Procedural Representation of CIC Proof Terms

Ferruccio Guidi*

Department of Computer Science
Mura Anteo Zamboni 7, 40127 Bologna, ITALY.
fguidi@cs.unibo.it

Abstract. In this paper we propose an effective procedure for translating a proof term of the Calculus of Inductive Constructions (CIC), which is very similar to a program written in a prototypal functional programming language, into a tactical expression of the high-level specification language of a CIC-based proof assistant like COQ [1] or MATITA [2]. As a use case, we report on our implementation of this procedure in MATITA and on the translation of 668 proofs generated by COQ 7.3.1 [3][1], from their logical representation as CIC proof terms to their high-level representation as tactical expressions of MATITA's user interface language.

1 Introduction

Proof assistants are increasingly used to build large-scale digital libraries of formalised mathematical knowledge. In principle, these libraries should allow users to benefit from a large set of basic units of knowledge (*i.e.* definitions, axioms, proofs, etc.) upon which to build up their own developments. However in practice, users often face difficulties in adapting existing formalisations to their needs and start their developments nearly from scratch, leading to duplicate work and unintegrated contributions.

The major difficulties are due to the fact that different proof assistants are very likely to use different knowledge representation formats both at the logical level and at the user interface level.

At the logical level the mathematical notions are expressed in a framework which usually is a type theory or a set theory, while at the user interface level these notions are expressed by a set of instructions (commands or tactics) that drive the proof assistant in the generation of the logical level contents. In some cases these instructions form a programming language.

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

[1] These proofs belong to a formal development of the author on λ-typed λ-calculus [4] that was finished when COQ 7.3.1 was outdated. Any user-level porting of the development to a newer version of COQ up to 8.1 failed or was unsatisfactory, so the author did not submit his proofs to COQ's library and, being a member of the MATITA development group, decided to migrate to MATITA implementing the procedure presented in this paper. We stress that the porting was done at the user level because the work in [4] is still evolving and the proofs need to be maintained accordingly.

Sharing knowledge among different proof assistants at the logical level is usually less difficult since the relationships between various logical frameworks are known and some techniques that facilitate knowledge reuse have been provided [5]. But in actual fact this is not always satisfactory because proof assistants allow to edit an existing piece of knowledge only starting from its user interface representation, so the adaptation and maintenance of this knowledge can not be performed at the logical level. Therefore sharing knowledge at the user interface level appears more desirable although much more difficult because in most cases the instructions used at this level do not have a formal semantics and their effects heavily depend on the execution context and on implementation details.

Nevertheless there is a good chance that a proof assistant can translate a knowledge item from its own logical level to its own user level by using its own interface instructions. So we propose to approach the problem of sharing knowledge at the user interface level by reducing it to the problem of sharing its counterpart at the logical level while leaving to each proof assistant the task of translating this knowledge from one level to the other.

Transforming a knowledge item from the user interface level to the logical level yields no problems apart from the possible loss of the information about the item that is not captured by its logical representation. In that event the logical representation of the item can be annotated [6].

In this paper we consider the opposite transformation in the case of the proof assistants COQ [1] and MATITA [2] when the knowledge item to be translated is a proof. This amounts to compiling a proof term of the Calculus of Inductive Constructions (CIC) [7] in a tactical expression of the user interface language used by these systems. In Section 2 we will overview CIC proof terms and we will discuss the tactical expressions we want to use in the transformation. The transformation itself will be presented in Section 3 and a use case will be outlined in Section 4. Section 5 contains our concluding remarks.

2 CIC Proof Terms and Tactical Expressions

In this section we introduce the domain and the codomain of our transformation, *i.e.* the CIC proof terms and tactical expressions we will use, while providing some related definitions and results.

Throughout the paper we assume the "Barendregt's convention" [8], *i.e.* the names of the bound variables and of the free variables are disjoint. Also notice that we will use the symbol $\equiv$ for definitional equality.

2.1 CIC Proof Terms and Related Definitions

CIC [7] is a powerful typed λ-calculus that, through the "propositions as types and proofs as terms" (PAT) interpretation [8] (*i.e.* the Curry-Howard isomorphism), serves as logical framework for some proof assistants like COQ and MATITA. In the following we recall some concepts about CIC just to set the

notation we will use in the paper (therefore this description of CIC is not meant to be complete). In particular the letters t, u, v, w will denote terms.

Here are some constructions that appear only in the CIC terms used as types:

Definition 1 (some type constructions).

1. *(type of a proposition according to the PAT interpretation)* `Prop`*;*
2. *(type of a local declaration of x of type v in t)* $\Pi x{:}v.t$*.*

The terms of CIC representing proofs are those whose type is of type `Prop` and they are built using the following constructions:

Definition 2 (proof terms).

1. *(reference to a global declaration or definition) c;*
   this includes a reference to a constructor of an (co)inductive type;
2. *(reference to a local declaration or definition) x;*
3. *(local declaration of x of type w in t)* $\lambda x{:}w.t$*;*
4. *(local definition of x as v in t)* $x{\leftarrow}v.t$*;*
5. *(application of the function t to the arguments $v_1 \cdots v_n$)* $(t\ v_1 \cdots v_n)$*;*
   when $n = 0$ it is convenient to set $(t) \equiv t$;
6. *(explicit cast of the type of t to w)* $(t : w)$*;*
7. *(type casted case analysis on v)* $(v \Rightarrow t_1 \cdots t_n : w)$*;*
   v inhabits a (co)inductive type [7] with constructors $c_1 \cdots c_n$ and t_i is the branch taken when v is an instance of c_i. Here the term w is the type of the whole expression.

Local definition by (co)recursion [7] is also available in the calculus but we do not consider it at the moment since it is rarely found inside proofs.

From now on $t[w_1 \cdots w_n/v_1 \cdots v_n]$ denotes the sequential replacement of $v_1 \cdots v_n$ with $w_1 \cdots w_n$ in t, $\Gamma \vdash t_1 \leftrightarrow t_2$ denotes the CIC conversion judgement (*i.e.* t_1 and t_2 are convertible according to the reduction rules of CIC under the premises in Γ, that can be local declarations or definitions of the form $x{:}u$ and $x{\leftarrow}v$ respectively), and $\Gamma \vdash t : u$ denotes the CIC type judgement (*i.e.* the type of t is u according to the type rules of CIC under the premises in Γ). Notice that we are not using CIC terms with meta-variables (*i.e.* placeholders).

2.2 Contents and structure of proofs

In general terms a proof has two main aspects: its contents, *i.e. what* is proved step by step, and its structure, *i.e. how* the proof is developed step by step. Evidently a representation of a proof focused on its structure is less readable for the human user, but appears to be easier to maintain. In fact adapting or maintaining a proof usually amounts to changing some aspects of its contents while trying to preserve its structure. Moreover in the perspective of identifying the common proof patterns occurring in a set of proofs, which can be useful in the design of automated proof procedures, we observe that such patterns are

more likely to concern the structure of these proofs rather than their contents. So we expect that a structure-oriented or *procedural* representation of the proofs would make these patterns more evident. On the contrary if focus our attention on proof readability at the user interface level, a contents-oriented or *declarative* representation of the proof is more desirable and effective especially if natural language rendering is exploited [9–11].

The distinction between the declarative aspect and the procedural aspect of a proof is captured at the logical level by the distinction between the parts of the proof term, *i.e.* the subterms, that denote types and the parts that do not.

