl, l) \mapsto \Pi((\text{ex_elim}_{T,P_1,P_2} \mathcal{A}[\![j, \exists x : T.P_1]\!]\ (\lambda x : T.\Lambda H : P_1.hd))) :: tl, l))$
$\mathcal{C}[\![j \textbf{ we have } P_1\ (H_1) \textbf{ and } P_2\ (H_2)]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) =$
$\quad ((\Gamma\ ;\ H_1 : P_1\ ;\ H_2 : P_2 \vdash P) :: \Sigma, \Sigma',$
$\qquad (hd :: tl, l) \mapsto \Pi((\text{and_elim}_{P_1,P_2,P}\ \mathcal{A}[\![j, P_1 \wedge P_2]\!]\ (\Lambda H_1 : P_1.\Lambda H_2 : P_2.hd))) :: tl, l))$

$\mathcal{C}[\![\textbf{we need to prove } P]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) =$
$\quad ((\Gamma \vdash P) :: \Sigma, \Sigma', (hd :: tl, l) \mapsto \Pi((hd) :: tl, l))$
$\mathcal{C}[\![\textbf{we need to prove } P_1 \textbf{ or equivalently } P_2]\!](\Gamma \vdash P_1 :: \Sigma, \Sigma', \Pi) =$
$\quad ((\Gamma \vdash P_2) :: \Sigma, \Sigma', (hd :: tl, l) \mapsto \Pi((hd : P_2 \equiv P_1)) :: tl, l))$
$\mathcal{C}[\![\textbf{we need to prove } P_1\ (H)]\!](\Gamma \vdash P_2 :: \Sigma, \Sigma', \Pi) =$
$\quad ((\Gamma \vdash P_1) :: (\Gamma\ ;\ H : P_1 \vdash P_2) :: \Sigma, \Sigma',$
$\qquad (hd_1 :: hd_2 :: tl, l) \mapsto \Pi((\text{Let } H : P_1 := hd_1 \text{ in } hd_2) :: tl, l))$
$\mathcal{C}[\![\textbf{we need to prove } P_1\ (H)\ \textbf{ or equivalently } P_2]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) =$
$\quad ((\Gamma \vdash P_2) :: (\Gamma\ ;\ H : P_1 \vdash P) :: \Sigma, \Sigma',$
$\qquad (hd_1 :: hd_2 :: tl, l) \mapsto \Pi((\text{Let } H : P_1 := (hd_1 : P_2 \equiv P_1) \text{ in } hd_2) :: tl, l))$

$\mathcal{C}[\![\textbf{conclude } E_1 = E_2\ j]\!](\Gamma \vdash E_1 = E_3 :: \Sigma, \Sigma', \Pi) =$

$((\Gamma \vdash E_2 = E_3) :: \Sigma, \Sigma',$
$(hd :: tl, l) \mapsto \Pi(((\text{eq_transitive } E_1\ E_2\ E_3\ \mathcal{A}[\![j, E_1 = E_2]\!]\ hd) :: tl, l)))$
$\mathcal{C}[\![\textbf{conclude } E_1 = E_2\ j\ \textbf{done}]\!](\Gamma \vdash E_1 = E_2 :: \Sigma, \Sigma', \Pi) =$
$(\Sigma, \Sigma', (tl, l) \mapsto \Pi(\mathcal{A}[\![j, E_1 = E_2]\!] :: tl, l))$

$\mathcal{C}[\![\textbf{obtain } H\ E_1 = E_2\ j]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) =$
$((\Gamma \vdash E_2 =?) :: (\Gamma\ ;\ H : E_1 =? \vdash P) :: \Sigma, (\Gamma \vdash \text{nat}) :: \Sigma',$
$(hd_1 :: hd_2 :: tl, hd' :: tl') \mapsto$
$\Pi((\text{Let } H : E_1 = hd' := (\text{eq_transitive } E_1\ E_2\ hd'\ \mathcal{A}[\![j, E_1 = E_2]\!]\ hd_1) \text{ in }\ hd_2) :: tl, tl'))$
$\mathcal{C}[\![\textbf{obtain } H\ E_1 = E_2\ j\ \textbf{done}]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) =$
$((\Gamma\ ;\ H : E_1 = E_2 \vdash P) :: \Sigma, \Sigma',$
$(hd :: tl, l) \mapsto \Pi((\text{Let } H : E_1 = E_2 := \mathcal{A}[\![j, E_1 = E_2]\!] \text{ in } hd) :: tl, l))$

$\mathcal{C}[\![= E_2\ j]\!](\Gamma \vdash E_1 =? :: \Sigma, \Sigma', \Pi) =$
$(\Gamma \vdash E_2 =? :: \Sigma, \Sigma', (hd :: tl, E_3 :: l) \mapsto \Pi(\text{eq_transitive } E_1\ E_2\ E_3\ \mathcal{A}[\![j, E_1 = E_2]\!]\ hd :: tl, l))$
$\mathcal{C}[\![= E_2\ j]\!](\Gamma \vdash E_1 = E_3 :: \Sigma, \Sigma', \Pi) =$
$(\Gamma \vdash E_2 = E_3 :: \Sigma, \Sigma', (hd :: tl, l) \mapsto \Pi(\text{eq_transitive } E_1\ E_2\ E_3\ \mathcal{A}[\![j, E_1 = E_2]\!]\ hd :: tl, l))$
$\mathcal{C}[\![= E_2\ j\ \textbf{done}]\!](\Gamma \vdash E_1 =? :: \Sigma, (\Gamma \vdash \text{nat}) :: \Sigma', \Pi) =$
$(\Sigma, \Sigma', (tl, l) \mapsto \Pi(\mathcal{A}[\![j, E_1 = E_2]\!] :: tl, E_2 :: l))$
$\mathcal{C}[\![= E_2\ j\ \textbf{done}]\!](\Gamma \vdash E_1 = E_2 :: \Sigma, \Sigma', \Pi) = (\Sigma, \Sigma', (tl, l) \mapsto \Pi(\mathcal{A}[\![j, E_1 = E_2]\!] :: tl, l))$

$\mathcal{C}[\![\textbf{we proceed by induction on } n \textbf{ to prove } P(n)]\!](\Gamma \vdash P(n) :: \Sigma, \Sigma', \Pi) =$
$((\Gamma \vdash P(O)) :: (\Gamma \vdash \forall m : \text{nat}.P(m) \Rightarrow P(S(m))) :: \Sigma, \Sigma',$
$(hd_1 :: hd_2 :: l, l') \mapsto \Pi((\text{nat_ind}_P\ n\ hd_1\ hd_2) :: l, l'))$
$\mathcal{C}[\![\textbf{we proceed by cases on } n \textbf{ to prove } P(n)]\!](\Gamma \vdash P(n) :: \Sigma, \Sigma', \Pi) =$
$((\Gamma \vdash P(O)) :: (\Gamma \vdash \forall m : \text{nat}.P(S(m))) :: \Sigma, \Sigma',$
$(hd_1 :: hd_2 :: l, l') \mapsto \Pi((\text{nat_cases}_P\ n\ hd_1\ hd_2) :: l, l'))$

$\mathcal{C}[\![\textbf{case } H\ arg_1 \cdots arg_n]\!] =\quad \mathcal{C}[\![arg_1]\!]_a\ \cdots \mathcal{C}[\![arg_n]\!]_a$
$\mathcal{C}[\![(x : T)]\!]_a = \mathcal{C}[\![\textbf{assume } x : T]\!]$
$\mathcal{C}[\![(H : P)]\!]_a = \mathcal{C}[\![\textbf{suppose } P\ (H)]\!]$

$\mathcal{C}[\![\textbf{by induction hypothesis we know } P\ (H)]\!] = \mathcal{C}[\![\textbf{suppose } P\ (H)]\!]$
$\mathcal{C}[\![\textbf{by induction hypothesis we know } P_1\ (H) \textbf{ that is equivalent to } P_2]\!] =$
$\mathcal{C}[\![\textbf{suppose } P_1\ (H) \textbf{ that is equivalent to } P_2]\!]$

$\mathcal{C}[\![\textbf{the thesis becomes } P]\!] = \mathcal{C}[\![\textbf{we need to prove } P]\!]$
$\mathcal{C}[\![\textbf{the thesis becomes } P_1 \textbf{ or equivalently } P_2]\!] =$
$\mathcal{C}[\![\textbf{we need to prove } P_1 \textbf{ or equivalently } P_2]\!]$

5 Natural language generation

We present in Table 7 the inverse translation $\mathcal{G}[\![-]\!]$ from proof terms to declarative proof scripts. The translation is recursive and proceeds by pattern matching over the proof term. Rules coming first take precedence. Recursion on equality chains is performed by the auxiliary function $\mathcal{G}[\![-]\!]^{=}_{-}$ where the argument in subscript position is used to remember the right hand side of the last step in the chain.

