Declarative Representation of Proof Terms

Claudio Sacerdoti Coen*

Department of Computer Science, University of Bologna
sacerdot@cs.unibo.it

Abstract. We present a declarative language inspired by the pseudo-natural language 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.

1 Introduction

In modern interactive theorem provers, proofs are likely to have several alternative representation inside the system. For instance, in 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 [11] 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.

It is then natural to investigate the translations between the different representations, wondering how much proof structure can be preserved in the translations. In [9] 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 [3] 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 still lacks the 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 an extension of System F 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.

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

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 [**?**] Ferruccio Guidi investigates the translation between proof terms of the Calculus of (Co)Inductive Constructions and a subset of the procedural language of Matita. 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 all the translations, which is the proof terms language.

An immediate application of this investigation, also explored in [**?**], 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 [12] and Mizar [13] we do not address the (sub-)language for justification of proof steps. This is left to future work. Right now justifications are either omitted (and provided by automation) 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 [10]. 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 [6].

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 terms 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. Gory 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 [8].

Some of the requirements, in particular the preservation of the structure of the user provided text, seem quite difficult to obtain. In [9] 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 simply 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 [5] we investigated as proof terms in [9,3]. 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 [7] (joint work with Florent Kirchner).

Matita proof terms are λ-terms of the Calculus of (Co)Inductive Constructions (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 System F obtained by adding explicit type conversions, local definitions and local proofs. The distinction between the last two corresponds to the distinction between λ and Λ in System F. We will present the proof terms for this fragment in Section 4.

assume id **:** type [**that is equivalent to** type]
suppose prop [(id)] [**that is equivalent to** prop]
let id **:=** term
by just **we proved** prop (id) [**that is equivalent to** prop]
by just **we proved** prop [**that is equivalent to** prop] **done**
by just **done**
by just **let** id **:** type **such that** prop (id)
by 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 **by** just [**done**]
obtain id term rel term **by** just [**done**]
rel term **by** just [**done**]

Non terminals:

id	identifiers	term	inhabitants of data types
type	data types	just	justifications, e.g. proof terms or automation
prop	propositions	rel	transitive relations (e.g. $=, \leq, <$)

Table 1. Syntax

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 [1]. It is also a super-set of the language proposed in [9] and studied also in [3]. The sub-language for justifications has not been developed yet. Thus currently a justification is either provided as a proof term or it is omitted and recovered by automation. Because of the lack of the sub-language for justifications we post-pone a comparison with other declarative languages.

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 normalizable 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.

We now present informally the semantics of the proposed language statements, whose syntax is summarised in Table 1.

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. Example:

we need to prove $\forall x : T.P(x)$
  assume x: carrier of T that is equivalent to $N \times N$
  let $y := \pi_1(x)$ *(* first projection *)*
  ...
  by _ **we proved** $P(x)$
done

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*

Introduced in the context a new local definition.

by *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*.

by *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.

by *just* **done**

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

by *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*.

by *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. Example:

we proceed by induction on n **to prove** $P(n)$
case 0
  the thesis becomes $P(0)$
  ...
case $S(m : nat)$
  by induction hypothesis we know $P(m)$
  the thesis becomes $P(S(m))$
  ...

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

obtain H
  $(x+y)^2 = (x+y)(x+y)$ **by** _
  $= x(x+y) + y(x+y)$ **by** *distributivity*
  $= x^2 + xy + yx + y^2$ **by** *distributivity*
  $= x^2 + 2xy + y^2$ **by** _
done

Types		
T	$::= T \to T$	function space
	$\mid$ nat	basic type
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
	$\mid O(E_1, \ldots, E_n)$	n-ary function
	$\mid$?	placeholder
Proof terms		
t	$::= \lambda x : T.t$	function
	$\mid \Lambda H : P.t$	type abstraction
	$\mid$ let $x := E$ in t	local definition
	$\mid$ Let $H : P := t$ 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 with 0 or more arguments; H is a bound variable ranging over proof terms or one of the constants below
	$\mid \text{and\_elim}_{P,P,P}$	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

Table 2. Proof term syntax

4 Formal Semantics

We now show the formal semantics of our language in terms of compilation of a declarative script to a proof term. In Tables 2 and 3 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.

