Electronic Notes in Theoretical Computer Science 59 No. 4 (2001)
URL: `http://www.elsevier.nl/locate/entcs/volume59.html` 21 pages

Certifying Term Rewriting Proofs in ELAN

Quang-Huy Nguyen [1]

LORIA & INRIA
BP 101, 54602 Villers-lès-Nancy Cedex, France

Abstract

Term rewriting has been shown to be a good environment for both programming and proving. For analysing and debugging rule-based programs, we propose in this work a formalism based on the rewriting calculus with explicit substitutions ($\rho\sigma$-calculus). This formalism also allows us to build the proof terms of rewriting derivations. Therefore, term rewriting proofs can be exported to other systems by translating them into the corresponding syntaxes. That is, using a proof checker, one can certify these proofs and vice versa, this method allows us to get term rewriting in proof assistants using an external system. Our method not only works with syntactic rewriting but also with rewriting modulo a set of axioms (*e.g.* associativity-commutativity).

1 Introduction

In a proof assistant, formal proofs are composed of deduction steps performed by the user with the possible help of tactics. These steps should be memorised if one wants to check the proofs and to communicate them with other systems [24,14]. In some proof assistants (*e.g.* ALF, Coq, LEGO), the proof terms of all deduction steps are explicitly stored. In other systems, proof term memorising was not given the first priority but some attempts have been done to get this feature, for example, in Isabelle [6] or in HOL [35].

Term rewriting has been shown to be fundamental for both programming and proving. In proof assistants, term rewriting is very useful since it eases equational reasoning and considerably simplify proofs by abstracting the computational arguments [16]. However, in a system like Coq [20], term rewriting is painful due to the size of its proof term. In order to justify a rewrite step, the context and the used substitution need to be kept in the proof term. This fact poses a serious space problem for Coq kernel in proof checking. Besides,

[1] Email: `Quang-Huy.Nguyen@loria.fr`

term rewriting in Coq still requires user interactions and its efficiency is also limited since Coq works in interpreted mode.

At this stage, one might exploit the performance of the rule-based programming environments to help proof assistants in dealing with term rewriting. However, most of these environments consider term rewriting as computation but not as deduction. That is, no proof term is generated during the execution of a ruled-based program. In fact, some systems have considered tabling rewriting but only for improving efficiency or termination behaviour [33] [23], and the stored information cannot be seen as proof terms of their computations.

Hence, proof assistants must, in this situation, trust programming environments and consider the computations they perform as axioms in their proofs. This approach is not completely reliable since these computations are not checked. In other words, when correctness is crucial, rule-based programs should return the proof terms of their computations.

A rule-based program takes a term as its input and returns zero, one, or several terms as its output. When considering equational interpretation, the input and the output are equal in the context of the considered program. The derivation between them can be seen as a proof of this equality. On the other hand, it is well-known that using a relevant strategy, efficiency and some behaviors (*e.g.* termination) of rule-based programs can be significantly improved. Recently, some systems (*e.g.* ELAN [28], STRATEGO [25]) allow the user to guide the applications of rewrite rules instead of using a strategy by default like leftmost-innermost. This new feature gives rise to a demand to memorise user-guided computations, for example by tracing them. From a logical point of view, when rewriting is equationally interpreted, this trace can be seen as a proof of equality between the input and the output. From the programming aspect, the trace is very helpful in analysing and in debugging programs. And last but not least, this trace gives a means to export the computations (or proofs) to other systems.

The information kept in the trace depend on how this trace is used later. For example, in order to *replay* a derivation in syntactic rewriting, the trace of a rewrite step may simply contain the position of redex and the applied rule as it is used in [2] since pattern matching is deterministic and not costly. But when considering rewriting modulo a set of axioms like associativity and commutativity (AC rewriting), pattern matching becomes much more difficult (non-deterministic and NP-complete [5]) and hence, the trace of a rewrite step should contain more information such as the used substitution. Moreover, to improve the efficiency of pattern matching modulo AC, most of systems (*e.g.* ELAN, MAUDE) put a term in its *AC canonical form* before reducing it. The canonical form is built based on a total ordering on ground terms. In other words, the position of a redex depends on this ordering and rewriting context might be a better choice for memorising this redex. At this stage, a unified representation of trace (or proof term for term rewriting) is desirable since this allows to analyse the program execution and facilitates the syntactic manipulations (*e.g.* compacting or translating) on this trace.