The inverse translation we propose generates fully explicit justifications in the form **using**$(H\ E_1 \dots E_n\ H_1 \dots H_m)$. With the same effort it could generate the more lightweight justification **by**$H, H_1, \dots, H_m$. In the implementation in Matita we even added a parameter to force automation to look only for proofs terms that have depth one, i.e. exactly of the form $(H\ E_1 \dots E_n\ H_1 \dots H_m)$.

Table 7: Natural language generation

$\mathcal{G}[\![let\ x := E\ in\ t]\!] =$ **let** $x := E\ \mathcal{G}[\![t]\!]$

$\mathcal{G}[\![\lambda x : T.t]\!] =$ **assume** $x : T\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![((\lambda x : T_2.t) : (\forall x : T_2.P) \equiv (\forall x : T_1.P))]\!] =$
assume $x : T_1$ **that is equivalent to** $T_2\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![((\lambda x : T_2.t) : (\forall x : T_2.P_2) \equiv (\forall x : T_1.P_1))]\!] =$
assume $x : T_1$ **that is equivalent to** T_2
we need to prove P_1 **or equivalently** $P_2\ \mathcal{G}[\![t]\!]$

$\mathcal{G}[\![\Lambda H : P.t]\!] =$ **suppose** $P\ (H)\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![((\Lambda H : P_2.t) : (P_2 \Rightarrow P) \equiv (P_1 \Rightarrow P))]\!] =$
suppose $P_1\ (H)$ **that is equivalent to** $P_2\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![((\Lambda H : P_2.t) : (P_2 \Rightarrow P_4) \equiv (P_1 \Rightarrow P_3))]\!] =$
suppose $P_1\ (H)$ **that is equivalent to** P_2
we need to prove P_3 **or equivalently** $P_4\ \mathcal{G}[\![t]\!]$

$\mathcal{G}[\![Let\ K : P := (H\ E_1 \dots E_n\ H_1 \dots H_m)\ in\ t]\!] =$
using $(H\ E_1 \dots E_n\ H_1 \dots H_m)$ **we proved** $P\ (K)\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![Let\ K : P_2 := ((H\ E_1 \dots E_n\ H_1 \dots H_m) : P_1 \equiv P_2)\ in\ t]\!] =$
using $(H\ E_1 \dots E_n\ H_1 \dots H_m)$ **we proved** $P_1\ (K)$
that is equivalent to $P_2\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![Let\ H : P_2 := (t_1 : P_1 \equiv P_2)\ in\ t_2]\!] =$
we need to prove $P_2\ (H)$ **or equivalently** $P_1\ \mathcal{G}[\![t_1]\!]\ \mathcal{G}[\![t_2]\!]$
$\mathcal{G}[\![Let\ H : E'_1 = E'_3 :=$
$(\text{eq_transitive}\ E'_1\ E'_2\ E'_3\ (H\ E_1 \dots E_n\ H_1 \dots H_m)\ t_2)\ in\ t_1]\!] =$
obtain $H\ E'_1 = E'_2$ **using** $(H\ E_1 \dots E_n\ H_1 \dots H_m)\ \mathcal{G}[\![t_2]\!]^{=}_{E'_3}\ \mathcal{G}[\![t_1]\!]$
$\mathcal{G}[\![Let\ H : P := t_1\ in\ t_2]\!] =$ **we need to prove** $P\ (H)\ \mathcal{G}[\![t_1]\!]\ \mathcal{G}[\![t_2]\!]$

$\mathcal{G}[\![(\text{eq_transitive}\ E'_1\ E'_2\ E'_3\ (H\ E_1 \dots E_n\ H_1 \dots H_m)\ t)]\!] =$
conclude $E'_1 = E'_2$ **using** $(H\ E_1 \dots E_n\ H_1 \dots H_m)\ \mathcal{G}[\![t]\!]^{=}_{E'_3}$
$\mathcal{G}[\![(H\ E_1 \dots E_n\ H_1 \dots H_n)]\!]^{=}_{E'} =$
$= E'$ **using** $(H\ E_1 \dots E_n\ H_1 \dots H_n)$ **done**
$\mathcal{G}[\![(\text{eq_transitive}\ E'_1\ E'_2\ E'_3\ (H\ E_1 \dots E_n\ H_1 \dots H_m)\ t)]\!]^{=}_{E'_3} =$
$= E_2$ **using** $(H\ E_1 \dots E_n\ H_1 \dots H_m)\mathcal{G}[\![\ t]\!]^{=}_{E'_3}$
$\mathcal{G}[\![(\text{ex_elim}\ T\ P_1\ P_2\ (H\ E_1 \dots E_n\ H_1 \dots H_m)\ (\lambda x : T.\Lambda H_2 : P_2.t))]\!] =$
using $(H\ E_1 \dots E_n\ H_1 \dots H_m)$ **let** $x : T$ **such that** $P_2\ (H_2)\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![(\text{and_elim}\ P_1\ P_2\ P_3\ (H\ E_1 \dots E_n\ H_1 \dots H_m)\ (\Lambda H_1 : P_1.\Lambda H_2 : P_2.t))]\!] =$
using $(H\ E_1 \dots E_n\ H_1 \dots H_m)$ **we have** $P_1\ (H_1)$ **and** $P_2\ (H_2)\ \mathcal{G}[\![t]\!]$

$\mathcal{G}[\![(\text{nat_ind}_P\ n\ t_1\ (\lambda m : \text{nat}.\Lambda H : P(m).t_2))]\!] =$
we proceed by induction on n **to prove** $P(n)$

case O
the thesis becomes $P(O)$
$\mathcal{G}[\![t_1]\!]$
case S $(m : nat)$
by induction hypothesis we know $P(m)$ (H)
the thesis becomes $P(S(m))$
$\mathcal{G}[\![t_2]\!]$
$\mathcal{G}[\![(\text{nat_ind}_P\ n\ t_1\ (\lambda m : \text{nat}.$
$((\Lambda H : P(m).t_2) : (P_2 \Rightarrow P(S(m))) \equiv (P(m) \Rightarrow P(S(m))))))]\!] =$
we proceed by induction on n **to prove** $P(n)$
case O
the thesis becomes $P(O)$
$\mathcal{G}[\![t_1]\!]$
case S $(m : nat)$
by induction hypothesis we know $P(m)$ (H)
that is equivalent to P_2
the thesis becomes $P(S(m))$
$\mathcal{G}[\![t_2]\!]$
$\mathcal{G}[\![(\text{nat_ind}_P\ n\ t_1\ (\lambda m : \text{nat}.$
$((\Lambda H : P_2.t_2) : (P(m) \Rightarrow P_3) \equiv (P(m) \Rightarrow P(S(m))))))]\!] =$
we proceed by induction on n **to prove** $P(n)$
case O
the thesis becomes $P(O)$
$\mathcal{G}[\![t_1]\!]$
case S $(m : nat)$
by induction hypothesis we know $P(m)$ (H)
that is equivalent to P_2
the thesis becomes $P(S(m))$ **or equivalently** P_3
$\mathcal{G}[\![t_2]\!]$
$\mathcal{G}[\![(\text{nat_ind}_P\ n\ t_1\ (\lambda m : \text{nat}.\Lambda H : P(m).(t_2 : P_2 \equiv P(S(m)))))]\!] =$
we proceed by induction on n **to prove** $P(n)$
case O
the thesis becomes $P(O)$
$\mathcal{G}[\![t_1]\!]$
case S $(m : nat)$
by induction hypothesis we know $P(m)$ (H)
the thesis becomes $P(S(m))$ **or equivalently** P_2
$\mathcal{G}[\![t_2]\!]$