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

Proof term typing rules.

$$\frac{\Gamma \vdash H : \forall x_1 : T_1 . \ldots . \forall x_n : T_n . P_1 \Rightarrow \ldots \Rightarrow P_m \Rightarrow P \quad \Gamma \vdash E_i : T_i \;\; \forall i \in \{1, \ldots, n\} \qquad \Gamma \vdash H_i : P_i\{E_1/x_1 \; ; \; \ldots \; ; \; E_n/x_n\} \;\; \forall i \in \{1, \ldots, m\}}{\Gamma \vdash (H \; E_1 \; \ldots \; E_n \; H_1 \; \ldots \; H_m) : P}$$

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

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

We also assume the following constant schemes (that are always supposed to be applied to arguments in β-long normal form):

$\text{and\_elim}_{P_1,P_2,P_3}$:	$P_1 \wedge P_2 \Rightarrow (P_1 \Rightarrow P_2 \Rightarrow P_3) \Rightarrow P_3$
$\text{ex\_elim}_{T,P_1,P_2}$:	$(\exists x : T . P_1) \Rightarrow (\forall x : T . P_1 \Rightarrow P_2) \Rightarrow P_2$
nat_ind_P :	$\forall n : \text{nat} . P(0) \Rightarrow (\forall m : \text{nat} . P(m) \rightarrow P(S(m))) \Rightarrow P(n)$
nat_cases_P :	$\forall n : \text{nat} . P(0) \Rightarrow (\forall m : \text{nat} . P(m)) \Rightarrow P(n)$
eq_transitive :	$\forall x, y, z : \text{nat} . x = y \Rightarrow y = z \Rightarrow x = z$

Table 3. Proof term typing rules (standard well-formed conditions on expressions, contexts, and types omitted)

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 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 placeholder.

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 → 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. 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.

The semantic function $\mathcal{C}[\![\cdot]\!]$ shown in Table 4 maps statements to functions from partial proof terms to partial proof terms. $\mathcal{C}[\![\cdot]\!]^\star$ 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 $(\mathcal{S}, P)$.

$$\mathcal{C}[\![\cdot]\!] : \text{statement} \to \text{partial\_proof} \to \text{partial\_proof}$$

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

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

For instance, 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 π (i.e. that $\mathcal{C}[\![\mathcal{S}]\!]^\star([x : \text{nat} \vdash P(x)], [], \Pi) = ([], [], ([], []) \mapsto \Pi([\pi], []))$)

We have:

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

"**obtain** $H\ E_1 = E_2$" is the only statement whose semantics 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.

Table 4: Formal semantics

$$\begin{aligned}
&\mathcal{C}[\![\textbf{assume } x : T]\!](\Gamma \vdash \forall x : T.P :: \Sigma, \Sigma', \Pi) = \\
&\quad ((\Gamma\ ;\ x : T \vdash P) :: \Sigma, \Sigma', (hd :: tl, l) \mapsto \Pi((\lambda x : T.hd) :: tl, l)) \\
&\mathcal{C}[\![\textbf{assume } x : T_1 \textbf{ that is equivalent to } T_2]\!](\Gamma \vdash \forall x : T_1.P :: \Sigma, \Sigma', \Pi) = \\
&\quad ((\Gamma\ ;\ x : T_2 \vdash P) :: \Sigma, \Sigma', \\
&\qquad \begin{cases} (hd : P' \equiv P) :: tl, l) \mapsto \Pi((\lambda x : T_2.hd : \forall x : T_2.P' \equiv \forall x : T_1.P) :: tl, l) \\ (hd :: tl, l) \qquad\qquad\quad \mapsto \Pi((\lambda x : T_2.hd : \forall x : T_2.P \equiv \forall x : T_1.P) :: tl, l) \end{cases} \\
&\mathcal{C}[\![\textbf{suppose } P_1\ (H)]\!](\Gamma \vdash \forall P : P_1.P_2 :: \Sigma, \Sigma', \Pi) = \\
&\quad ((\Gamma\ ;\ H : P_1 \vdash P_2) :: \Sigma, \Sigma', (hd :: tl, l) \mapsto \Pi((\Lambda H : P_1.hd) :: tl, l)) \\
&\mathcal{C}[\![\textbf{suppose } P_1\ (H) \textbf{ that is equivalent to } P_2]\!](\Gamma \vdash \forall H : P_1.P :: \Sigma, \Sigma', \Pi) = \\
&\quad ((\Gamma\ ;\ H : P_2 \vdash P) :: \Sigma, \Sigma',
\end{aligned}$$

$$\begin{cases} ((hd : P_3 \equiv P) :: tl, l) \mapsto \Pi((\Lambda H : P_2.hd : P_2 \to P_3 \equiv P_1 \to P) :: tl, l) \\ (hd :: tl, l) \qquad\qquad\quad \mapsto \Pi((\Lambda H : P_2.hd : P_2 \to P \equiv P_1 \to P) :: tl, l) \end{cases}$$