The *rewriting calculus* [10,9] or ρCal is a simple calculus whose first class objects are the different ingredients of term rewriting *i.e.* terms, rules, rule applications, strategies and sets of results. Therefore, ρCal can capture both non-determinism where rewriting returns a set of results and failure where this set is empty. In ρCal, the arrow rewrite symbol ($\rightarrow$) is considered as the abstractor, while the left hand sides are seen as binding patterns. This generalises the abstraction mechanism of λ-calculus where the binding pattern is simply a variable and only trivial pattern matching is required. In other words, ρCal integrates term rewriting and λ-calculus in a unified framework where both of them can be naturally expressed. As a further step, ρCal with explicit substitutions ($\rho\sigma$-calculus) was designed [9] in order to make the substitution process explicit in the calculus.

One of the main motivations of this work is to communicate the proof term of rewriting derivation that is in first order syntax with proof assistants that are mostly based on higher order logic. Therefore, ρCal and eventually $\rho\sigma$-calculus, since we need to explicitly manipulate substitutions, is a relevant framework to represent proof terms.

In this paper we present a representation of the proof term of rewriting derivation based on $\rho\sigma$-calculus. We have generated the proof term of syntactic rewriting and AC rewriting in ELAN. The translation of this proof term into Coq-syntax is described and is proved sound. This translation has also been implemented in ELAN. As a result, we get the corresponding proof term for Coq proof assistant. This proof term is then formally checked by Coq to ensure correctness. This work allows one to certify term rewriting proofs in ELAN and hence, to get term rewriting in Coq using ELAN.

The notations used in this paper are described in section 2. Section 3 reviews the syntax and the operational semantics of $\rho\sigma$-calculus. In section 4, we describe the proof term for term rewriting in this syntax. The proof term syntax of equality in Coq is presented in section 5. Section 6 describes the translation between the two above syntaxes. We describe and discuss our current implementation in section 7 before giving the conclusion.

2 Preliminaries

We mostly use the notations introduced in [15] and [19].

A rewrite rule is written $l \rightarrow r$ where l is not a variable and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. We call l and r respectively the *left hand side* (LHS) and the *right hand side* (RHS) of rule. A set of rewrite rules $\mathcal{R}$ is called a *term rewriting system* (TRS). Let $\mathcal{R}$ be a TRS. The term t rewrites to the term s in one *rewrite step* if there exist some rule $l \rightarrow r$ in $\mathcal{R}$, position p in t and substitution σ such that: $t|_p = l\sigma$ and $s = t[r\sigma]_p$. If p is not empty, then this step is called *non-root*. We denote a rewrite step in $\mathcal{R}$ by $t \rightarrow_{\mathcal{R}} s$ and the reflexive-transitive closure of the binary relation $\rightarrow_{\mathcal{R}}$ by $\rightarrow^{*}_{\mathcal{R}}$. This relation is also called *syntactic rewriting.* The subterm $t|_p = l\sigma$ is called a *redex* in t since it is an instance of a LHS of $\mathcal{R}$. If we replace this redex by a "hole", then we get the *rewriting context* of the rewrite step at position p. A term is said to be in *normal form* (NF) *w.r.t.* $\mathcal{R}$ if it contains no redex. A *(rewriting) derivation* in $\mathcal{R}$ is any (finite or infinite) sequence of rewrite steps.

Let $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ be two rewrite rules with distinct variables. If p is the position of a non-variable subterm of l_2, σ is a most general unifier of l_1 and $l_2|_p$, then the equation $r_2\sigma = l_2\sigma[r_1\sigma]_p$ is a *critical pair* formed from those rules.

A *conditional rewrite rule* is written $l \rightarrow r$ **if** c where c is called the *condition.* There are several methods to define semantics of conditional rewriting. In this work, a (conditional) rewrite step is performed if and only if $c\sigma$ (σ is the used substitution) can be evaluated to $True$ by the same rewrite system and by the reflexivity of equality.