2.3 Primitive Tactical Expressions

We recall that in general terms *tactical expressions* are a way of representing proofs at the user interface level. They are evaluated by the system in the context of a conjecture, *i.e.* the statement to prove (the conclusion) under a list of premises (local declarations and definitions) and the effect of the evaluation, *i.e.* the result of the expression, is the (virtual or real) construction of a proof for the conjecture. We also recall that the atomic tactical expressions are termed *tactics* and represent atomic proof steps at the user interface level.

The specification languages of COQ and MATITA include several classes of tactics each differing in the kind of information the user must provide to the system in order to perform the corresponding proof step This information appears as "arguments" (*i.e.* sub-expressions) inside the tactics themselves.

We can classify the tactics on the basis of their arguments as follows:

Definition 3 (classification of tactics according to their arguments).

1. *the* procedural arguments *are fragments of proof terms not including types; the* procedural tactics *are tactics having arguments of this kind;*
2. *the* declarative arguments *are types that, according to the PAT interpretation, may correspond to fragments of the statement to prove; the* declarative tactics *are tactics having at least one argument of this kind;*
3. *the* locative arguments *are pointers to parts of the conjecture (usually to its premises) used to localise the results of the expressions in which they appear; the* locative tactics *are tactics having arguments of this kind;*
4. *other arguments usually appear in* automated tactics*, i.e. tactics whose effect is to build fragments of proofs following some automated decision procedures; such arguments are used to tune these procedures.*

According to what we said in Subsection 2.2, if we represent a proof with declarative tactics, we obtain a view focused on its contents, while if we use procedural tactics, we obtain a view focused on its structure. Therefore in the perspective of adapting and maintaining proofs at the user interface level, we propose in this paper to limit the use of declarative tactics while pursuing the use of procedural tactics.

In the following we present the tactical expressions we will use in the transformation we are discussing, explaining their semantics in terms of their result,

so let $[\![\Gamma \vdash u, U]\!]$ be the CIC proof term produced by the evaluation of the expression U in the context of the conjecture $\Gamma \vdash u$, when the evaluation succeeds.

In the notation below we implicitly use the tactical $U;\ [W_1 \cdots W_n]$ that denotes the generalised sequential composition of $W_1 \cdots W_n$ after U.

Definition 4 (primitive tactical expressions).

1. `intro as` $x;\ [U]$ *builds a local declaration in front of the proof term resulting from the evaluation of* U*; formally we give the following semantics:*
   $[\![\Gamma \vdash \Pi x{:}v.u, \texttt{intro as}\ x;\ [U]]\!] \equiv \lambda x{:}v.[\![\Gamma.x{:}v \vdash u, U]\!]$*;*
   notice that we can expect the system to perform weak head reduction and α*-conversion on the conclusion of the conjecture if needed;*
   also notice that `intro` *is a procedural tactic according to our definition because its argument (i.e.* x*) is part of the constructed proof term;*
2. `pose` v `as` $x;\ [U]$ *builds a local definition in front of the proof term resulting from the evaluation of* U*; formally we give the following semantics:*
   $[\![\Gamma \vdash u, \texttt{pose}\ v\ \texttt{as}\ x;\ [U]]\!] \equiv x{\leftarrow}v.[\![\Gamma.x{\leftarrow}v \vdash u, U]\!]$*;*
   notice that `pose` *is a procedural tactic according to our definition because its arguments (i.e.* x *and* v*) are parts of the constructed proof term;*
3. `cut` w `as` $x;\ [U\ W]$ *is similar to* `pose` v `as` $x;\ [U]$ *but works under some restrictions; formally: if* $\Gamma \vdash w : \texttt{Prop}$ *then* $[\![\Gamma \vdash u, \texttt{cut}\ w\ \texttt{as}\ x;\ [U\ W]]\!] \equiv x{\leftarrow}[\![\Gamma \vdash w, W]\!].[\![\Gamma.x{:}w \vdash u, U]\!]$*;*
   notice that `cut` w `as` x *is a declarative tactic since* w *is a type;*
4. `apply` $t;\ [W_1 \cdots W_n]$ *builds an application in front of the proof terms resulting from the evaluation of* $W_1 \cdots W_n$*; formally we give the following semantics: if* $\Gamma \vdash t : \Pi x_1 \cdots x_n{:}w_1 \cdots w_n.u$ *and* $v_1 \equiv [\![\Gamma \vdash w_1, W_1]\!]$ *and ... and* $v_n \equiv [\![\Gamma \vdash w_n[v_1 \cdots v_{n-1}/x_1 \cdots x_{n-1}], W_n]\!]$ *then*
   $[\![\Gamma \vdash u[v_1 \cdots v_n/x_1 \cdots x_n], \texttt{apply}\ t;\ [W_1 \cdots W_n]]\!] \equiv (t\ v_1 \cdots v_n)$*;*
   notice that `apply` t *is a procedural tactic according to our definition because its argument (i.e.* t*) is part of the constructed proof term; also notice that in practice the system can infer some of the* $v_1 \cdots v_n$ *by unification, so the corresponding expression* W_i *must not be specified in the list* $[W_1 \cdots W_n]$ *that follows* `apply`*; furthermore the expressions in this list are evaluated respecting the order so when each* W_i *is evaluated the terms* $v_1 \cdots v_{i-1}$ *to place in the conclusion of the corresponding conjecture, i.e.* w_i*, are known;*
   Finally for $n = 0$ *we obtain the base case of the induction by which we are defining the tactical expressions: if* $\Gamma \vdash t : u$ *then* $[\![\Gamma \vdash u, \texttt{apply}\ t]\!] \equiv t$*;*
   Notice that in real implementations $\Gamma \vdash t : u$ *implies* $[\![\Gamma \vdash u, \texttt{apply}\ t]\!] \equiv t$ *also when* u *is a function type as an extension of the previous case.*
5. `cases` $v;\ [W_1 \cdots W_n]$ *builds a proof by cases on* v *whose branches result from the evaluation of* $W_1 \cdots W_n$*; formally we give the following semantics: if* v *belongs to an inductive type with constructors* $c_1 \cdots c_n$ *and if* $\Gamma \vdash c_i : w_i$*, then* $[\![\Gamma \vdash u, \texttt{cases}\ v;\ [W_1 \cdots W_n]]\!] \equiv (v \Rightarrow [\![\Gamma \vdash w_1, W_1]\!] \cdots [\![\Gamma \vdash w_n, W_n]\!] : u)$*;*
   notice that `cases` *is a procedural tactic according to our definition;*
6. `change` $u_2;\ [U]$ *changes the conclusion of the conjecture in which* U *is evaluated; formally if* $\Gamma \vdash u_2 \leftrightarrow u_1$ *then* $[\![\Gamma \vdash u_1, \texttt{change}\ u_2;\ [U]]\!] \equiv [\![\Gamma \vdash u_2, U]\!]$*;*
   notice that `change` *is a declarative tactic because its argument* u_2 *is a type;*

7. `change` w_2 `in` x; $[U]$ *changes the type of the local declaration of* x *in the premises of the conjecture in which* U *is evaluated; formally if* $\Gamma_1 \vdash w_2 \leftrightarrow w_1$ *then* $[\![\Gamma_1.x{:}w_1.\Gamma_2 \vdash u, \texttt{change}\ w_2\ \texttt{in}\ x;\ [U]]\!] \equiv [\![\Gamma_1.x{:}w_2.\Gamma_2 \vdash u, U]\!]$*;*
this is the locative version of the `change` *tactic.*
8. *We define* U *as "well-formed" in the context of* $\Gamma \vdash u$ *if* $[\![\Gamma \vdash u, U]\!]$ *exists.*

Some tactics of the above list are declarative. In COQ and MATITA `change` has some procedural counterparts (mainly `simplify` and `unfold`) whose results are not easily predictable, and `cut` can be replaced by `lapply` in some cases.