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

$\mathcal{C}[\![\textbf{by } j \textbf{ we proved } P_1\ (H)]\!](\Gamma \vdash P_2 :: \Sigma, \Sigma', \Pi) =$
$((\Gamma\ ;\ H : P_1 \vdash P_2) :: \Sigma, \Sigma', (hd :: tl, l) \mapsto \Pi((\text{Let } H : P_1 := j \text{ in } hd) :: tl, l))$

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

$\mathcal{C}[\![\textbf{by } j \textbf{ we proved } P \textbf{ done}]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) =$
$(\Sigma, \Sigma', (tl, l) \mapsto \Pi(j :: tl, l))$

$\mathcal{C}[\![\textbf{by } j \textbf{ we proved } P_1 \textbf{ that is equivalent to } P_2 \textbf{ done}]\!](\Gamma \vdash P_2 :: \Sigma, \Sigma', \Pi) =$
$(\Sigma, \Sigma', (tl, l) \mapsto \Pi((j : P_1 \equiv P_2)) :: tl, l))$

$\mathcal{C}[\![\textbf{by } j \textbf{ done}]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) = (\Sigma, \Sigma', (tl, l) \mapsto \Pi(j :: tl, l))$

$\mathcal{C}[\![\textbf{by } j \textbf{ let } x : T \textbf{ such that } P_1\ (H)]\!](\Gamma \vdash P_2 :: \Sigma, \Sigma', \Pi) =$
$((\Gamma\ ;\ x : T\ ;\ H : P_1 \vdash P_2) :: \Sigma, \Sigma',$
$(hd :: tl, l) \mapsto \Pi((\text{ex\_elim}_{T, P_1, P_2} j\ (\lambda x : T.\Lambda H : P_1.hd))) :: tl, l))$

$\mathcal{C}[\![\textbf{by } j \textbf{ we have } P_1\ (H_1) \textbf{ and } P_2\ (H_2)]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) =$
$((\Gamma\ ;\ H_1 : P_1\ ;\ H_2 : P_2 \vdash P) :: \Sigma, \Sigma',$
$(hd :: tl, l) \mapsto \Pi((\text{and\_elim}_{P_1, P_2, P}\ j\ (\Lambda H_1 : P_1.\Lambda H_2 : P_2.hd))) :: tl, l))$

$\mathcal{C}[\![\textbf{we need to prove } P]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) =$
$((\Gamma \vdash P) :: \Sigma, \Sigma', (hd :: tl, l) \mapsto \Pi((hd) :: tl, l))$

$\mathcal{C}[\![\textbf{we need to prove } P_1 \textbf{ or equivalently } P_2]\!](\Gamma \vdash P_1 :: \Sigma, \Sigma', \Pi) =$
$((\Gamma \vdash P_2) :: \Sigma, \Sigma', (hd :: tl, l) \mapsto \Pi((hd : P_2 \equiv P_1)) :: tl, l))$

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

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

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

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

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

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

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

$\mathcal{C}[\![= E_2 \textbf{ by } j \textbf{ done}]\!](\Gamma \vdash E_1 = E_2 :: \Sigma, \Sigma', \Pi) = (\Sigma, \Sigma', (tl, l) \mapsto \Pi(j :: tl, l))$

$\mathcal{C}[\![\textbf{we proceed by induction on } n \textbf{ to prove } P(n)]\!](\Gamma \vdash P(n) :: \Sigma, \Sigma', \Pi) =$