We denote the equality modulo an equational theory T by $=_T$: $t =_T s$ if and only if $T \models t = s$. Associativity and commutativity (AC) is one of the most useful equational theories. By replacing syntactic equality by equality modulo AC in the definition of rewrite steps we get AC rewriting relation. By $l \ll^{?}_{T} t$ we denote the matching problem between term t and pattern l in equational theory T. A solution of this problem is the substitution σ such that $l\sigma =_T t$. The set of such solutions is written $Sol(l \ll^{?}_{T} t)$.

When working with Coq-syntax, the bold Sans serif font (*e.g.* **Lemma**) is used for Coq keywords, while the bold font (*e.g.* A) is used for Coq identifiers. The notation t : A means that the type of t is A. The arrow symbol ($\rightarrow$) in a type expression denotes a functional type and is right associative. The application of term f on term t is denoted by $(f\ t)$. Term abstractor (λ) is written [], *e.g.* $[x, y : \mathsf{A}; z : \mathsf{B}]x$ denotes $\lambda xy : \mathsf{A}\lambda z : \mathsf{B}.x$. Type abstractor ($\Pi$) is written (), *e.g.* $(x, y : \mathsf{A}; z : \mathsf{B})x$ denotes $\Pi xy : \mathsf{A}\Pi z : \mathsf{B}.x$.

3 $\rho\sigma$-calculus

We only introduce here the notions and notations which are useful in this work. For a general presentation of explicit substitution calculi, the reader is referred for example, to [1,13].

Syntax For all $x \in \mathcal{V}$ and $f \in \mathcal{F}$:

$$\begin{array}{rrcl} \text{terms} & t & ::= & x \mid f(t,...,t) \mid \{t,...,t\} \mid [t](t) \mid t \rightarrow t \mid t\langle s\rangle \\ \text{substitutions} & s & ::= & \mathbb{ID} \mid \uparrow \mid \Uparrow \mid t.s \mid s \circ s \end{array}$$

In the term syntax, $t \rightarrow u$ denotes a rewrite rule (*a.k.a.*ρ-abstraction), $[t](u)$ represents the application of t on u, the application of substitution s on t is denoted by binary operator $t\langle s\rangle$. The substitution syntax is composed of the identity substitution ($\mathbb{ID}$), the *shift* ($\uparrow$), the composition operator ($s \circ v$), the substitution concatenator ($s.v$) and the *lift* ($\Uparrow(s)$).

Operational semantics Figure 1 describes the evaluation rules of the $\rho\sigma$-calculus. The rule **F**ire describes the application of a rewrite rule $l \to r$ at root position of term t. This rewrite step replaces t by a set of instantiated RHSs $\{r\langle\sigma\rangle\}$ where σ is a solution of the pattern matching problem between t and l. The rule **C**$ongruence$ allows one to apply a rule on a non-root position. The five rules **D**$istrib$, **B**$atch$, **S**$witch$, **O**$pOnSet$ and **Fl**at are added to manipulate the sets of results.

(**F**)ire	$[l \to r](t)$	$\mapstochar\twoheadrightarrow$	$\{r\langle\sigma\rangle\}$ where $\sigma \in Sol(l \ll^?_\emptyset t)$
(**C**)$ongruence$	$[f(s_1, ..., s_n)](f(t_1, ..., t_n))$	$\mapstochar\twoheadrightarrow$	$\{f([s_1](t_1), ..., [s_n](t_n))\}$
(**C**)$ong\_$(**f**)ail	$[f(s_1, ..., s_n)](g(t_1, ..., t_m))$	$\mapstochar\twoheadrightarrow$	$\emptyset$
(**D**)$istrib$	$[\{s_1, ..., s_n\}](t)$	$\mapstochar\twoheadrightarrow$	$\{[s_1](t), ..., [s_n](t)\}$
(**B**)$atch$	$[s](\{t_1, ..., t_n\})$	$\mapstochar\twoheadrightarrow$	$\{[s](t_1), ..., [s](t_n)\}$
(**S**)$witch$	$s \to \{t_1, ..., t_n\}$	$\mapstochar\twoheadrightarrow$	$\{s \to t_1, \ldots, s \to t_n\}$
(**O**)$pOnSet$	$f(s_1, ..., \{t_1, ..., t_m\}, ..., s_n\}$	$\mapstochar\twoheadrightarrow$	$\{f(s_1, ..., t_1, ..., s_n), ...,$ $f(s_1, ..., t_m, ..., s_n)\}$
(**Fl**)at	$\{s_1, ..., \{t_1, ..., t_m\}, ..., s_n\}$	$\mapstochar\twoheadrightarrow$	$\{s_1, ..., t_1, ..., t_m, ..., s_n\}$