$\mathcal{G}[\![(H\ E_1 \ldots E_n\ H_1 \ldots H_m)]\!] =$ **using** $(H\ E_1 \ldots E_n\ H_1 \ldots H_m)$ **done**
$\mathcal{G}[\![((H\ E_1 \ldots E_n\ H_1 \ldots H_m) : P_1 \equiv P_2)]\!] =$
using $(H\ E_1 \ldots E_n\ H_1 \ldots H_m)$ **we proved** P_1
that is equivalent to P_2 **done**

$\mathcal{G}[\![(t_1 : P_1 \equiv P_2)]\!] =$
we need to prove P_2 **or equivalently** P_1
$\mathcal{G}[\![t_1]\!]$

The following important theorem shows that the proof term obtained processing a declarative script generated from a given proof term is identical to the starting proof term. Thus, we fully satisfy the strongest requirement of Section 2 about re-generation of declarative scripts.

Theorem 1 (Round-tripping from proof terms)

1. *For all Γ, P, t such that $\Gamma \vdash t : P$ and for all Σ, Π there exists an unique Π' such that*
   (a) $\mathcal{C}[\![\mathcal{G}[\![t]\!]]\!]^\star((\Gamma \vdash P) :: \Sigma, [], \Pi) = (\Sigma, [], \Pi')$
   (b) *for all* $l, \Pi(t :: l, []) = \Pi'(l, [])$
2. *For all Γ, E_1, E_2, t such that $\Gamma \vdash t : E_1 = E_2$, and for all Σ, Π there exists an unique Π' such that*
   (a) $\mathcal{C}[\![\mathcal{G}[\![t]\!]^{=}_{E_2}]\!]^\star((\Gamma \vdash E_1 = E_2) :: \Sigma, [], \Pi) = (\Sigma, [], \Pi')$
   (b) *for all* $l, \Pi(t :: l, []) = \Pi'(l, [])$
3. *For all $\Gamma, \Gamma', E_1, E_2,$ such that $\Gamma \vdash t : E_1 = E_2$, and for all Σ, Π there exists an unique Π' such that*
   (a) $\mathcal{C}[\![\mathcal{G}[\![t]\!]^{=}_{E_2}]\!]^\star((\Gamma \vdash E_1 =?) :: \Sigma, [\Gamma' \vdash \mathbb{N}], \Pi) = (\Sigma, [], \Pi')$
   (b) *for all* $l, \Pi(t :: l, [E_2]) = \Pi'(l, [])$.

In particular, for all P, t such that $\vdash t : P$ we have $\mathcal{C}[\![\mathcal{G}[\![t]\!], P]\!]_s = t$

The statement is made of three parts, one for $\mathcal{G}[\![\cdot]\!]$ and two for $\mathcal{G}[\![\cdot]\!]^{=}_{\cdot}$. As a particular case of the first part we have the last statement that justifies the name of the theorem. The third part handles the case when we are plugging the generated script in a proof of an equality chain whose final conclusion is yet unknown and will be inferred from the generated script.

Proof We prove the particular case first assuming that the first statement holds. $\mathcal{C}[\![\mathcal{G}[\![t]\!], P]\!]_s = \pi_3(\mathcal{C}[\![\mathcal{G}[\![t]\!]]\!]^\star\ ([\vdash P], [], ([H], []) \mapsto H))([], []) = \Pi'([], [])$ where $([H], []) \mapsto H)([t], []) = t = \Pi'([], [])$. Hence $\mathcal{C}[\![\mathcal{G}[\![t]\!], P]\!]_s = t$.

The remaining three statements are proved by structural induction on t. The proofs of the second and third one are required to complete the proof of the first one. We only show two significant cases.

Case λ + conversion for the first statement:

Let t be $(\lambda x : T_2.t') : (\forall x : T_2.P_2) \equiv (\forall x : T_1.P_1)$. We have $\mathcal{G}[\![t]\!] = S_1\ S_2\ \mathcal{G}[\![t']\!]$ where $S_1 =$ "**assume** $x : T_1$ **that is equivalent to** T_2" and $S_2 =$ "**we need to prove** P_1 **or equivalently** P_2". Assume generic, but fixed Σ and Π. We have

$$
\begin{aligned}
&\mathcal{C}[\![S_1\ S_2\ \mathcal{G}[\![t']\!]]\!]^\star((\Gamma \vdash \forall x : T_1.P_1) :: \Sigma, [], \Pi) \\
&= (\mathcal{C}[\![\mathcal{G}[\![t']\!]]\!] \circ \mathcal{C}[\![S_2]\!] \circ \mathcal{C}[\![\text{“ \textbf{assume} } x : T_1 \text{ \textbf{that is equivalent to} } T_2''\,]\!]) \\
&\qquad ((\Gamma \vdash \forall x : T_1.P_1) :: \Sigma, [], \Pi) \\
&= \mathcal{C}[\![\mathcal{G}[\![t']\!]]\!](\mathcal{C}[\![S_2]\!]((\Gamma\ ;\ x : T_2 \vdash P_1) :: \Sigma, [], \\
&\qquad \left\{ \begin{array}{ll} ((hd : P_2 \equiv P_1) :: tl, []) & \mapsto \Pi(((\lambda x : T_2.hd) : (\forall x : T_2.P_2) \equiv (\forall x : T_1.P_1)) :: tl, []) \\ (hd :: tl, []) & \mapsto \Pi(((\lambda x : T_2.hd) : (\forall x : T_2.P_1) \equiv (\forall x : T_1.P_1)) :: tl, []) \end{array} \right.)) \\
&= \mathcal{C}[\![\mathcal{G}[\![t']\!]]\!](((\Gamma\ ;\ x : T_2 \vdash P_2) :: \Sigma, [], \\
&\qquad (hd :: tl, []) \mapsto \Pi(((\lambda x : T_2.hd) : (\forall x : T_2.P_2) \equiv (\forall x : T_1.P_1)) :: tl, []))) \\
&= (\Sigma, [], (l, []) \mapsto \Pi(((\lambda x : T_2.t') : (\forall x : T_2.P_2) \equiv (\forall x : T_1.P_1)) :: l, [])
\end{aligned}
$$

The last identity is justified by the inductive hypothesis on t'. To conclude the case we just need to verify that $\forall l, \Pi(((\lambda x : T_2.t') : (\forall x : T_2.P_2) \equiv (\forall x : T_1.P_1)) :: l, []) = \Pi(((\lambda x : T_2.t') : (\forall x : T_2.P_2) \equiv (\forall x : T_1.P_1)) :: l, [])$, which is trivially true.

Case application for the third statement:

Let t be $(H\ F_1 \dots F_n\ H_1 \dots H_m)$.