A result on the tactical expressions defined above is easily provable and explains in exact terms why such expressions represent proofs:

Theorem 1 (soundness).
If U *is well formed in* $\Gamma \vdash u$ *then* $\Gamma \vdash [\![\Gamma \vdash u, U]\!] : u$.

Proof. By induction on the structure of U using the type rules of CIC [1]. □

2.4 Induction Principles and Related Tactical Expressions

A CIC-based proof assistant allows to define data structures by means of inductive types and provides automatically defined lemmas proving forms of structural induction on the elements of such types. These lemmas are termed default induction principles. Induction principles have a particular shape, whose description goes beyond the scope of this paper, and the system provides specialised versions of the `apply` tactic as facilities for invoking these principles in proofs.

In particular we want to mention the following tactical expressions:

Definition 5 (tactical expressions related to induction principles).

1. *if* v *is an element of an inductive type* w *with* n *constructors and* t *is an elimination principle over* w *then* `elim` v `using` t; $[W_1 \cdots W_n]$ *has the semantics of* `apply` t; $[\cdots\ W_1 \cdots W_n\ \cdots\ \texttt{apply}\ v]$*; the extra arguments denoted by* $\cdots$ *are inferred by the system;*
2. `rewrite right` v; $[W]$ *and* `rewrite left` v; $[W]$ *are special cases of the above for two induction principles over the type denoting Leibniz equality;*
3. `elim` v `using` t `in` $y_1 \cdots y_n$ `as` x; $[U]$ *is the locative version of* `elim` *and has the semantics of* `pose` $(t\ \cdots\ y_1 \cdots y_n\ \cdots\ v)$ `as` x; $[U]$*; this represents structural induction applied in a forward reasoning manner;*
4. *the locative version of* `rewrite` *are also provided:*
`rewrite right` v `in` y `as` x; $[U]$ *and* `rewrite left` v `in` y `as` x; $[U]$.

This definition is informal because these tactical expressions are not strictly necessary since in principle they can be expressed in terms of other expressions). Nevertheless we decide to use them in our transformation since the user certainly expects to see them in the resulting description of the proof.

Moreover `elim` and `rewrite`, as implemented in MATITA, may take an argument, representing a CIC term pattern, that is not considered in the above syntax and that is involved in the construction of the resulting term.

3 From CIC Proof Terms to Tactical Expressions

The purpose of the transformation ϵ we are presenting in this section is to construct a tactical expression whose result is a given proof term. Formally speaking, given t such that $\Gamma \vdash t : u$ we seek $\epsilon(\Gamma, t)$ such that $[\![\Gamma \vdash u, \epsilon(\Gamma, t)]\!] = t$. The auxiliary precondition $\Gamma \vdash u : \texttt{Prop}$ ensures that t is really a proof term.

The expression $\epsilon(\Gamma, t)$ is computed in two steps: first we preprocess t with a proof term optimiser returning a term $\omega_1(\Gamma, t)$ such that $\Gamma \vdash \omega_1(\Gamma, t) \leftrightarrow t$, then we seek the tactical expression representing $\omega_1(\Gamma, t)$ using a function π. So in the end ϵ is π after ω_1. The subsections below illustrate these functions in detail.

3.1 CIC Proof Term Optimisation

Proof terms are optimised to improve the quality of the proofs represented at the user interface level in the perspective of an easier maintenance of these proofs.

The optimised version of a proof term t, which we denote by $\omega_1(\Gamma, t)$, is computed by repeatedly applying a number of conversion steps to t until no application is possible. The optimisation works on every sub-term of t but we apply it only to proof terms because normally a conversion performed elsewhere does not improve the the final representation of the proof significantly. Some conversion steps performed by ω_1 appear in [4] as part of a calculus termed $\lambda\delta$.

According to the "direct proof paradigm", *i.e.* the simplest proof strategy coming from standard mathematical practice, the part of a proof carried out by forward reasoning should precede the part of that proof carried out by backward reasoning. At the logical level this means that abbreviations (*i.e.* local definitions) should precede applications in the proof term (recall that the application of a lemma v in a backward reasoning manner is represented by the construction $(v\ \cdots)$ while the same lemma applied in a forward reasoning manner is represented as $x{\leftarrow}(v\ \cdots).t)$. Since in our experience exploiting this paradigm contributes to clarify the structure of the final proof, our optimisation procedure takes care of moving the abbreviations towards the top part of the proof term whenever possible while leaving the applications in the bottom part.

The sub-terms that are certainly shown as procedural arguments after the transformation π are the head of applications (appearing in `apply` expressions), the first argument of proofs by cases (appearing in `cases` expressions) and the last argument of the applications of an inductive principle (appearing in `elim` expressions or the like). In principle these sub-terms can be very complex and verbose so we choose to abbreviate them when they are not atomic (*i.e.* we perform an anticipation of these sub-terms by ζ-expansion [1]).

If t expects m formal parameters then `apply` t; $[W_1 \cdots W_n]$ can be well formed only if $n \leq m$. This means that an application of t represented using `apply` t is better handled if the number of its arguments is less or equal to m. In this paper an application with this property will be termed sober. To ensure that all applications are sober, ω_1 exploits anticipation by ζ-expansion [1].

Formally $\omega_1(\Gamma, t)$ is defined by iterating the following rules until a fixed point is reached. The main properties of ω_1 are listed below.

Definition 6 (the rules for ω_1).