Fig. 1. Evaluation rules of $\rho\sigma$-calculus

Since the substitution process is explicit in this calculus, a set of evaluation rules for $t\langle s\rangle$ is also presented in [9]. We do not describe these rules here due to space reason but using them, one can reduce $t\langle\sigma\rangle$ to $t\sigma$ for any first order term t and substitution σ, as in other explicit substitution calculi. In the sequel, $\mapstochar\twoheadrightarrow_\sigma$ denotes the reduction relation by these rules while $\mapstochar\twoheadrightarrow_\rho$ stands for the reduction relation by the rules in figure 1. The combination of these two relations is written $\mapstochar\twoheadrightarrow_{\rho\sigma}$.

4 Proof Term for Term Rewriting in $\rho\sigma$-syntax

Definition 4.1 (Proof term) Let $\mathcal{R}$ be a TRS and t, s be two terms. The $\rho\sigma$-term π is called a proof term of the derivation $t \to^*_{\mathcal{R}} s$ if $[\pi](t) \mapstochar\twoheadrightarrow^*_{\rho\sigma} \{s\}$.

The proof term of the identity derivation $t \to t$ is given by the $\rho\sigma$-term $\mathbf{id} \equiv [x \to x]$ since $[\mathbf{id}](t) \equiv [x \to x](t) \mapstochar\twoheadrightarrow_{\mathbf{F}} \{t\}$ for every term t.

Lemma 4.2 (Rewrite step at root position) *If $l \to r$ is the applied rule and σ is the used substitution, then $l\langle\sigma\rangle \to r\langle\sigma\rangle$ is a proof term of the one-step derivation $l\sigma \to r\sigma$.*

Proof. By the following derivation:

$$[l\langle\sigma\rangle \to r\langle\sigma\rangle](l\sigma) \mapsto\!\!\!\twoheadrightarrow_{\mathbf{F}} \{r\langle\sigma\rangle\langle\mathbb{ID}\rangle\} \mapsto\!\!\!\twoheadrightarrow_{\mathbf{I}} \{r\langle\sigma\rangle\} \mapsto\!\!\!\twoheadrightarrow^*_{\sigma} \{r\sigma\}$$

□

Lemma 4.3 (Conditional rewrite step at root position) *Let $l \to r$* **if** *c be the applied rule and σ be the used substitution. If π_c is a proof term of the derivation $c\sigma \to^*_{\mathcal{R}} True$, then $l\langle\sigma\rangle \to [True \to r\langle\sigma\rangle]([\pi_c](c\langle\sigma\rangle))$ is a proof term of the one-conditional-step derivation $l\sigma \to r\sigma$.*

Proof. By the following derivation:

$$\begin{array}{llll}
 & [l\langle\sigma\rangle \to [True \to r\langle\sigma\rangle]([\pi_c](c\langle\sigma\rangle))](l\sigma) & \mapsto\!\!\!\twoheadrightarrow_{\mathbf{F}} & [True \to r\langle\sigma\rangle]([\pi_c](c\langle\sigma\rangle)) \\
\mapsto\!\!\!\twoheadrightarrow^*_{\sigma} & [True \to r\langle\sigma\rangle]([\pi_c](c\sigma)) & \mapsto\!\!\!\twoheadrightarrow^*_{\rho\sigma} & [True \to r\langle\sigma\rangle](\{True\}) \\
\mapsto\!\!\!\twoheadrightarrow_{\mathbf{B}} & \{[True \to r\langle\sigma\rangle](True)\} & \mapsto\!\!\!\twoheadrightarrow_{\mathbf{F}} & \{\{r\langle\sigma\rangle\}\} \\
\mapsto\!\!\!\twoheadrightarrow_{\mathbf{Fl}} & \{r\langle\sigma\rangle\} & \mapsto\!\!\!\twoheadrightarrow^*_{\sigma} & \{r\sigma\}
\end{array}$$