We have $\mathcal{G}[\![t]\!]^{=}_{E_2} =$ "$= E_2$ **using** $(H\ F_1 \dots F_n\ H_1 \dots H_m)$ **done**". Assume generic but fixed Σ and Π. We have

$$\begin{aligned}
&\mathcal{C}[\![\text{``} = E_2 \textbf{ using } (H\ F_1 \dots F_n\ H_1 \dots H_m) \textbf{ done}''\,]\!]^\star((\Gamma \vdash E_1 =?) :: \Sigma, [\Gamma' \vdash \mathbb{N}], \Pi)\\
&= (\Sigma, [], (tl, [])) \mapsto \Pi(\mathcal{A}[\![\textbf{ using } (H\ F_1 \dots F_n\ H_1 \dots H_m), E_1 = E_2]\!] :: tl, E_2 :: []))\\
&= (\Sigma, [], (tl, [])) \mapsto \Pi((H\ F_1 \dots F_n\ H_1 \dots H_m) :: tl, E_2 :: []))
\end{aligned}$$

Once again, a quick verification by reflexivity completes the proof.

□

The next theorem shows that all the requirements about re-generation of declarative scripts of Section 2 are fulfilled: the declarative script re-generated from a proof term is an improved version of the starting declarative script. Moreover re-generation is idempotent. Improvement is captured by the map $\rhd$ that consists in:

1) interpolating new statements corresponding to the explicitation of justifications previously found automatically or given by means of a proof term more complex than an application. For instance

$$\text{``}\textbf{using } \Lambda H : A.H \textbf{ done}\text{''} \rhd \text{``}\textbf{suppose } A\ (H)\text{''}\ \ \text{``}\textbf{using } H \textbf{ done}\text{''}$$

2) replacing sequences of statements with semantically equivalent ones that are more appropriate in their context. For instance the formal semantics of Table 6 shows that **the thesis becomes** P is equivalent to **we need to prove** P. However the former is supposed to be used only to state the thesis of a branch in a proof by induction or case analysis. The map $\rhd$ also captures the notion of "being less appropriate then". For instance

$$\begin{aligned}
&\text{``}\textbf{conclude } E_1 = E_2\ j_1\text{''}\ \ \text{``}j_2 \textbf{ done}''\\
\rhd\ &\text{``}\textbf{conclude } E_1 = E_2\ j_1\text{''}\ \ \text{``} = E_3\ j_2 \textbf{ done}\text{''}
\end{aligned}$$

since, once a chain of inequation is started, the same style must be used until the end of the chain.

3) replacing sequences of statements with shorter ones that are semantically equivalent. In particular, equality chains with just one step are changed to other statements. The following is an example that does not involve an equality chain.

$$\begin{aligned}
&\text{``}\textbf{we need to prove } P_1\ (H)\text{''}\\
&\quad \text{``}\textbf{we need to prove } P_1 \textbf{ that is equivalent to } P_2\text{''}\\
\rhd\ &\text{``}\textbf{we need to prove } P_1\ (H) \textbf{ that is equivalent to } P_2\text{''}
\end{aligned}$$

Of the three kind of transformations, only the latter one can modify the script against the user will. This happens, for instance, when the user explicitly recalls the

current thesis where not necessary. This can be avoided by improving only selected parts of the script, since improvement is (almost) structure preserving.

The main property of the map $\rhd$ is that it is reflexive only on scripts that cannot be improved and that is reaches a fixpoint in one step (i.e. that it maximally improves the script it is applied to).

Formally, a script $\mathcal{S}_1$ is improved in a script $\mathcal{S}_2$, i.e. $\mathcal{S}_1 \rhd \mathcal{S}_2$, if $\mathcal{S}_2$ is the normal form of $\mathcal{S}_1$ according to the contextual and conditional rewriting system given in Table 8. The rewriting system is confluent, since it has no critical pairs, and terminating. All commands generated by automation in the rewriting rules are justified by simple steps and are already in improved form.

Table 8: Improvement map on declarative scripts

"$C[S_1 \ \dots \ S_n]$" $\rhd$ "$C[S'_1 \ \dots \ S'_{n'}]$" for every context $C[\cdot]$
when "$S_1 \ \dots \ S_n$" $\rhd$ "$S'_1 \ \dots \ S'_{n'}$" and no other contextual rewriting rule apply

"C[**assume** $x : nat$]" $\rhd$ "C[**case** S $(x : nat)$]" when $C[\cdot]$ is the context
"**we proceed by (cases|induction) on** t **to prove** P **case** O $S_1 \ \dots \ S_n$ $\cdot$ "
and S_n is the last step in the proof of the base case. This condition can be detected syntactically.

"C[**suppose** P_1 (H)]" $\rhd$ "C[**by induction hypothesis we know** P_1 (H)]"
when $C[\cdot]$ is the context
"**we proceed by induction on** t **to prove** P **case** O $S_1 \ \dots \ S_n$
case S $(x : nat) \cdot$ "
and S_n is the last step in the proof of the base case.
"C[**suppose** P_1 (H) **that is equivalent to** P_2]" $\rhd$
"C[**by induction hypothesis we know** P_1 (H) **that is equivalent to** P_2]"
when $C[\cdot]$ is the context
"**we proceed by induction on** t **to prove** P **case** O $S_1 \ \dots \ S_n$
case S $(x : nat) \cdot$ "
and S_n is the last step in the proof of the base case.

"j **we proved** P (K)" $\rhd$ "**using** $(H \ E_1 \ \dots \ E_n \ H_1 \ \dots \ H_m)$ **we proved** P (K)"
when j is (equivalent to) a simple justification
"j **we proved** $E_1 = E_2$ (H)" $\rhd$ "**obtain** $H E_1 = E_2$ $S'_1 \ \cdots \ S'_n$"
when j is not simple and it generates the rewriting chain $S'_1 \ \dots \ S'_n$
"j **we proved** P_1 (H)" $\rhd$ "**we need to prove** P_1 (H) $S'_1 \ \cdots \ S'_n$"
when P_1 is not an equality and j is not simple, it does not immediately applies conversion and it generates $S'_1 \ \dots \ S'_n$
"j **we proved** P_1 (H)" $\rhd$
"**using** $(H \ E_1 \ \dots \ E_n \ H_1 \ \dots \ H_m)$ **we proved** P_2 (H) **that is equivalent to** P_1"
when j is not simple, it immediately converts P_1 to P_2 and it generates a simple justification
"j **we proved** P_1 (H)" $\rhd$
"**we need to prove** P_1 (H) **or equivalently** P_2 $S'_1 \ \cdots \ S'_n$"
when j is not simple, it immediately converts P_1 to P_2 and it generates $S'_1 \ \dots \ S'_n$
"j **we proved** P_1 (K) **that is equivalent to** P_2" $\rhd$
"**using** $(H \ E_1 \ \dots \ E_n \ H_1 \ \dots \ H_m)$ **we proved** P_1 (K) **that is equivalent to** P_2"

when j is (equivalent to) a simple justification
"j **we proved** P_1 (H) **that is equivalent to** P_2" $\triangleright$
"**we need to prove** P_2 (H) **or equivalently** P_1 $S'_1 \cdots S'_{n'}$"
when j is not simple and it generates $S'_1 \ldots S'_{n'}$
"$C[j$ **we proved** P **done**$]$" $\triangleright$ "$C[S'_1 \cdots S'_{n'}]$" when j generates $S'_1 \ldots S'_{n'}$
In particular $S'_1 \ldots S'_{n'}$ can be the last steps of an equality chain when the hole of $C[\cdot]$ is at the end of an equality chain.
"$C[j$ **done**$]$" $\triangleright$ "$C[S'_1 \cdots S'_{n'}]$" when j generates $S'_1 \ldots S'_{n'}$
In particular $S'_1 \ldots S'_{n'}$ can be the last steps of an equality chain when the hole of $C[\cdot]$ is at the end of an equality chain.
"$C[j$ **we proved** P_1 **that is equivalent to** P_2 **done**$]$" $\triangleright$
"$C'[$**using** $(H\ E_1 \ldots E_n\ H_1 \ldots H_m)$ **we proved** P_1 (K)
that is equivalent to P_2 **done**$]$"
when $C[\cdot]$ is "$C'[$**we need to prove** P_2 (K) $\cdot]$" and j is (equivalent to) a simple justification
"$C[j$ **we proved** P_1 **that is equivalent to** P_2 **done**$]$" $\triangleright$
"$C'[$**we need to proved** P_2 (K) **that is equivalent to** P_1 **done** $S'_1 \ldots S'_{n'}]$"
when $C[\cdot]$ is "$C'[$**we need to prove** P_2 (K) $\cdot]$" and j generates $S'_1 \ldots S'_{n'}$
"j **we proved** P_1 **that is equivalent to** P_2 **done**" $\triangleright$
"**using** $(H\ E_1 \ldots E_n\ H_1 \ldots H_m)$ **we proved** P_1
that is equivalent to P_2 **done**"
when j is (equivalent to) a simple justification
"j **we proved** P_1 **that is equivalent to** P_2 **done**" $\triangleright$
"**we need to prove** P_2 **or equivalently** P_1 $S'_1 \cdots S'_{n'}$"
when j is not simple and generates $S'_1 \ldots S'_{n'}$