$((\Gamma \vdash P(O)) :: (\Gamma \vdash \forall m : \mathrm{nat}.P(m) \Rightarrow P(S(m))) :: \Sigma, \Sigma',$
$\quad (hd_1 :: hd_2 :: l, l') \mapsto \Pi((\mathrm{nat\_ind}_P\ n\ hd_1\ hd_2) :: l, l'))$
$\mathcal{C}[\![\textbf{we proceed by cases on } n \textbf{ to prove } P(n)]\!](\Gamma \vdash P(n) :: \Sigma, \Sigma', \Pi) =$
$\quad ((\Gamma \vdash P(O)) :: (\Gamma \vdash \forall m : \mathrm{nat}.P(S(m))) :: \Sigma, \Sigma',$
$\quad (hd_1 :: hd_2 :: l, l') \mapsto \Pi((\mathrm{nat\_cases}_P\ n\ hd_1\ hd_2) :: l, l'))$
$\mathcal{C}[\![\textbf{case } H\ arg_1 \cdots arg_n]\!] = \quad \mathcal{C}[\![arg_1]\!]_a\ \cdots \mathcal{C}[\![arg_n]\!]_a$
$\mathcal{C}[\![(x : T)]\!]_a = \mathcal{C}[\![\textbf{assume } x : T]\!]$
$\mathcal{C}[\![(H : P)]\!]_a = \mathcal{C}[\![\textbf{suppose } P\ (H)]\!]$
$\mathcal{C}[\![\textbf{by induction hypothesis we know } P\ (H)]\!] = \mathcal{C}[\![\textbf{suppose } P\ (H)]\!]$
$\mathcal{C}[\![\textbf{by induction hypothesis we know } P_1\ (H) \textbf{ that is equivalent to } P_2]\!] =$
$\quad \mathcal{C}[\![\textbf{suppose } P_1\ (H) \textbf{ that is equivalent to } P_2]\!]$
$\mathcal{C}[\![\textbf{the thesis becomes } P]\!] = \mathcal{C}[\![\textbf{we need to prove } P]\!]$
$\mathcal{C}[\![\textbf{the thesis becomes } P_1 \textbf{ or equivalently } P_2]\!] =$
$\quad \mathcal{C}[\![\textbf{we need to prove } P_1 \textbf{ or equivalently } P_2]\!]$

5 Natural Language Generation

We present in Table 5 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.

Table 5: Natural language generation

$\mathcal{G}[\![\lambda x : T.t]\!] = \textbf{assume } x : T\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![\Lambda H : P.t]\!] = \textbf{suppose } P\ (H)\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![let\ x := E\ in\ t]\!] = \textbf{let } x := E\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![(\lambda x : T_2.t : \forall x : T_2.P \equiv \forall x : T_1.P)]\!] =$
$\quad \textbf{assume } x : T_1 \textbf{ that is equivalent to } T_2\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![(\Lambda H : P_2.t : P_2 \Rightarrow P \equiv P_1 \Rightarrow P)]\!] =$
$\quad \textbf{suppose } P_1\ (H) \textbf{ that is equivalent to } P_2\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![(\lambda x : T_2.t : \forall x : T_2.P_2 \equiv \forall x : T_1.P_1)]\!] =$
$\quad \textbf{assume } x : T_1 \textbf{ that is equivalent to } T_2$
$\quad \textbf{we need to prove } P_1 \textbf{ or equivalently } P_2\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![(\Lambda H : P_2.t : P_2 \Rightarrow P_4 \equiv P_1 \Rightarrow P_3)]\!] =$
$\quad \textbf{suppose } P_1\ (H) \textbf{ that is equivalent to } P_2$
$\quad \textbf{we need to prove } P_3 \textbf{ or equivalently } P_4\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![Let\ K : P := (H\ E_1 \ldots E_n\ H_1 \ldots H_m)\ in\ t]\!] =$
$\quad \textbf{by } (H\ E_1 \ldots E_n\ H_1 \ldots H_m) \textbf{ we proved } P\ (K)\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![Let\ K : P_2 := ((H\ E_1 \ldots E_n\ H_1 \ldots H_m) : P_1 \equiv P_2)\ in\ t]\!] =$
$\quad \textbf{by } (H\ E_1 \ldots E_n\ H_1 \ldots H_m) \textbf{ we proved } P_1\ (K)$
$\quad \textbf{that is equivalent to } P_2\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![Let\ H : P_2 := (t_1 : P_1 \equiv P_2)\ in\ t_2]\!] =$

we need to prove $P_2\ (H)$ **or equivalently** $P_1\ \mathcal{G}[\![t_1]\!]\ \mathcal{G}[\![t_2]\!]$

$\mathcal{G}[\![(\text{eq\_transitive}\ E'_1\ E'_2\ E'_3\ (H\ E_1 \dots E_n\ H_1 \dots H_m)\ t_{2,3})]\!] =$

conclude $E'_1 = E'_2$ by $(H\ E_1 \dots E_n\ H_1 \dots H_m)\ \mathcal{G}[\![t_{2,3}]\!]^{=}_{E'_3}$