1. *if t is not of sort* `Prop` *in Γ then $\omega_1(\Gamma, t) \equiv t$; else*
2. $\omega_1(\Gamma, c) \equiv c$;
3. $\omega_1(\Gamma, x) \equiv x$;
4. *if $\omega_1(\Gamma.x{\leftarrow}v, t_1) = t_2$ and x does not occur in t_2 then $\omega_1(\Gamma, x{\leftarrow}v.t_1) \equiv t_2$ (ζ-contraction of [4]); else*
5. *if v_1 is of sort* `Prop` *and $\omega_1(\Gamma, v_1) = y{\leftarrow}v.v_2$ then $\omega_1(\Gamma, x{\leftarrow}v_1.t) \equiv y{\leftarrow}v.\omega_1(\Gamma, x{\leftarrow}v_2.t)$; else*
6. *if $\omega_1(\Gamma, v) = (v_0\ \cdots\ v_1 \cdots v_{i-1}\ v_i\ v_{i+1} \cdots v_n\ \cdots)$ where v_0 is an inductive principle and $v_1 \cdots v_n$ are the proofs of the inductive cases, if v_i is not a local reference and if y is fresh in Γ then $\omega_1(\Gamma, x{\leftarrow}v.t) \equiv \omega_1(\Gamma, x{\leftarrow}y{\leftarrow}v_i.(v_0\ \cdots\ v_1 \cdots v_{i-1}\ y\ v_{i+1} \cdots v_n\ \cdots).t)$ (ζ-expansion of [1] or $\delta\zeta$-expansion of [4]); else*
7. $\omega_1(\Gamma, x{\leftarrow}v.t) \equiv x{\leftarrow}\omega_1(\Gamma, v).\omega_1(\Gamma.x{\leftarrow}v, t)$;
8. *if $\omega_1(\Gamma.x{:}w, t_1) = y{\leftarrow}v.t_2$ and x does not occur in v then $\omega_1(\Gamma, \lambda x{:}w.t_1) \equiv y{\leftarrow}v.\omega_1(\Gamma, \lambda x{:}w.t_2)$; else*
9. $\omega_1(\Gamma, \lambda x{:}w.t) \equiv \lambda x{:}\omega_1(\Gamma, w).\omega_1(\Gamma.x{:}w, t)$;
10. *if $\omega_1(\Gamma, t_1) = x{\leftarrow}v.t_2$ then $\omega_1(\Gamma, (t_1\ v_1 \cdots v_n)) \equiv x{\leftarrow}v.\omega_1(\Gamma, (t_2\ v_1 \cdots v_n))$ (υ-swap of [4]); else*
11. *if $\omega_1(\Gamma, t_1) = \lambda x{:}w.t_2$ and t_2 contains at most one free occurrence of x then $\omega_1(\Gamma, (t_1\ v\ v_1 \cdots v_n)) \equiv \omega_1(\Gamma, (t_2[v/x]\ v_1 \cdots v_n))$ (β-contraction of [1]); else*
12. *if $\omega_1(\Gamma, t_1) = \lambda x{:}w.t_2$ then $\omega_1(\Gamma, (t_1\ v\ v_1 \cdots v_n)) \equiv \omega_1(\Gamma, (x{\leftarrow}v.t_2\ v_1 \cdots v_n))$ (β-contraction of [4]); else*
13. *if $\omega_1(\Gamma, t_1) = (t_2\ v'_1 \cdots v'_m)$ then $\omega_1(\Gamma, (t_1\ v_1 \cdots v_n)) \equiv \omega_1(\Gamma, (t_2\ v'_1 \cdots v'_m\ v_1 \cdots v_n))$; else*
14. *if $\omega_1(\Gamma, t_1) = t_2$ and $\Gamma \vdash t_2 : \Pi x_1 \cdots x_m{:}w_1 \cdots w_m.u$ where u is not a function type and $m > 0$, if x is fresh in Γ and $n > 0$, then $\omega_1(\Gamma, (t_1\ v'_1 \cdots v'_m\ v_1 \cdots v_n)) \equiv \omega_1(\Gamma, x{\leftarrow}(t_1\ v'_1 \cdots v'_m).(x\ v_1 \cdots v_n))$ (ζ-expansion of [1] or $\delta\zeta$-expansion of [4]); else*
15. *if $\omega_1(\Gamma, v_i) = x{\leftarrow}v.v'_i$ and v_i is of sort* `Prop` *in Γ then $\omega_1(\Gamma, (t\ v_1 \cdots v_n)) \equiv x{\leftarrow}v.\omega_1(\Gamma, (t\ v_1 \cdots v_{i-1}\ v'_i\ v_{i+1} \cdots v_n))$; else*
16. *if $\omega_1(\Gamma, t_1) = t_2$ and $\Gamma \vdash t_2 : \Pi x_1 \cdots x_m{:}w_1 \cdots w_n.u$ where u is not a function type and $n > 0$, if t_2 is an induction principle and v_n is not atomic and x is fresh in Γ then $\omega_1(\Gamma, (t_1\ v_1 \cdots v_n)) \equiv \omega_1(\Gamma, x{\leftarrow}v_n.(t_1\ v_1 \cdots v_{n-1}\ x))$ (ζ-expansion of [1] or $\delta\zeta$-expansion of [4]); else*
17. $\omega_1(\Gamma, (t\ v_1 \cdots v_n)) \equiv (\omega_1(\Gamma, t)\ \omega_1(\Gamma, v_1) \cdots \omega_1(\Gamma, v_n))$;
18. $\omega_1(\Gamma, (t : v)) \equiv \omega_1(\Gamma, t)$ (ϵ-contraction of [4]);
19. *if $\omega_1(\Gamma, t_1) = x{\leftarrow}v.t_2$ with t_1 of sort* `Prop` *in Γ then $\omega_1(\Gamma, (t_1 \Rightarrow v_1 \cdots v_n : w)) \equiv x{\leftarrow}v.\omega_1(\Gamma, (t_2 \Rightarrow v_1 \cdots v_n : w))$*
20. *if $\omega_1(\Gamma, t_1) = (c_i\ v'_1 \cdots v'_m)$ then $\omega_1(\Gamma, (t_1 \Rightarrow v_1 \cdots v_n : w)) \equiv \omega_1(\Gamma, (v_i\ v'_1 \cdots v'_m))$ (ι-contraction of [1]); else*
21. *if $\omega_1(\Gamma, t_1) = t_2$ and t_2 is not atomic and x is fresh in Γ, then $\omega_1(\Gamma, (t_1 \Rightarrow v_1 \cdots v_n : w)) \equiv x{\leftarrow}t_2.\omega_1(\Gamma, (x \Rightarrow v_1 \cdots v_n : w))$ (ζ-expansion of [1] or $\delta\zeta$-expansion of [4]); else*
22. *if $\omega_1(\Gamma, v_i) = x{\leftarrow}v.v'_i$ and v_i is of sort* `Prop` *in Γ then $\omega_1(\Gamma, (t \Rightarrow v_1 \cdots v_n : w)) \equiv x{\leftarrow}v.\omega_1(\Gamma, (t \Rightarrow v_1 \cdots v_{i-1}\ v'_i\ v_{i+1} \cdots v_n : w))$;*

23. $\omega_1(\Gamma, (t \Rightarrow v_1 \cdots v_n : w)) \equiv (\omega_1(\Gamma, t) \Rightarrow \omega_1(\Gamma, v_1) \cdots \omega_1(\Gamma, v_n) : \omega_1(\Gamma, w))$.

Coming now to the issue of proving some properties of the function ω_1, we observe that any fully developed proof by induction on twenty-three cases can not be placed in the present paper, so the following proofs will be only sketched.

In particular we conjecture that the computation of $\omega_1(\Gamma, t)$ always terminates and that its complexity can be exponential in the size of the term t.

Theorem 2 (fixed point).

1. *(fixed point)* ω_1 *after* ω_1 *is* ω_1*;*
2. *(compatibility) if* $\Gamma \vdash t : u$ *then* $\Gamma \vdash \omega_1(\Gamma, t) : u$ *and* $\Gamma \vdash \omega_1(\Gamma, t) \leftrightarrow t$.

Proof.

1. $\omega_1(\Gamma, t)$ is defined as a fixed point;
2. ω_1 performs conversion steps that are known to preserve the type [7]. □

3.2 Representation of Optimised CIC Proof Terms

Here we present the function π that produces a tactical expression U starting from a CIC term t typable in a context Γ. If t is optimised in the sense of Subsection 3.1, we guarantee that U can be evaluated and that its result is t.

The expression U includes two sorts of proof steps: the construction steps corresponding to the constructors of t and the conversion steps that we detect by comparing the inferred type and the expected type of the subterms of t according to Coscoy's double type-inference [11]. Notice that the inferred type of a term v is always defined while the expected type of v is defined only in some cases, so when it is not defined, we set it as the inferred type of v for convenience.