"j **let** $x : T$ **such that** P_1 (H)" $\triangleright$
"**using** $(H\ E_1 \ldots E_n\ H_1 \ldots H_m)$ **let** $x : T$ **such that** P_1 (H)"
since j must be (equivalent to) a simple justification
"j **we have** P_1 (H_1) **and** P_2 (H_2)" $\triangleright$
"**using** $(H\ E_1 \ldots E_n\ H_1 \ldots H_m)$ **we have** P_1 (H_1) **and** P_2 (H_2)"
since j must be (equivalent to) a simple justification

"$C[$**we need to prove** $P]$" $\triangleright$ "$C[$**the thesis becomes** $P]$"
when $C[\cdot]$ has its hole just after "**case** O" or just after
"**by induction hypothesis . . .**" and the hole is not followed by
"**the thesis becomes . . .**"
"**we need to prove** P" $\triangleright$
"$C[$**we need to prove** $\forall x : T_1.P_1$ **or equivalently** $\forall x : T_2.P_1]$" $\triangleright$
"$C'[$**assume** $x : T_1$ **that is equivalent to** $T_2]$"
when $C[\cdot]$ is $C'[\cdot$ "**assume** $x : T_2$"$]$
"$C[$**we need to prove** $\forall x : T_1.P_1$ **or equivalently** $\forall x : T_2.P_2]$" $\triangleright$
"$C'[$**assume** $x : T_1$ **that is equivalent to** T_2 **we need to prove** P_1
or equivalently $P_2]$"
when $C[\cdot]$ is $C'[\cdot$ "**assume** $x : T_2$"$]$
"**we need to prove** P" $\triangleright$
"$C[$**we need to prove** $P_1 \Rightarrow P_2$ **or equivalently** $P'_1 \Rightarrow P_2]$" $\triangleright$
"$C'[$**suppose** P_1 (H) **that is equivalent to** $P'_1]$"
when $C[\cdot]$ is $C'[\cdot$ "**suppose** P'_1 (H)"$]$

“C[**we need to prove** $P_1 \Rightarrow P_2$ **or equivalently** $P_1' \Rightarrow P_2'$]” ▷
  “C'[**suppose** P_1 (H) **that is equivalent to** P_1' **we need to prove** P_2
    or equivalently P_2']”
  when $C[\cdot]$ is $C'[\cdot$ “**suppose** P_1' (H)”]
“C[**we need to prove** P_1 **or equivalently** P_2]” ▷
  “C'[**using** $(H\ E_1\ \dots\ E_n\ H_1\ \dots\ H_m)$ **we proved** P_2
    that is equivalent to P_1 **done**]”
  when $C[\cdot]$ is either $C'[\cdot$ “j **done**”] or $C'[\cdot$ “j **we proved** P_2 **done**”]
  and j is (equivalent to) a simple justification
“C[**we need to prove** P_1 (K)]” ▷
  “C'[**using** $(H\ E_1\ \dots\ E_n\ H_1\ \dots\ H_m)$ **we proved** P_1 (K)]”
  when $C[\cdot]$ is either $C'[\cdot$ “j **done**”] or $C'[\cdot$ “j **we proved** P_2 **done**”]
  and j is (equivalent to) a simple justification
“C[**we need to prove** P_1 (K)]” ▷
  “C'[**using** $(H\ E_1\ \dots\ E_n\ H_1\ \dots\ H_m)$ **we proved** P_2 (K)
    that is equivalent to P_1]”
  when $C[\cdot]$ is $C[\cdot$ “j **we proved** P_2 **that is equivalent to** P_1”]
  and j is (equivalent to) a simple justification
“C[**we need to prove** $F_1 = F_2$ (K)]” ▷
  “C'[**obtain** K $F_1 = F_2$ **using** $(H\ E_1\ \dots\ E_n\ H_1\ \dots\ H_m)$]”
  when $C[\cdot]$ is $C'[\cdot$ “**conclude** $F_1 = F_2$ j”] and j is (equivalent to) a
  simple justification
“C[**we need to prove** P_1 (K)]” ▷
  “C'[**we need to prove** P_1 (K) **or equivalently** P_2]”
  when $C[\cdot]$ is $C'[\cdot$ “**we need to prove** P_1 **or equivalently** P_2”]
“C[**we need to prove** P_1 (K) **or equivalently** P_2]” ▷
  “C'[**using** $(H\ E_1\ \dots\ E_n\ H_1\ \dots\ H_m)$ **we proved** P_2 (K)
    that is equivalent to P_1]”
  when $C[\cdot]$ is either $C'[\cdot$ “j **done**”] or $C'[\cdot$ “j **we proved** P_2 **done**”]
  and j is (equivalent to) a simple justification

“C[**conclude** $F_1 = F_2$ j **done**]” ▷ “$C[S_1'\ \dots\ S_{n'}']$”
  where j generates the script $S_1'\ \dots\ S_{n'}'$ In particular, the generated script can
  be an equality chain or it can continue an equality chain if the hole in $C[\cdot]$ is
  at the end of an equality chain
“**conclude** $F_1 = F_2$ j” ▷
  “**conclude** $F_1 = F_2$ **using** $(H\ E_1\ \dots\ E_n\ H_1\ \dots\ H_m)$ $S_1'\ \dots\ S_{n'}'$”
  where $S_1'\ \dots\ S_{n'}'$ are all equality chain steps generated from j

“**obtain** K $F_1 = F_2$ j **done**” ▷
  “**using** $(H\ E_1\ \dots\ E_n\ H_1\ \dots\ H_m)$ **we proved** $F_1 = F_2$ (K)”
  when j is (equivalent to) a simple justification
“**obtain** K $F_1 = F_2$ j **done**” ▷
  “**using** $(H\ E_1\ \dots\ E_n\ H_1\ \dots\ H_m)$ **we proved** P
    that is equivalent to $F_1 = F_2$ (K)”
  when j immediately performs a conversion and then it is equivalent to a
  simple justification
“**obtain** K $F_1 = F_2$ j **done**” ▷
  “**we need to prove** $F_1 = F_2$ (K) $S_1'\ \dots\ S_{n'}'$”

when j generates the commands $S'_1 \ \ldots \ S'_{n'}$
"**obtain** $K \ F_1 = F_2 \ j$ **done**"$\triangleright$
"**we need to prove** $F_1 = F_2 \ (K)$ **or equivalently** $P \ S'_1 \ \ldots \ S'_{n'}$"
when j immediately performs a conversion and then it generates the commands $S'_1 \ \ldots \ S'_{n'}$
"**obtain** $K \ F_1 = F_2 \ j$"$\triangleright$
"**obtain** $K \ F_1 = F_2$ **using** $(H \ E_1 \ \ldots \ E_n \ H_1 \ \ldots \ H_m) \ S'_1 \ \ldots \ S'_{n'}$"
where $S'_1 \ \ldots \ S'_{n'}$ are all equality chain steps generated from j