$\mathcal{G}[\![Let\ H : E'_1 = E'_3 :=$

$(\text{eq\_transitive}\ E'_1\ E'_2\ E'_3\ (H\ E_1 \dots E_n\ H_1 \dots H_m)\ t_{2,3})\ in\ t]\!] =$

obtain $H\ E'_1 = E'_2$ **by** $(H\ E_1 \dots E_n\ H_1 \dots H_m)\mathcal{G}[\![t_{2,3}]\!]^{=}_{E'_3}$

$\mathcal{G}[\![Let\ H : P := t_1\ in\ t_2]\!] =$ **we need to prove** $P\ (H)\ \mathcal{G}[\![t_1]\!]\ \mathcal{G}[\![t_2]\!]$

$\mathcal{G}[\![(H\ E_1 \dots E_n\ H_1 \dots H_n)]\!]^{=}_{E'} =$

$= E'$ **by** $(H\ E_1 \dots E_n\ H_1 \dots H_n)$ **done**

$\mathcal{G}[\![(\text{eq\_transitive}\ E'_1\ E'_2\ E'_3\ (H\ E_1 \dots E_n\ H_1 \dots H_m)\ t_{2,3})]\!]^{=}_{E'_3} =$

$= E_2$ **by** $(H\ E_1 \dots E_n\ H_1 \dots H_m)\mathcal{G}[\![\ t_{2,3}]\!]^{=}_{E'_3}$

$\mathcal{G}[\![(\text{ex\_elim}\ t\ P_1\ P_2\ H\ (\lambda x : T.\Lambda H_2 : P_2.t))]\!] =$

by H **let** $x : T$ **such that** $P_2\ (H_2)\ \mathcal{G}[\![t]\!]$

$\mathcal{G}[\![(\text{and\_elim}\ P_1\ P_2\ P_3\ H\ (\Lambda H_1 : P_1.\Lambda H_2 : P_2.t))]\!] =$

by H **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

$\mathcal{G}[\![t_2]\!]$

$\mathcal{G}[\![((H\ E_1 \ldots E_n\ H_1 \ldots H_m) : P_1 \equiv P_2)]\!] =$

by $(H\ E_1 \ldots E_n\ H_1 \ldots H_m)$ **we proved** P_1

that is equivalent to P_2 **done**

$\mathcal{G}[\![(H\ E_1 \ldots E_n\ H_1 \ldots H_m)]\!] =$ **by** $(H\ E_1 \ldots E_n\ H_1 \ldots H_m)$ **done**

The following important theorem shows that the proof term generated 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).
$\forall \Gamma, \forall P, \forall t$ *such that* $\Gamma \vdash t : P$, $\forall \Sigma, \Sigma', \Pi$ *we have*

$\mathcal{C}[\![\mathcal{G}[\![t]\!], P]\!]_s((\Gamma \vdash P) :: \Sigma, \Sigma', \Pi) = (\Sigma, \Sigma'', \Pi')$

and

either $\Sigma' = \Sigma''$ *and* $\forall l, l', \Pi(t :: l, l') = \Pi'(l, l')$

or $\Sigma' = (\Gamma' \vdash \text{nat}) :: \Sigma''$ *and* $\exists E, \forall l, l', \Pi(t :: l, E :: l') = \Pi'(l, l')$.

The two branches of the statement deserve an explanation. They differ in whether a placeholder for expressions can be closed. This is the case only if $\mathcal{G}[\![t]\!]$ syntactically contains the last step of a rewriting chain and if one placeholder occurs in Π (and, consequently, Σ' is not empty). The existentially quantified placeholder instantiation E is proved to be the right hand side of the last step in the equality chain.

Proof. The proof is by structural induction on t. We only show one significant case.

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, Σ', Π. We have

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

$= \mathcal{C}[\![\mathcal{G}[\![t']\!]]\!](((\Gamma\ ;\ x : T_2 \vdash P_2) :: \Sigma, \Sigma',$
$\quad (hd :: tl, l) \mapsto \Pi((\lambda x : T_2.hd : \forall x : T_2.P_2 \equiv \forall x : T_1.P_1) :: tl, l)))$
$= (\Sigma, \Sigma', (l, l') \mapsto \Pi((\lambda x : T_2.t' : \forall x : T_2.P_2 \equiv \forall x : T_1.P_1) :: l, l')$

The last identity is justified by the inductive hypothesis on t'.