The most delicate aspect of the design of π is the translation of an application $t' \equiv (t\ v_1 \cdots v_n)$ into an effective invocation of the `apply` t tactic. In particular we have to make sure that the conclusion u of the conjecture in the context of which this tactic is evaluated, can be unified with the inferred type of t'. Since the amount of conversion performed during unification depends on the particular system and is unpredictable in practice, we choose to assume that `apply` performs no conversion when unification is invoked, and we convert u by hand by an explicit invocation of the `change` tactic. Being u the expected type of t' [11], this conversion is needed only when the expected type of t' differs from the inferred type of t' (this can happen if t' is an argument of an application).

A similar problem arises when we detect a difference between the inferred type and the expected type of the argument, say v, of a construction that we want to render with a `cases` v or an `elim` v `using` $\cdots$, since evaluating such a tactic requires computing the inferred type of v. In this case v is a reference to a local premise x (after optimisation) so we invoke `change` $\ldots$ `in` x. In the other cases the explicit conversion can be safely omitted.

The rules defining the function π are given below with the rules of two auxiliary functions γ_1 and γ_2 used to handle conversion. Notice that we make no assumptions on how the inferred and expected types of a term are computed.

Definition 7 (the rules for γ_1).

1. *if the inferred type u of t differs from the expected type of t then* $\gamma_1(\Gamma, t, U) \equiv$ `change` u; $[U]$*; else*
2. $\gamma_1(\Gamma, t, U) \equiv U$.

Definition 8 (the rules for γ_2).

1. *if* $\Gamma \equiv \Gamma_1.x{:}w.\Gamma_2$ *and if w is not the inferred type of x then* $\gamma_2(\Gamma, x, U) \equiv$ `change` w `in` x; $[U]$*; else*
2. $\gamma_2(\Gamma, t, U) \equiv U$.

Definition 9 (the rules for π).

1. *if t is not of sort* `Prop` *then* $\pi(\Gamma, t) \equiv \gamma_1(\Gamma, t,$ `apply` $t)$*; else*
2. *if* $t \equiv c$ *or* $t \equiv x$ *then* $\pi(\Gamma, t) \equiv \gamma_1(\Gamma, t,$ `apply` $t)$*;*
3. $\pi(\Gamma, \lambda x{:}w.t) \equiv$ `intro as` x; $[\pi(\Gamma.x{:}w, t)]$;
4. *if* $v' \equiv (t' \cdots y_1 \cdots y_n \cdots v)$ *where t' is an induction principle and $y_1 \cdots y_n$ are the proofs of the inductive cases, if* $\Gamma \vdash v' : w'$ *and* $\Gamma \vdash w'$: `Prop` *then* $\pi(\Gamma, x{\leftarrow}v'.t) \equiv \gamma_2(\Gamma, v, \gamma_2(\Gamma, y_1, \cdots\ \gamma_2(\Gamma, y_n, U)\ \cdots))$ *for* $U \equiv$ `elim` v `using` t' `in` $y_1 \cdots y_n$ `as` x; $[\pi(\Gamma.x{:}w', t)]$ *when* $\Gamma.x{\leftarrow}v' \vdash t : t'$ *and x does not occur in t'; else*
5. *if* $v' \equiv (t' \cdots y_1 \cdots y_n \cdots v)$ *where t' is an induction principle and $y_1 \cdots y_n$ are the proofs of the inductive cases then* $\pi(\Gamma, x{\leftarrow}v'.t) \equiv \gamma_2(\Gamma, v, \gamma_2(\Gamma, y_1, \cdots\ \gamma_2(\Gamma, y_n, U)\ \cdots))$ *for* $U \equiv$ `elim` v `using` t' `in` $y_1 \cdots y_n$ `as` x; $[\pi(\Gamma.x{\leftarrow}v', t)]$*; else*
6. *If* $\Gamma \vdash v : w$ *and* $\Gamma \vdash w$: `Prop` *and* $\Gamma.x{\leftarrow}v \vdash t : t'$ *and x does not occur in t', then* $\pi(\Gamma, x{\leftarrow}v.t) \equiv$ `cut` w `as` x; $[\pi(\Gamma.x{:}w, t)\ \pi(\Gamma, v)]$*; else*
7. $\pi(\Gamma, x{\leftarrow}v.t) \equiv$ `pose` v `as` x; $[\pi(\Gamma.x{\leftarrow}v, t)]$;
8. *if* $t' \equiv (t \cdots v_1 \cdots v_n \cdots v)$ *where t is an induction principle and $v_1 \cdots v_n$ are the proofs of the inductive cases then* $\pi(\Gamma, t') \equiv \gamma_2(\Gamma, v, \gamma_1(\Gamma, t',$ `elim` v `using` t; $[\pi(\Gamma, v_1) \cdots \pi(\Gamma, v_n)]))$*; else*
9. *if* $t' \equiv (t\ v_1 \cdots v_n)$ *then* $\pi(\Gamma, t') \equiv \gamma_1(\Gamma, t',$ `apply` t; $[\pi(\Gamma, v_1) \cdots \pi(\Gamma, v_n)])$;
10. *if* $t \equiv (v \Rightarrow t_1 \cdots t_n : w)$ *then* $\pi(\Gamma, t) \equiv \gamma_2(\Gamma, v, \gamma_1(\Gamma, t, U))$*; for* $U \equiv$ `cases` v; $[\pi(\Gamma, v_1) \cdots \pi(\Gamma, v_n)]$;
11. $\pi(\Gamma, (t : w)) \equiv \pi(\Gamma, t)$*;*
12. `rewrite` *tactics are used in place of* `elim` *tactics when possible.*

We can set the following results about the transformation π:

Theorem 3 (correctness).

1. *(correctness) if* $\Gamma \vdash t : u$ *then* $[\![\Gamma \vdash u, \epsilon(\Gamma, t)]\!] = \omega_1(\Gamma, t)$*;*
2. *(round trip) if* $\Gamma \vdash t : u$ *and if* $U \equiv \epsilon(\Gamma, t)$ *then* $\epsilon(\Gamma, [\![\Gamma \vdash u, U]\!]) = U$.

Proof.

1. Once the statement is rephrased like $\Gamma \vdash t : u$ and $t' \equiv \omega_1(\Gamma, t)$ implies $[\![\Gamma \vdash u, \pi(\Gamma, t')]\!] = t'$, given that t' is optimised and that γ_1 and γ_2 do not affect the result of the expression they are applied to, the proof becomes straightforward;
2. unfolding t', clause 1 gives $[\![\Gamma \vdash u, \epsilon(\Gamma, t)]\!] = \omega_1(\Gamma, t)$ that, by Theorem 2.1, implies $\omega_1(\Gamma, [\![\Gamma \vdash u, \epsilon(\Gamma, t)]\!]) = \omega_1(\Gamma, t)$ by which we can conclude $\epsilon(\Gamma, [\![\Gamma \vdash u, \epsilon(\Gamma, t)]\!]) = \epsilon(\Gamma, t)$ meaning that ϵ after $[\![\ ]\!]$ after ϵ is ϵ. □

4 Implementation and Testing in a Real Case

In this section we discuss our own implementation of the transformation ϵ (see Section 3) in the proof assistant MATITA and the tests we used to verify it.