□

Lemma 4.4 (Non-root rewrite step) *If π_i is a proof term of the rewrite step $t_i \to s_i$, then $f(\mathbf{id}, ..., \pi_i, ..., \mathbf{id})$ is a proof term of the one-step derivation $f(t_1, ..., t_i, ..., t_n) \to f(t_1, ..., s_i, ..., t_n)$*

Proof. By the following derivation:

$$\begin{array}{llll}
 & [f(\mathbf{id}, ..., \pi_i, ..., \mathbf{id})](f(t_1, ..., t_i, ..., t_n)) & & \\
\mapsto\!\!\!\twoheadrightarrow_{\mathbf{C}} & \{f([\mathbf{id}](t_1), ..., [\pi_i](t_i), ..., [\mathbf{id}](t_n)\} & \mapsto\!\!\!\twoheadrightarrow^*_{\rho\sigma} & \{f(t_1, ..., \{s_i\}, ..., t_n)\} \\
\mapsto\!\!\!\twoheadrightarrow_{\mathbf{O}} & \{\{f(t_1, ..., s_i, ..., t_n)\}\} & \mapsto\!\!\!\twoheadrightarrow_{\mathbf{Fl}} & \{f(t_1, ..., s_i, ..., t_n)\}
\end{array}$$

□

Lemma 4.5 (Rewriting derivation) *If $\pi_1, ..., \pi_n$ are respectively a proof term of the derivations $t_1 \to t_2$,...,$t_n \to t_{n+1}$, then $x \to [\pi_n](\ldots[\pi_1](x)\ldots)$ is a proof term of the n-steps derivation $t_1 \to \ldots \to t_{n+1}$*

Proof. By the following derivation:

$$\begin{array}{llll} & [x \to [\pi_n](\ldots[\pi_1](x)\ldots)](t_1) & \mapsto\!\!\!\twoheadrightarrow_{\mathbf{F}} & \{[\pi_n](\ldots[\pi_1](t_1)\ldots)\} \\ \mapsto\!\!\!\twoheadrightarrow^*_{\rho\sigma} & \{[\pi_n](\ldots[\pi_2](\{t_2\})\ldots)\} & \mapsto\!\!\!\twoheadrightarrow_{\mathbf{B}} & \{[\pi_n](\ldots\{[\pi_2](t_2)\}\ldots)\} \\ \mapsto\!\!\!\twoheadrightarrow_{\rho\sigma} & \ldots & \mapsto\!\!\!\twoheadrightarrow_{\rho\sigma} & \{[\pi_n](t_n)\} \\ \mapsto\!\!\!\twoheadrightarrow^*_{\rho\sigma} & \{\{t_{n+1}\}\} & \mapsto\!\!\!\twoheadrightarrow_{\mathbf{Fl}} & \{t_{n+1}\} \end{array}$$

□

For short, in the sequel, we denote $x \to [\pi_n](\ldots[\pi_1](x)\ldots)$ by $\pi_1;\ldots;\pi_n$.

Example 4.6 Consider the TRS

$$\mathcal{R} = \left\{ f(x) \to x \right.$$

The derivation $g(f(a), f(b)) \to^*_{\mathcal{R}} g(a, b)$ has a proof term given by the $\rho\sigma-$term:

$$\pi = g(f(x)\langle x \mapsto a\rangle \to x\langle x \mapsto a\rangle, \mathbf{id}); g(\mathbf{id}, f(x)\langle x \mapsto b\rangle \to x\langle x \mapsto b\rangle)$$