Thus $\Sigma'' = \Sigma'$ and we have to prove the "either" part of the thesis i.e. $\forall l, l', \Pi((\lambda x : T_2.t' : \forall x : T_2.P_2 \equiv \forall x : T_1.P_1) :: l, l') = \Pi((\lambda x : T_2.t' : \forall x : T_2.P_2 \equiv \forall x : T_1.P_1) :: l, l')$, which is trivially true. □

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 relation $\preceq$ (not formally described here due to space constraints). It 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

$$\text{“}\textbf{by}\ \Lambda H : A.H\ \textbf{done}\text{”} \preceq \text{“}\textbf{suppose}\ A\ (H)\text{”}\ \ \text{“}\textbf{by}\ H\ \textbf{done}\text{”}$$

2) replacing statements with other statements that are more appropriate with respect to the context and have the same semantics. For instance the formal semantics of Table 4 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 relation $\preceq$ also captures the notion of "being less appropriate then". For instance

$$\begin{array}{c} \text{“}\textbf{conclude}\ E_1 = E_2\ \textbf{by}\ j_1\text{”}\ \ \text{“}\textbf{by}\ j_2\ \textbf{done}''\\ \preceq \text{“}\textbf{conclude}\ E_1 = E_2\ \textbf{by}\ j_1\text{”}\ \ \text{“} = E_3\ \textbf{by}\ j_2\ \textbf{done}\text{”} \end{array} \quad (1)$$

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

Note that the relation $\preceq$ is not an order relation since it is reflexive only on scripts that cannot be improved (i.e. only on scripts that have reached the fixpoint).

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

Theorem 2 (Round-tripping from declarative scripts).
$\forall S_1, \ldots, S_n, \forall \Sigma, \Sigma', \Pi$, *if* $\mathcal{C}[\![S_1\ \cdots\ S_n]\!]^\star(\Sigma, \Sigma', \Pi) = (\Sigma_1, \Sigma'_1, \Pi')$ *where* $\Sigma_1 = \Sigma_2 @ \Sigma$ *(*Σ_2 *of length* k*) then* $\exists! C$ *such that*
either $\Sigma' = \Sigma'_1$ *and* $\forall l, l', l_1, l'_1, \Pi(C[l_1, l'_1] :: l, l') = \Pi'(l_1 @ l, l'_1 @ l')$
or $\Sigma' = E :: \Sigma'_1$ *and* $\forall l, l', l_1, l'_1, \Pi(C[l_1, l'_1] :: l, E :: l') = \Pi'(l_1 @ l, l'_1 @ l')$
and $\forall t_1, \ldots, t_k, \forall E_1, \ldots, E_h$
if $\mathcal{G}[\![C[t_1, \ldots, t_k, E_1, \ldots, E_h]]\!] = S'_1 \cdots S'_m$ *then*
$S_1\ \cdots\ S_n\ \mathcal{G}[\![t_1]\!]\ \cdots\ \mathcal{G}[\![t_k]\!] \preceq S'_1\ \cdots\ S'_m$.

The second branch in the statement correspond to the case where one S_i is "**obtain** H $E_1 = E_2$" and the equality chain is not terminated in $S_1, \ldots, S_n$. In this case Σ' will contain a placeholder whereas Σ was empty.

Proof. The proof is by induction on n and then by structural induction on S_1. We only show one simple, but significant case (since it shows an improvement of the script).

Let $n = 2, S_1 =$ "**conclude** $E_1 = E_2$ **by** j_1", $S_2 =$ "**by** j_2 **done**".

$$\begin{aligned}
&\mathcal{C}[\![S_1\ S_2]\!]^\star((\Gamma \vdash E_1 = E_3) :: \Sigma, \Sigma', \Pi) \\
&= (\mathcal{C}[\![S_2]\!] \circ \mathcal{C}[\![S_1]\!])((\Gamma \vdash E_1 = E_3) :: \Sigma, \Sigma', \Pi) \\
&= \mathcal{C}[\![S_2]\!](\mathcal{C}[\![S_1]\!]((\Gamma \vdash E_1 = E_3) :: \Sigma, \Sigma', \Pi)) \\
&= \mathcal{C}[\![S_2]\!]((\Gamma \vdash E_2 = E_3) :: \Sigma, \Sigma', \\
&\qquad (hd :: tl, l) \mapsto \Pi(((\text{eq_transitive } E_1\ E_2\ E_3\ j_1\ hd) :: tl, l))) \\
&= (\Sigma, \Sigma', (l, l') \mapsto \Pi(((\text{eq_transitive } E_1\ E_2\ E_3\ j_1\ j_2) :: l, l'))
\end{aligned}$$