"$= F_2 \ j$" $\triangleright$ "$= F_3$ **using** $(H \ E_1 \ \ldots \ E_n \ H_1 \ \ldots \ H_m) \ S'_1 \ \ldots \ S'_{n'}$"
where j generates the rewriting steps $S'_1 \ \ldots \ S'_{n'}$ that prove $F_3 = F_2$
"$= F_2 \ j$ **done**" $\triangleright$ "$= F_2$ **using** $(H \ E_1 \ \ldots \ E_n \ H_1 \ \ldots \ H_m)$ **done**"
where j is (equivalent to) a simple justification
"$= F_2 \ j$ **done**" $\triangleright$ "$= F_3$ **using** $(H \ E_1 \ \ldots \ E_n \ H_1 \ \ldots \ H_m) \ S'_1 \ \ldots \ S'_{n'}$"
where j generates the rewriting steps $S'_1 \ \ldots \ S'_{n'}$ that prove $F_3 = F_2$ and $S'_{n'}$ ends the chain

"C[**case** $H \ arg_1 \ \cdots \ arg_n$]" $\triangleright$ "$C[S_1 \ \ldots \ S_n]$"
when the hole in $C[\cdot]$ is not at the beginning of a case in an inductive proof or a proof by cases, and S_i is "**assume** $x : T$" when arg_i is $(x : T)$ and "**suppose** $P \ (K)$" when arg_i is $(K : P)$

"C[**by induction hypothesis we know** $P \ (H)$]" $\triangleright$ "C[**suppose** $P \ (H)$]"
when the hole in $C[\cdot]$ does not follow a "**case**" command in a proof by induction
"C[**by induction hypothesis we know** $P_1 \ (H)$ **that is equivalent to** P_2]"$\triangleright$
"C[**suppose** $P_1 \ (H)$ **that is equivalent to** P_2]"
when the hole in $C[\cdot]$ does not follow a "**case**" command in a proof by induction

"$C[]$" $\triangleright$ "C[**the thesis becomes** P]"
when the hole in $C[\cdot]$ immediately follows a "**case** O" or a "**by induction hypothesis...**" command and it is not followed by a "**the thesis becomes ...**" or a "**we need to prove ...**" command

Idempotence of improvement requires a lengthy but simple verification. Thus we have the following fact.

Fact 1 (Idempotence of improvement)
For all $S_1, \ldots, S_n, S'_1, \ldots S'_m, S''_1, \ldots S''_{m'}$, *if* $S_1 \ \cdots \ S_n \triangleright S'_1 \ \cdots \ S'_m \triangleright S''_1 \ \cdots \ S''_{m'}$ *then* $m = m'$ *and* $\forall i \leq m, S'_i = S''_i$

In order to make the proof of our last theorem easier, we introduce the notion of improvement in an equality chain.

Definition 1 (Improvement in an equality chain) We write $S_1 \ \ldots \ S_n \triangleright^= S'_1 \ \ldots \ S'_m$ iff $C[S_1 \ \ldots \ S_n] \triangleright^= C[S'_1 \ \ldots \ S'_m]$ where $C[\cdot] =$ "**conclude** $0 = 0 \ \cdot$ ".

The chosen equality $0 = 0$ in the previous definition is just dummy, since all contextual rules in the definition of improvement ignore the exact equality stated in a **conclude** or an **obtain** command.

Theorem 2 (Round-tripping from declarative scripts)
For all $n \in \mathbb{N}$ and for all $S_1, \ldots, S_n, \Sigma, \Sigma', \Pi$, if $\mathcal{C}[\![S_1 \ \cdots \ S_n]\!]^\star(\Sigma, \Sigma', \Pi) = (\Sigma_1, \Sigma'_1, \Pi_1)$ where $\Sigma = [\Gamma_1 \vdash P_1; \ldots; \Gamma_v \vdash P_v]$ then there exists $k \leq v$, unique v' and h and unique $t_1, \ldots, t_k$ and l' such that:

1. $\Sigma_1 = [\Gamma'_1 \vdash P'_1; \ldots; \Gamma'_{v'} \vdash P'_{v'}; \Gamma_{k+1} \vdash P_{k+1}; \ldots; \Gamma_v \vdash P_v]$
   $\Sigma'_1 = [\Gamma''_1 \vdash T''_1; \ldots; \Gamma''_h \vdash T''_h]$
2. *for each $i < k$, the term t_i is closed in Γ_i and has type P_i (or one instance of P_1 if $i = 1$ and Σ' is not empty)*
3. *$t_k[l, l']$ is a context (a term with holes for proof terms and expressions) that behaves as the map $(l, l') \mapsto s$; the list l binds a proof term variable for each proof goal $\Gamma'_i \vdash P'_i$; once applied, the output of the map is a term closed in Γ_k that has type P_k (or its instance where the placeholder ? has been instantiated with the element of the singleton list l', if Σ'_1 is not empty)*
4. *For all $r_1, \ldots, r_{v'}, E_1, \ldots, E_h, l$ there exists l'' such that*
   $$\Pi(t_1 :: \cdots :: t_{k-1} :: t_k[r_1 :: \cdots :: r_{v'}, E_1 :: \cdots E_h] :: l, l') = \Pi_1(r_1 :: \cdots :: r_{v'} :: l, l'')$$
5. *For all $r_1, \ldots, r_{v'}, E_1, \ldots, E_h$,*
   $$S_1 \ \cdots \ S_n \ \mathcal{G}[\![r_1]\!] \ \cdots \ \mathcal{G}[\![r_{v'}]\!] \rhd \mathcal{G}[\![t_1]\!] \ \cdots \ \mathcal{G}[\![t_{k-1}]\!] \ \mathcal{G}[\![t_k[[r_1, \ldots, r_{v'}], [E_1, \ldots, E_h]]]\!]$$
   Moreover if t_1 has type "$F_1 = F_2$" then for all $r_1, \ldots, r_{v'}, E_1, \ldots, E_h$,
   $$S_1 \ \cdots \ S_n \ \mathcal{G}[\![r_1]\!] \ \cdots \ \mathcal{G}[\![r_{v'}]\!] \rhd^{=} \mathcal{G}[\![t_1]\!]^{=}_{F_2} \ \mathcal{G}[\![t_2]\!] \ \cdots \ \mathcal{G}[\![t_{k-1}]\!] \ \mathcal{G}[\![t_k[[r_1, \ldots, r_{v'}], [E_1, \ldots, E_h]]]\!]$$

In particular, for $\Sigma = [\vdash P]$, $\Sigma' = []$, $\Sigma_1 = []$ and $\Sigma'_1 = []$ we have $k = 1$ and $t_k[[], []]$ is a closed term of type P such that $S_1 \ \cdots \ S_n \rhd \mathcal{G}[\![t_k[[], []]]\!]$ (if P is not an equality) or $S_1 \ \cdots \ S_n \rhd \mathcal{G}[\![t_k[[], []]]\!]^{=}_{F_2}$ (if P is $F_1 = F_2$).

The statement of the theorem is rather technical, but corresponds to a clear intuition, captured by the first four conditions above. The intuition is the following: the sequence of commands $S_1, \ldots, S_n$ completely proves the first $k - 1$ goals of Σ and begins the proof of the k-th goal by opening v' new goals. Thus the user is left (in Σ_1) with the new goals and with the remaining old goals. Moreover, the proof of the first goal may have instantiated any placeholder declared in Σ', and the commands that have started the proof of the k-th goal may have introduced a new placeholder declaration in Σ'_1. The proof terms that inhabit the statements of the closed goals are $t_1, \ldots, t_{k-1}$ and the proof term context $t_k[l, l']$ generates a proof term for the k-th statement once filled with proof terms for the new v' goals.

Condition 5 is the one that forces round-tripping: it states that the script $S_1 \ \cdots \ S_n$, followed by a script to prove all the new goals, is improved in the script generated from the proof terms that inhabits the closed goals. Moreover, from the previous lemma, we know that the improved script is a fixpoint of the transformation.