Proof term compacting Proof terms need to be as concise as possible so that they can efficiently be stored, communicated and checked by proof assistants. The rules in figure 2 allow to reduce the size of proof terms by combining proof terms in different branches of a function symbol and by eliminating the proof term of identity. This rewrite system is terminating since the size of a proof term strictly decreases after the application of each rule on it, and confluent since the only critical pair formed by *Id_elim_l* and *Id_elim_r* is trivial.

$$\begin{array}{llll} \mathit{Disjoint} & f(t_1, ..., t_n); f(s_1, ..., s_n) & \mapsto\!\!\!\twoheadrightarrow & f(t_1; s_1, ..., t_n; s_n) \\ \mathit{Id\_elim\_l} & \mathbf{id}; t & \mapsto\!\!\!\twoheadrightarrow & t \\ \mathit{Id\_elim\_r} & t; \mathbf{id} & \mapsto\!\!\!\twoheadrightarrow & t \end{array}$$

Fig. 2. Reduction rules for proof terms in $\rho\sigma$-syntax

Example 4.7 The proof term π in example 4.6 can be reduced as follows:

$$\begin{array}{rl} \pi = & g(f(x)\langle x \mapsto a\rangle \to x\langle x \mapsto a\rangle, \mathbf{id}); g(\mathbf{id}, f(x)\langle x \mapsto b\rangle \to x\langle x \mapsto b\rangle) \\ \mapsto\!\!\!\twoheadrightarrow_{\mathit{Disjoint}} & g(f(x)\langle x \mapsto a\rangle \to x\langle x \mapsto a\rangle; \mathbf{id}\ ,\ \mathbf{id}; f(x)\langle x \mapsto b\rangle \to x\langle x \mapsto b\rangle) \\ \mapsto\!\!\!\twoheadrightarrow_{\mathit{Id\_elim\_r}} & g(f(x)\langle x \mapsto a\rangle \to x\langle x \mapsto a\rangle\ ,\ \mathbf{id}; f(x)\langle x \mapsto b\rangle \to x\langle x \mapsto b\rangle) \\ \mapsto\!\!\!\twoheadrightarrow_{\mathit{Id\_elim\_l}} & g(f(x)\langle x \mapsto a\rangle \to x\langle x \mapsto a\rangle\ ,\ f(x)\langle x \mapsto b\rangle \to x\langle x \mapsto b\rangle) = \pi_{\rho\sigma} \end{array}$$

Related work There are several works in the literature on representing proofs in rewriting logic. First of them was Meseguer's work on modelling concurrency by (conditional) rewriting logic [22] (see also [18]). Our formalism is more related to the representation proposed in Gadducci's work on *flat rewriting logic* [17] since we orient towards an implementation in sequential machines and hence, do not allow concurrent rewrite steps. Our choice of $\rho\sigma$-calculus is motivated by the need of a readable syntax and of a strong relationship with type theory [11]. Besides, conditional rewriting is smoothly incorporated in our formalism thanks to the evaluation mechanism of $\rho\sigma$-calculus.

5 Proof Term of Equality in Coq

Coq is a proof assistant based on the Calculus of Inductive Constructions (the Calculus of Constructions [12] with inductive data types [27]). Proofs in Coq are constructive and hence, assuming excluded middle is up to the user. By Curry-Howard isomorphism, logical propositions are interpreted as types. A proposition is provable if and only if it is inhabited by some terms built using Coq pre-defined *constants*. These terms are the proof terms of that proposition. Coq proof mode is *interactive*: the user builds proofs in a *backward* style using Coq tactics. At each step, the applied tactic generates a list of subgoals needed to be proved in order to conclude the current goal. The proof terms generated in deduction steps are stored and certified (*i.e.* type checked) by Coq kernel. This approach has some advantages: correctness is ensured by the reliability of a tiny kernel which has formally been proved in [3], a certified (functional) program can be extracted from the proof of its specification [26], etc. However, this mechanism requires to keep all informations concerning each deduction step in the proof term and sometimes, poses a serious problem of space.