Since $\Sigma_1 = \Sigma$ and $\Sigma'_1 = \Sigma'$ we have $k = 0$ and $C[] =$ (eq_transitive E_1 E_2 E_3 j_1 j_2). Hence we need to prove: $\mathcal{G}[\![C[]]\!] = S'_1\ \ldots\ S'_m$ for some m and $S_1\ S_2 \preceq S'_1\ \ldots\ S'_m$. Now

$$\begin{aligned}
&\mathcal{G}[\![C[]]\!] \\
&= \mathcal{G}[\![(\text{eq_transitive } E_1\ E_2\ E_3\ j_1\ j_2)]\!] \\
&= \text{"}\mathbf{conclude}\ E_1 = E_2\ \mathbf{by} j_1\text{"}\ \mathcal{G}[\![j_2]\!]^{=}_{E_3} \\
&= \text{"}\mathbf{conclude}\ E_1 = E_2\ \mathbf{by} j_1\text{"}\ \text{" } = E_3\ \mathbf{by}\ j_2\ \mathbf{done}\text{"}
\end{aligned}$$

Hence the thesis by (1). □

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 (which lacks an elaborated sub-language for justifications) and on proof terms for a sub-calculus of the Calculus of (Co)Inductive Constructions used in Matita. The actual implementation in Matita already considers a larger calculus that comprises, for instance, fully general inductive types.

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 [4].

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 their use in education. This way it is possible to present to students or mathematicians, which only understand the declarative language, proofs found in the procedural style.

The work must be completed by designing the (mostly orthogonal) language for proof step justifications and by extending the translations proposed to cover the full language. Whether we will be able to be as close to the mathematical vernacular as Isar is, retaining automatic generation of declarative scripts with the current properties, is another open question.

References

1. Andrea Asperti, Iris Loeb, and Claudio Sacerdoti Coen. Stylesheets to intermediate representation and presentational stylesheets. MoWGLI Report D2d,D2f, 2003.
2. Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 2007. Special Issue on User Interfaces for Theorem Proving. To appear.
3. Serge Autexier and Claudio Sacerdoti Coen. A formal correspondence between omdoc with alternative proofs and the lambda-bar-mu-mu-tilde-calculus. In *Proceedings of Mathematical Knowledge Management 2006*, volume 4108 of *Lectures Notes in Artificial Intelligence*, pages 67–81. Springer-Verlag, 2006.
4. Pierre Corbineau. A declarative proof language for the Coq proof assistant. Poster at Bricks midterm Simposium.
5. Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 233–243, New York, NY, USA, 2000. ACM Press.
6. John Harrison. A Mizar Mode for HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *LNCS*, pages 203–220. Springer-Verlag, 1996.
7. Florent Kirchner and Claudio Sacerdoti Coen. The Fellowship super-prover. `http://www.lix.polytechnique.fr/Labo/Florent.Kirchner/fellowship/`.
8. Claudio Sacerdoti Coen. Tactics in modern proof-assistants: the bad habit of overkilling. In *Supplementary Proceedings of the 14th International Conference TPHOLS 2001*, pages 352–367, 2001.
9. Claudio Sacerdoti Coen. Explanation in natural language of lambda-bar-mu-mu-tilde-terms. In Andrea Asperti, Bruno Buchberger, and James H. Davenport, editors, *Post-Proceedings of the Fourth International Conference on Mathematical Knowledge Management, MKM 2005*, volume 3863 of *Lecture Notes in Computer Science*, pages 234–249. Springer-Verlag, 2006.
10. Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Tinycals: step by step tacticals. In *Proceedings of User Interface for Theorem Provers 2006*, volume 174 of *Electronic Notes in Theoretical Computer Science*, pages 125–142. Elsevier Science, 2006.
11. The Coq Development Team. The Coq proof assistant reference manual, 2005.
12. Markus Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *Theorem Proving in Higher Order Logics*, pages 167–184, 1999.
13. Markus Wenzel and Freek Wiedijk. A comparison of Mizar and Isar. *J. Autom. Reasoning*, 29(3-4):389–411, 2002.