4.1 Implementation Issues

The current version of the proof assistant MATITA provides our transformation ϵ as an experimental feature whose implementation is still under development.

Given the uniform resource identifier (URI) of a theorem in the Hypertextual Electronic Library of Mathematics (HELM) [12], that is the digital library of MATITA, the tactical expression representing its proof is computed as follows:

Definition 10 (ϵ transformation pipeline).

1. *the proof is read from the library obtaining its representation at the logical level as a plain CIC proof term t_0, i.e. the initial version of the proof;*
2. *the transformation ω_1 (see Subsection 3.1) is applied to t_0 giving a plain CIC proof term t_1, i.e. the optimised version of t_0;*
3. *t_1 is type-checked and every sub-term is annotated with its inferred and expected types according to Coscoy's double type-inference algorithm [11]; this returns an annotated CIC proof term t_2;*
4. *the transformation π (see Subsection 3.2) is applied to t_2 giving a tactical expression T_3 represented in an ad hoc intermediate format designed by us;*
5. *T_3 is encoded into a tactical expression T_4 of Grafite, the knowledge representation language of MATITA at the user interface level; this format contains many details that are omitted in our intermediate format;*
6. *T_4, the final representation of the proof, is processed by the Grafite pretty printer and rendered as a script that the user can store or modify at will.*

We stress that the segmentation of the above pipeline decouples its transformation stages from its type checking and rendering stages allowing a factorised implementation of the whole procedure.

Currently Definition 6.6 and Definition 6.8 are not implemented while Definition 6.11 is implemented only in the case of t_2 containing no free occurrences of x; Definition 6.12 is used otherwise. Furthermore Definition 9.4 and Definition 9.5 are implemented only in the case `rewrite` tactics can be used, since MATITA does not implement the tactic `elim` $\cdots$ `in` $\cdots$ at the moment. In addition, the current implementation of `rewrite` $\cdots$ v `in` y `as` x in this system does not follow the semantics of Definition 5.3 when y refers to a local definition.

Taking this argument into account would force us to discuss the structure of inductive principles and this is not in the scope of the present paper.

Concerning Definition 9.3, we exploit a generalisation of `intro as` x; $[U]$ provided by MATITA in the form `intro as` $x_1 \cdots x_n$; $[U]$ with the semantics of a repeated `intro` before U. Concerning Definition 9.6 we use a variant of `cut` w `as` x; $[U\ W]$ provided by MATITA in the form `cut` w `as` x; [`id` W]; $[U]$

because we want to present W before U in the resulting proof. This approach forces us to introduce the `id` (identity) tactic having the meaning of a placeholder for a postponed expression. Formally: $[\![\Gamma \vdash u, \mathtt{id};\ [U]]\!] \equiv [\![\Gamma \vdash u, U]\!]$ and $[\![\Gamma \vdash u, T;\ [U_1 \cdots U_i\ \mathtt{id}\ U'_1 \cdots U'_j];\ [U]]\!] \equiv [\![\Gamma \vdash u, T;\ [U_1 \cdots U_i\ U\ U'_1 \cdots U'_j]]\!]$.

Finally, concerning Definition 9.9, we said in Subsection 2.3 that the system can infer some terms among $v_1 \cdots v_n$ by unification, but it is a matter of facts that higher order unification may fail in some cases.

To solve this inconvenient we take advantage of the possibility offered by MATITA to indicate the unifiers used by the `apply` tactic. In particular we detect the inferable functional terms among $v_1 \cdots v_n$ and we specify them as explicit unifiers in the `apply` tactical expression. Notice that although these terms appear as procedural arguments in this expression, we judge that anticipating them by ζ-expansion brings no benefit to the resulting proof.

Each proof by cases can be turned into a proof by induction carried out by applying a system-provided theorem denoting a default induction principle.

We may extend ω_1 with this transformation step for two reasons: firstly we may want to reuse the optimised proof term in those logical frameworks that support proofs by induction in place of proofs by cases, as the Minimal Type Theory [13]; secondly the set of constructions appearing in the optimised proof term is reduced. We strongly stress that this transformation is not a conversion according to CIC rules, so a term may not be well typed after this extended optimisation. Nevertheless it is well typed in the great majority of the real cases.

Definition 11 (critical optimisation steps).

1. *if $\Gamma \vdash v : w$ and c is the default induction principle of w for the sort* `Prop` *then $\omega_1(\Gamma, (v \Rightarrow t_1 \cdots t_n : w)) \equiv \omega_1(\Gamma, (c \cdots v))$.*

4.2 Testing Issues

We tested our implementation of the transformation ϵ on the 668 proofs of [4]. These proofs, originally appearing as tactical expressions of the Gallina specification language version 7 [3, 14], were processed by the proof assistant COQ, which turned them into CIC proof terms. These terms where processed by the proof assistant MATITA, which turned them into objects of HELM and they are available in this form inside the HELM directories cic:/matita/LAMBDA-TYPES/Base-1 and cic:/matita/LAMBDA-TYPES/LambdaDelta-1. The transformation ϵ was applied at this stage.

Three proofs of these were actually generated by COQ without evaluating a tactical expression[2] and make the development self-contained.

Figure 1 shows the frequency of the optimisations performed by ω_1. Definition 6.18 is never applied because the proofs do not contain any type cast construction. All proofs successfully type-check after optimisation.

Sixteen proofs do not reach stage 4 of Definition 10 because of problems in the current implementation of Coscoy's algorithm in MATITA.

[2] These proofs were produced by the GENERATE INVERSION directive of COQ 7.3.1 [3]

Optimisation	Action	Applied
Definition 6.4	information removal	250 times
Definition 6.5	definition lifting	429 times
Definition 6.10	definition lifting	2360 times
Definition 6.12	information removal	227 times
Definition 6.13	nested application	494 times
Definition 6.14	anticipation	645 times
Definition 6.15	definition lifting	3781 times
Definition 6.16	anticipation	2163 times
Definition 11.1	critical step	254 times
Any of the above		10603 times

Fig. 1. Frequency of the optimisations

Fourteen tactical expressions resulting from the transformation π applied to the remaining proofs, are not evaluated correctly by MATITA because of problems in the current implementation of the `elim` tactic.

We stress that the generated proofs should always be checked by MATITA especially if critical optimisation steps are allowed as in the case we are discussing.

Source (content)	Scripts size (type)	Tactics
Initial COQ input (668 proofs − 3)	0.4 Mbytes (Gallina)	9879
Output of COQ (668 CIC proof terms)	2.8 Mbytes (Gallina)	
Initial MATITA input (668 CIC proof terms)	4.1 Mbytes (Grafite)	
Output of π (668 proofs − 16)	2.1 Mbytes (Grafite)	51289

Fig. 2. Volume of the data

Figure 2 shows the volume of the data (in scripts size and complexity) involved in the whole transformation from initial COQ input to final π output.

Notice that the scripts are self-contained so they include the definitions not found in COQ's library. This additional content is considered in the calculation of size, but it is not considered in the calculation of complexity. Commented texts and unnecessary black characters are not considered in the computation of sizes.