The particular case at the end of the statement is the one that gives the name to the theorem. It says that, given a statement and a declarative script that proves the statement, executing the script generates a closed proof term (the proof object for the script) that can be used to re-generate an improved and equivalent declarative script.

Proof The particular case is a simple instantiation of the general case. The proof of the general case is by induction on n (or, equivalently, by structural induction on the list $S_1 \ \cdots \ S_n$). The base case ($n = 0$) is vacuously true. In the inductive case we

know that the theorem holds for $S_2\ \cdots\ S_n$ and we must show that it holds also for $S_1\ S_2\ \cdots\ S_n$. We proceed by cases on S_1 and we do some simple verifications.

Since the definition of improvement is contextual, in a few cases we need to perform a one-step look-ahead by proceeding by cases also on S_2 and using the induction hypothesis on $S_3\ \cdots\ S_n$. These are the cases where the contextual rule for $\rhd$ considers a context $C[\cdot]$ of the form $C'[\cdot\ S]$ for some particular S. The remaining contextual rules either deal with equality chains or with proofs by cases/induction. To handle the first ones we have strengthened the statement of the theorem with the second requirement of the fifth condition, that deals with improvement in equality chains. Thus the induction hypothesis is sufficient without requiring any unbounded look-ahead.

We only show one simple, but significant case (since it shows a potential improvement of the script). Let $S_1 =$ "j **done**" and let P_1 be "$F_1 = F_2$" (hence $\Sigma' = []$).

$$\begin{aligned}
&\mathcal{C}[\![S_1\ S_2\ \cdots\ S_n]\!]^\star(\Gamma \vdash F_1 = F_2 :: \Sigma, [], \Pi)\\
&= (\mathcal{C}[\![S_2\ \cdots\ S_n]\!]^\star \circ \mathcal{C}[\![S_1]\!])(\Gamma \vdash F_1 = F_2 :: \Sigma, [], \Pi)\\
&= \mathcal{C}[\![S_2\ \cdots\ S_n]\!](\Sigma, [], (l, []) \mapsto \Pi(\mathcal{A}[\![j, F_1 = F_2]\!] :: l, []))\\
&= (\Sigma_1, \Sigma_1', \Pi_1)
\end{aligned}$$

By induction hypothesis we obtain $k < v-1, t_1, \ldots, t_k$ and l'. We need to prove that $\exists \bar{k}$ and $\exists \bar{t}_1, \ldots, \bar{t}_k$ and $\bar{l}'$ that satisfy the required properties. We take $k+1$ for $\bar{k}$, $\mathcal{A}[\![j, F_1 = F_2]\!], t_1, \ldots, t_k$ for $\bar{t}_1, \ldots, \bar{t}_{\bar{k}}$ and l' for $\bar{l}'$. Properties 2 and 3 are trivial. To prove property 4, we know by induction hypothesis that $\forall r_1, \ldots, r_{v'}, \forall E_1, \ldots, E_h, \forall l, \exists l''$,

$$\begin{aligned}
&\Pi_1(r_1 :: \cdots :: r_v :: l, l'')\\
&= \Pi(\mathcal{A}[\![j, F_1 = F_2]\!] :: t_1 :: \cdots :: t_{k-1} :: t_k[r_1 :: \cdots :: r_v, E_1 :: \cdots E_h] :: l, l')\\
&= \Pi(\bar{t}_1 :: \cdots :: \bar{t}_{\bar{k}-1} :: t_{\bar{k}}[r_1 :: \cdots :: r_v, E_1 :: \cdots E_h] :: l, l')
\end{aligned}$$

Hence property 4 holds. Finally, to prove property 5, by induction hypothesis we know that $\forall r_1, \ldots, r_{v'}, \forall E_1, \ldots, E_h$,

$$S_2\ \cdots\ S_n\ \mathcal{G}[\![r_1]\!]\ \cdots\ \mathcal{G}[\![r_{v'}]\!] \rhd \mathcal{G}[\![t_1]\!]\ \cdots\ \mathcal{G}[\![t_{k-1}]\!]\ \mathcal{G}[\![t_k[[r_1, \ldots, r_{v'}], [E_1, \ldots, E_h]]]\!]$$

Moreover if t_1 has type "$G_1 = G_2$" then $\forall r_1, \ldots, r_{v'}, \forall E_1, \ldots, E_h$,

$$\begin{aligned}
&S_2\ \cdots\ S_n\ \mathcal{G}[\![r_1]\!]\ \cdots\ \mathcal{G}[\![r_{v'}]\!] \rhd^=\\
&\quad \mathcal{G}[\![t_1]\!]^=_{G_2}\ \mathcal{G}[\![t_2]\!]\ \cdots\ \mathcal{G}[\![t_{k-1}]\!]\ \mathcal{G}[\![t_k[[r_1, \ldots, r_{v'}], [E_1, \ldots, E_h]]]\!]
\end{aligned}$$

Since $\bar{t}_1 = \mathcal{A}[\![j, F_1 = F_2]\!]$ has type $F_1 = F_2$, we must prove both conditions of property five. Both proofs use the inductive hypothesis. We only show the case where the justification is equivalent to a simple one, i.e. $\bar{t}_1 = (H\ G_1\ \ldots\ G_n\ H_1\ \ldots\ H_m)$. We have $\forall r_1, \ldots, r_{v'}, \forall E_1, \ldots, E_h$,

$$\begin{aligned}
&\text{“}j\ \mathbf{done}\text{”}\ S_2\ \cdots\ S_n\ \mathcal{G}[\![r_1]\!]\ \cdots\ \mathcal{G}[\![r_{v'}]\!]\\
&\rhd \text{“}\mathbf{using}\ (H\ G_1\ \ldots\ G_n\ H_1\ \ldots\ H_m)\ \mathbf{done}\text{”}\\
&\quad \mathcal{G}[\![t_1]\!]\ \cdots\ \mathcal{G}[\![t_{k-1}]\!]\ \mathcal{G}[\![t_k[[r_1, \ldots, r_{v'}], [E_1, \ldots, E_h]]]\!]\\
&= \mathcal{G}[\![\mathcal{A}[\![j, F_1 = F_2]\!]]\!]\ \mathcal{G}[\![t_1]\!]\ \cdots\ \mathcal{G}[\![t_{k-1}]\!]\ \mathcal{G}[\![t_k[[r_1, \ldots, r_{v'}], [E_1, \ldots, E_h]]]\!]\\
&= \mathcal{G}[\![\bar{t}_1]\!]\ \mathcal{G}[\![\bar{t}_2]\!]\ \cdots\ \mathcal{G}[\![\bar{t}_{\bar{k}-1}]\!]\ \mathcal{G}[\![\bar{t}_{\bar{k}}[[r_1, \ldots, r_{v'}], [E_1, \ldots, E_h]]]\!]
\end{aligned}$$

and

$$
\begin{array}{ll}
& \text{“}j\ \textbf{done}\text{”}\ S_2\ \cdots\ S_n\ \mathcal{G}[\![r_1]\!]\ \cdots\ \mathcal{G}[\![r_{v'}]\!] \\
\rhd^{=} & \text{“} = F_2\ \textbf{using}\ (H\ G_1\ \ldots\ G_n\ H_1\ \ldots\ H_m)\ \textbf{done}\text{”} \\
& \mathcal{G}[\![t_1]\!]\ \cdots\ \mathcal{G}[\![t_{k-1}]\!]\ \mathcal{G}[\![t_k[[r_1,\ldots,r_{v'}],[E_1,\ldots,E_h]]]\!] \\
= & \mathcal{G}[\![\mathcal{A}[\![j,F_1=F_2]\!]]\!]^{=}_{F_2}\ \mathcal{G}[\![t_1]\!]\ \cdots\ \mathcal{G}[\![t_{k-1}]\!]\ \mathcal{G}[\![t_k[[r_1,\ldots,r_{v'}],[E_1,\ldots,E_h]]]\!] \\
= & \mathcal{G}[\![\bar{t}_1]\!]^{=}_{F_2}\ \mathcal{G}[\![\bar{t}_2]\!]\ \cdots\ \mathcal{G}[\![\bar{t}_{\bar{k}-1}]\!]\ \mathcal{G}[\![\bar{t}_{\bar{k}}[[r_1,\ldots,r_{v'}],[E_1,\ldots,E_h]]]\!]
\end{array}
$$

Hence property 5 also holds. Moreover, we see that the statement “j **done**” is improved to the statement “$= F_2$ **using**... **done**” when it occurs in an equality chain.