Notice that the initial Gallina scripts are smaller than the final Grafite scripts both in size and in amount of tactics. This is because in the initial scripts we highly exploit automation tactics and code factorisation through macro tactics written in the LTAC language, while in the final scripts we use just primitive tactics without factorisation. In particular the scripts size increases by a factor 5.3 while the number of tactics, removing from the final scripts the `id` tactics that we could avoid and the other tactics of the 3 generated proofs (4912 all together), increases by a factor 4.7. We stress that these values are comparable with other values of the "de Bruijn loss factor" [15] found in the literature [16, 17]. This factor represents the increment in complexity occurring when the information hidden by automation and abstraction is fully displayed. The two factors we

give here can be considered the "apparent loss factor" and the "intrinsic loss factor" in the sense of [17]. Remarkably the difference of size between the final COQ output and the initial MATITA input (increment factor: 1.5) is entirely due to the greater verbosity of Grafite with respect to Gallina, since all data use the same concrete representation format. *i.e.* the ASCII-7 encoding.

It would be interesting to compare the evaluation times of the two sets of scripts once provided to the respective proof assistants (running in the same conditions) and to see if many primitive tactics are evaluated faster or slower than few high-level tactics producing the same proof terms. Unfortunately we can not perform tests like this one at the moment because COQ 7.3.1 and MATITA can not parse their own outputs in full at the moment.

4.3 A Running Example

In this section we present the result of the transformation ϵ applied to one of the 668 proofs mentioned in Subsection 4.2. Namely we consider the statement:

$$\text{theorem } le\_x\_pred\_y : \forall(y : nat).(\forall(x : nat).((lt\ x\ y) \rightarrow (le\ x\ (pred\ y))))$$

that formalises the property of the natural numbers: "$x < y$ implies $x \leq y-1$". Its HELM URI is cic:/matita/LAMBDA-TYPES/Base-1/ext/arith/le_x_pred_y.con and the constants occurring in it, *i.e.* *nat*, *lt*, *le*, *pred*, refer to entities defined in the standard library of COQ as included in HELM.

Figure 3 shows the initial proof term of the statement as produced by COQ and translated faithfully in Grafite. Notice the placeholder ?, added during the translation process, for a term that MATITA can infer.

The optimised version of the proof term is shown in Figure 4 and results from the application of the transformations described by Definition 6.10 (twice), Definition 6.13, Definition 6.14 and Definition 11.1. In particular by Definition 6.10 the construction $((\text{let } H2 \equiv \ldots \text{ in } (False\_ind\ \ldots\ H2))\ H0)$ becomes
$\text{let } H2 \equiv \ldots \text{ in } ((False\_ind\ \ldots\ H2)\ H0)$, then the nested application is detected and we get $\text{let } H2 \equiv \ldots \text{ in } (False\_ind\ \ldots\ H2\ H0)$ by Definition 6.13, then the non-sober application is detected and by Definition 6.14 we get
$\text{let } H2 \equiv \ldots \text{ in } ((\text{let } LOCAL \equiv (False\_ind\ \ldots\ H2) \text{ in } LOCAL)\ H0)$, which, by the second application of Definition 6.10, becomes
$\text{let } H2 \equiv \ldots \text{ in let } LOCAL \equiv (False\_ind\ \ldots\ H2) \text{ in } (LOCAL\ H0)$. Secondly by Definition 11.1 the construction match H in le ... (proof by cases) becomes $(le\_ind\ \ldots\ H)$ (corresponding proof by induction carried out by the application of a default induction principle). The other match constructions are not affected by this transformations because they do not represent proofs.

Figure 5 shows the Grafite proof script derived from the optimised version of the proof term. Notice that the tactics `change` and `elim` take a locative argument that denotes a conjecture pattern. This pattern contains the placeholders `%` and `?` representing the terms (or subterms) on which these tactics must act and the other terms (or subterms) respectively. Look at [2] Section 3.2 for details. Also notice that the `intros` tactic can take an `_` in place of a premise name

to mean that the introduced premise will not be used to complete the proof and is to be removed from the current conjecture context. More generally the author is working on an improved version of π that exploits the `clear` tactic to remove a premise from the current conjecture context as soon as the proof can be completed without it.

5 Conclusions and Future Work

In the previous sections we proposed an effective procedure for translating a proof term of the Calculus of Inductive Constructions (CIC), which is very similar to a program written in a prototypal functional programming language, into a tactical expression of a CIC-based proof assistant's user interface language.

This procedure allows to convert a proof encoded at the logical level and coming from any source, *i.e.* from a digital library or from another proof development system, into an equivalent proof presented in the proof assistant's editable high-level format. In particular we can improve the quality of user-provided proof scripts by regenerating them from their logical level content. In fact the scripts generated by the transformation we presented are clean (*i.e.* they contain no detours or unused information), they are specifically designed to be easily maintained and in the end they may contain many useful optimisations that perhaps the user would not introduce systematically in hand-written proof scripts, especially in the context of a large-scale formal development.

As a use case, we reported on our implementation of the procedure in the system MATITA [2] and on the translation of 668 proofs generated by the system COQ 7.3.1 [3], from their logical representation as CIC proof terms to their high-level representation as tactical expressions of MATITA's user interface language.

We noticed that the comparison between the initial COQ scripts producing the proofs and the final MATITA scripts resulting from the conversion, gives an increment factor in size and complexity that is compatible with other values of the "de Bruijn loss factor" [15] found in the literature. This increment occurs because the initial scripts are based on sophisticated tactics providing automation and code factorisation, whereas the final scripts are based on primitive tactics.

Our next objective is to improve the conversion procedure so that the increment factor is reduced as much as possible. In order to achieve this goal, the current output of our procedure needs further processing (before stage 5 of Definition 10) aimed at replacing groups of primitive tactics with advanced tactics having the same semantics. This means that advanced tactics with a formal semantics must be provided. We are also interested in limiting the use of declarative tactics in Definition 9 favouring their procedural counterparts.

Notice that the measure of complexity we used for the scripts is the number of tactics they contain. This measure does not take into account the complexity of the CIC terms appearing in the scripts as tactical arguments. We can estimate such complexity by counting the number of nodes occurring in the tree representation of these terms. So we plan to improve the accuracy of our tests by including this measure as well.