□

6 Conclusions

In this paper we study the compilation of declarative scripts into proof terms, and the opposite translation of proof terms into declarative scripts. The study is done on the declarative language of the Matita interactive theorem prover and on proof terms for a sub-calculus of the Calculus of (Co)Inductive Constructions (CIC) used in Matita. The actual implementation in Matita already considers a larger calculus that comprises, for instance, fully general inductive types. However, regeneration of declarative scripts on fully general inductive types currently fails to round-trip. This is due to two technical limitations that will be lifted in the next major version of the system and that are linked to the representation of elimination principles and case analysis in Matita. The former is done by automatically defining a theorem for each inductive type t that is called t`_ind`$_P$ and that generalizes the `nat_ind`$_P$ constant we consider in the paper. For historical reasons, in the hypotheses of the theorem the arguments of the constructors are interleaved with the relative inductive hypotheses. For instance, the second hypothesis of the elimination theorem over binary trees B is

$$\forall b_1 : B.P\ b_1 \Rightarrow \forall b_2 : B.P\ b_2 \Rightarrow P\ (\texttt{Node}\ b_1\ b_2)$$

where `Node` is the constructor for the tree nodes. Thus, it is not possible to extend the semantics $\mathcal{C}[\![\cdot]\!]$ of “**we proceed by induction on** term **to prove** prop” and of “**case** `Node` (id$_1$**:**type) (id$_2$**:**type)” since the two λ-abstractions generated by the former command to inhabit $\forall b_1$ and $\forall b_2$ and the two Λ-abstractions generated by the latter to inhabit the implications need to be interleaved in the proof term. Creating some β-redexes it is possible to obtain the wanted effect, but then the redexes prevent a correct regeneration of the script, that will contain commands that correspond to the redexes. The solution simply consists in declaring the elimination theorem with the alternative type

$$\forall b_1, b_2 : B.P\ b_1 \Rightarrow P\ b_2 \Rightarrow P\ (\texttt{Node}\ b_1\ b_2)$$

The technical problem due to case analysis is similar: the semantics of “**we proceed by cases on** term **to prove** prop” must instantiate in Matita one occurrence of the case analysis operator that is used in CIC in place of a case analysis constant. Then the “**case** `Node` (id$_1$**:**type) (id$_2$**:**type)” command must bind the two variables b_1 and b_2 in the branches of the case analysis operator. However, the current implementation of Matita does not allow to represent case analysis operators before the binding phase,

which is instead allowed in Coq that implements the same calculus. Both technical limitations will be lifted in the next major version of Matita.

We observe that the translation from declarative scripts to declarative scripts via proof terms respects the initial script structure and can even improve it by fixing misuses of statements. Moreover this (double) translation is idempotent. It is an open question whether the same results can be achieved for more complex declarative languages whose statements could alter partial proof terms in a non structural way. Our understanding is that this is the case at least for the proof language presented in [18].

Exportation of formalised results between proof assistants having the same proof terms but different high level proof languages is an immediate application of our technique. Another obvious application is the translation of procedural scripts into executable declarative scripts, for instance for their use in education. This way it is possible to present to students or mathematicians, who better understand the declarative language, proofs found in the procedural style.

The proposed justification sub-language is currently less elaborated than the corresponding one in Isar and Mizar. Nevertheless an extension of the proposed approach to more elaborated justification sub-languages is certainly feasible, at the price of changing the improvement map. We note, however, that the interest in the round-tripping theorem for declarative scripts decreases when the improvement map starts dropping too much user-provided structure and information. Thus the study of improvement maps for elaborated justification sub-languages is an interesting future research question that we believe could be driven again by the analysis of justification languages in terms of $\bar{\lambda}\mu\tilde{\mu}$-calculus terms.

References

1. The Coq Development Team: The Coq proof assistant reference manual (2005)
2. Asperti, A., Sacerdoti Coen, C., Tassi, E., Zacchiroli, S.: User interaction with the Matita proof assistant. Journal of Automated Reasoning **39**(2), 109–139 (2007)
3. Sacerdoti Coen, C.: Explanation in natural language of lambda-bar-mu-mu-tilde-terms. In: A. Asperti, B. Buchberger, J.H. Davenport (eds.) Post-Proceedings of the Fourth International Conference on Mathematical Knowledge Management, MKM 2005, *Lecture Notes in Computer Science*, vol. 3863, pp. 234–249. Springer-Verlag (2006)
4. Autexier, S., Sacerdoti Coen, C.: A formal correspondence between omdoc with alternative proofs and the lambda-bar-mu-mu-tilde-calculus. In: Proceedings of Mathematical Knowledge Management 2006, *Lectures Notes in Artificial Intelligence*, vol. 4108, pp. 67–81. Springer-Verlag (2006)
5. Guidi, F.: Procedural Representation of CIC Proof Terms. In this issue of the Journal of Automated Reasoning
6. Christophe Raffalli: The Proof checker Documentation, version 0.85, manual of the PhoX Proof Assistant (2005)
7. Wenzel, M.: Isar - a generic interpretative approach to readable formal proof documents. In: Theorem Proving in Higher Order Logics, *LNCS*, vol. 1690, pp. 167–184. Springer (1999)
8. Wenzel, M., Wiedijk, F.: A comparison of Mizar and Isar. J. Autom. Reasoning **29**(3-4), 389–411 (2002)
9. Sacerdoti Coen, C., Tassi, E., Zacchiroli, S.: Tinycals: step by step tacticals. In: Proceedings of User Interface for Theorem Provers 2006, *Electronic Notes in Theoretical Computer Science*, vol. 174, pp. 125–142. Elsevier Science (2006)
10. Harrison, J.: A Mizar Mode for HOL. In: J. von Wright, J. Grundy, J. Harrison (eds.) Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96, *LNCS*, vol. 1125, pp. 203–220. Springer-Verlag (1996)

11. Sacerdoti Coen, C.: Tactics in modern proof-assistants: the bad habit of overkilling. In: Supplementary Proceedings of the 14th International Conference TPHOLS 2001, pp. 352–367 (2001)
12. Curien, P.L., Herbelin, H.: The duality of computation. In: ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, pp. 233–243. ACM Press, New York, NY, USA (2000). DOI http://doi.acm.org/10.1145/351240.351262
13. Kirchner, F., Sacerdoti Coen, C.: The Fellowship super-prover. `http://www.lix.polytechnique.fr/Labo/Florent.Kirchner/fellowship/`
14. Kirchner, F.: Interoperable proof systems. Ph.D. thesis, École Polytechnique (2007)
15. Asperti, A., Loeb, I., Sacerdoti Coen, C.: Stylesheets to intermediate representation and presentational stylesheets. MoWGLI Report D2d,D2f (2003)
16. Asperti, A., Tassi, E.: An interactive driver for goal directed proof strategies. In: Proc. of User Interfaces for Theorem Provers 2008. Montreal, CA, August 2008 (2009). To be published
17. Coscoy, Y., Kahn, G., Thery, L.: Extracting Text from Proofs. Technical Report RR-2459, Inria (Institut National de Recherche en Informatique et en Automatique), France (1995)
18. Corbineau, P.: A declarative proof language for the Coq proof assistant. In: TYPES 2007: Types for Proof and Programs, *LNCS*, vol. 4941. Springer-Verlag (2008)