References

1. Coq development team: The Coq Proof Assistant Reference Manual Version 8.1. INRIA, Orsay (Feb 2007)
2. Asperti, A., Sacerdoti Coen, C., Tassi, E., Zacchiroli, S.: User Interaction with the Matita Proof Assistant. Journal of Automated Reasoning, Special Issue on User Interfaces for Theorem Proving. To appear (2006)
3. Coq development team: The Coq Proof Assistant Reference Manual Version 7.3.1. INRIA, Orsay (Oct 2002)
4. Guidi, F.: Lambda-Types on the Lambda-Calculus with Abbreviations. Submitted to ACM TOCL (Nov 2006) http://arxiv.org/cs.LO/0611040.
5. Barthe, G., Pons, O.: Type Isomorphisms and Proof Reuse in Dependent Type Theory. In Honsell, F., Miculan, M., eds.: 4th International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2001). Volume 2030 of LNCS., Springer (2001) 57–71
6. Sacerdoti Coen, C.: From Proof-Assistants to Distributed Libraries of Mathematics: Tips and Pitfalls. In: Mathematical Knowledge Management 2003. Volume 2594 of LNCS., Springer (2003) 30–44
7. Coquand, T., Paulin-Mohring, C.: Inductively defined types. In Martin-Löf, P., Mints, G., eds.: Proceedings of the International Conference on Computer Logic (Colog'88). Volume 417 of LNCS., Springer (1990)
8. Kamareddine, F., Laan, T., Nederpelt, R.: A Modern Perspective on Type Theory From its Origins Until Today. Volume 29 of Applied Logic Series. Kluwer Academic Publishers, Norwell (May 2004)
9. Sacerdoti Coen, C.: Declarative Representation of Proof Terms. Submitted to: Programming Languages for Mechanised Mathematics Workshop (PLMMS07) (2007)
10. Sacerdoti Coen, C.: Explanation in Natural Language of $\bar{\lambda}\mu\tilde{\mu}$-terms. In: 4th International Conference on Mathematical Knowledge Management (MKM2005). Volume 3863 of LNAI., Springer (2006) 234–249
11. Coscoy, Y.: A Natural Language Explanation for Formal Proofs. In Retoré, C., ed.: Int. Conf. on Logical Aspects of Computational Linguistics (LACL). Volume 1328 of LNAI., Springer (Sept 1996) 149–167
12. Asperti, A., Padovani, L., Sacerdoti Coen, C., Guidi, F., Schena, I.: Mathematical Knowledge Management in HELM. Annals of Mathematics and Artificial Intelligence **38**(1) (May 2003) 27–46
13. Maietti, M., Sambin, G.: Towards a minimalist foundation for constructive mathematics. In Crosilla, L., Schuster, P., eds.: From Sets and Types to Topology and Analysis: Practicable Foundations for Constructive Mathematics. Oxford University Press, Oxford (2005) Forthcoming.
14. Guidi, F.: λ-Types on the λ-Calculus with Abbreviations. Formal development with the proof assistant COQ (Jan 2006) http://www.cs.unibo.it/~fguidi/download/LAMBDA-TYPES.tgz.
15. de Bruijn, N.: A survey of the project Automath. In: Selected Papers on Automath. North-Holland, Amsterdam (1994) 141–161
16. van Benthem Jutting, L.: Checking Landau's Grundlagen in the Automath System. In: Selected Papers on Automath. North-Holland, Amsterdam (1994) 299–301,701–720,721–732,763–799,805–808
17. Wiedijk, F.: The De Bruijn Factor. Typescript note (2000) http://citeseer.ist.psu.edu/wiedijk00de.html.

$\lambda(y : nat).(nat_ind\ (\lambda(n : nat).(\forall(x : nat).((lt\ x\ n) \to (le\ x\ (pred\ n))))))\ (\lambda(x : nat).(\lambda(H : lt\ x\ O)).$let $H0 \equiv$ (match **H** in **le** return $(\lambda(n : nat).(\lambda(\_ : (le\ ?\ n)).((eq\ nat\ n\ O) \to (le\ x\ O))))$ with $[le_n \Rightarrow (\lambda(H0 : (eq\ nat\ (S\ x)\ O)).$let $H1 \equiv (eq_ind\ nat\ (S\ x)\ (\lambda(e : nat).($match e in nat return $(\lambda(\_ : nat).Prop)$ with $[O \Rightarrow False \mid (S\ \_) \Rightarrow True]))\ I\ O\ H0)$ in $(False\_ind\ (le\ x\ O)\ H1)) \mid (le\_S\ m\ H0) \Rightarrow (\lambda(H1 : (eq\ nat\ (S\ m)\ O)).((let$ **H2** $\equiv (eq_ind\ nat\ (S\ m)\ (\lambda(e : nat).($match e in nat return $(\lambda(\_ : nat).Prop)$ with $[O \Rightarrow False \mid (S\ \_) \Rightarrow True]))\ I\ O\ H1)$ in (**False_ind** $((le\ (S\ x)\ m) \to (le\ x\ O))$ **H2**)) **H0**)]) in $(H0\ (refl\_equal\ nat\ O)))\ (\lambda(n : nat).(\lambda(\_ : ((\forall(x : nat).((lt\ x\ n) \to (le\ x\ (pred\ n))))))).(\lambda(x : nat).(\lambda(H0 : (lt\ x\ (S\ n))).(le\_S\_n\ x\ n\ H0)))))\ y)$

Fig. 3. The initial proof term

$\lambda(y : nat).(nat_ind\ (\lambda(n : nat).(\forall(x : nat).((lt\ x\ n) \to (le\ x\ (pred\ n))))))\ (\lambda(x : nat).(\lambda(H : (lt\ x\ O)).$let $H0 \equiv$ (**le_ind** $(S\ x)\ (\lambda(n : nat).((eq\ nat\ n\ O) \to (le\ x\ O)))\ (\lambda(H0 : (eq\ nat\ (S\ x)\ O)).$let $H1 \equiv (eq_ind\ nat\ (Sx)\ (\lambda(e : nat).$match e return $(\lambda\_ : nat.Prop)$ with $[O \Rightarrow False \mid (S(\_ : nat)) \Rightarrow True])\ I\ O\ H0)$ in $(False_ind\ (le\ x\ O)\ H1))\ (\lambda(m : nat).(\lambda(H0 : (le\ (S\ x)\ m)).(\lambda(_ : ((eq\ nat\ m\ O) \to (le\ x\ O))).(\lambda(H1 : (eq\ nat\ (S\ m)\ O)).$let **H2** $\equiv (eq_ind\ nat\ (Sm)\ (\lambda e : nat.$match e return $(\lambda\_ : nat.Prop)$ with $[O \Rightarrow False \mid (S(\_ : nat)) \Rightarrow True])\ I\ O\ H1)$ in let **LOCAL** $\equiv$ (**False_ind** $((le\ (S\ x)\ m) \to (le\ x\ O))$ **H2**) in (**LOCAL H0**))))) O **H**) in $(H0\ (refl\_equal\ nat\ O))))\ (\lambda(n : nat).(\lambda(\_ : (\forall(x : nat).((lt\ x\ n) \to (le\ x\ (pred\ n))))))).(\lambda(x : nat).(\lambda(H0 : (lt\ x\ (S\ n))).(le\_S\_n\ x\ n\ H0)))))\ y)$

Fig. 4. The intermediate optimised proof term

```
theorem le_x_pred_y:
 (\forall y:nat.\forall x:nat.x<y -> x<=pred y).
intros 1 (y);
elim y using nat_ind in |- ((? -> ? ? % -> ? ? (? %))) names 0; [
intros 2 (x H); change in |- (%) with (x<=0);
cut (0=0 -> x<=0) as H0; [ id | change in H:(%) with (S x<=0);
elim H using le_ind in |- ((? ? % ? -> ?)) names 0; [
intros 1 (H0); cut match 0 in nat return \lambda _:nat.Prop with
 [0 -> False|S (_:nat) -> True] as H1; [ id |
rewrite < H0 in |- (%); change in |- (%) with True; apply I];
change in H1:(%) with False;
elim H1 using False_ind in |- (?) names 0 | intros 4 (m H0 _ H1);
cut match 0 in nat return \lambda _:nat.Prop with
 [0 -> False|S (_:nat) -> True] as H2; [ id |
rewrite < H1 in |- (%); change in |- (%) with True; apply I];
cut (S x<=m -> x<=0) as LOCAL; [ id | change in H2:(%) with False;
elim H2 using False_ind in |- (?) names 0];
apply LOCAL; apply H0]]; apply H0; apply refl_equal |
intros 4 (n _ x H0); change in |- (%) with (x<=n);
apply le_S_n; change in |- (%) with (x<S n); apply H0];
qed.
```

Fig. 5. The final proof script