5.1 Equality in Coq

Coq uses Leibniz equality. Let A be the carrier type and **Prop** be the type which represents logical propositions in Coq. Leibniz equality is an equivalence relation $=_{\mathsf{A}}$ on A such that:

$$\forall x, y : \mathsf{A}.x =_{\mathsf{A}} y \;\Leftrightarrow\; \forall \Phi : \mathsf{A} \to \textbf{Prop}.(\Phi\ x) \to (\Phi\ y)$$

Leibniz equality enjoys standard properties such as reflexivity, transitivity and substitutivity. For equality proofs using these properties, Coq provides corresponding constants allowing to build proof terms.

$$\frac{x:\mathsf{A}}{(\textsf{\textbf{refl\_equal}}\ \mathsf{A}\ x):\ x =_{\mathsf{A}} x}\ \text{Reflexivity}$$

$$\frac{x, y, z:\mathsf{A}\quad \pi_1 : x =_{\mathsf{A}} y\quad \pi_2 : y =_{\mathsf{A}} z}{(\textsf{\textbf{trans\_equal}}\ \mathsf{A}\ x\ y\ z\ \pi_1\ \pi_2):\ x =_{\mathsf{A}} z}\ \text{Transitivity}$$

$$\frac{\Phi:\mathsf{A} \to \textsf{\textbf{Prop}}\quad x, y:\mathsf{A}\quad \pi_1 : x =_{\mathsf{A}} y\quad \pi_2 : (\Phi\ x)}{(\textsf{\textbf{eq\_ind}}\ \mathsf{A}\ x\ \Phi\ \pi_2\ y\ \pi_1) : (\Phi\ y)}\ \text{Substitutivity}$$

Reflexivity: If t is a term of type A, then ($\textsf{\textbf{refl\_equal}}$ A t) is a proof of $t =_{\mathsf{A}} t$.

Transitivity: Let t, s and v be three terms of type A. If π_1 and π_2 are respectively a proof of $t =_{\mathsf{A}} s$ and $s =_{\mathsf{A}} v$, then ($\textsf{\textbf{trans\_equal}}$ A $t\ s\ v\ \pi_1\ \pi_2$) is a proof of $t =_{\mathsf{A}} v$.

Substitutivity: Let t, s be two terms of type A and $\Phi : \mathsf{A} \to \textsf{\textbf{Prop}}$ be a predicate. If π_1 and π_2 are respectively a proof of $t =_{\mathsf{A}} s$ and $(\Phi\ t)$, then ($\textsf{\textbf{eq\_ind}}$ A $t\ \Phi\ \pi_2\ s\ \pi_1$) is a proof of $(\Phi\ s)$.

This rule actually allows us to simplify a goal by term rewriting inside a context (Φ). The proof term of the rewrite step $(\Phi\ t) \to (\Phi\ s)$ contains rewriting context Φ, instantiated LHS t, instantiated RHS s and a proof of $t =_{\mathsf{A}} s$.

Notice that these constants correspond to the case where the type of A is $\textsf{\textbf{Set}}$. The other case (A : $\textsf{\textbf{Prop}}$) requires two other constants but their usage is similar. In the sequel, for short, if A is involved in the context, then we use $=$ to denote Leibniz equality on A instead of $=_{\mathsf{A}}$.

5.2 Rewrite rule

The equational axiom $l = r$ is specified in Coq by the following axiom:

$$\ell : (x_1, ..., x_n : \mathsf{A})\ l = r$$

where ℓ is the label of this axiom and $x_1, ..., x_n$ are the variables of l and r ($\{x_1, ..., x_n\} \equiv \mathcal{V}ar(l) \cup \mathcal{V}ar(r)$). If $t_1, ..., t_n$ are n terms of type A, then $(\ell\ t_1\ \ldots\ t_n)$ is of type (or a proof) of $l\{x_1 \mapsto t_1\}\ldots\{x_n \mapsto t_n\} = r\{x_1 \mapsto t_1\}\ldots\{x_n \mapsto t_n\}$.

5.3 Proof term factorisation

The contexts stored in the proof term of a rewriting derivation are usually redundant. Redundancy appear in particular when some rewrite steps are performed at the same non-root position or when some rewrite steps are performed at disjoint branches of a function symbol.

Non-root rewrite steps The context needs to be separated from the proof term of a rewrite step in order to avoid repeating identical context when performing several rewrite steps at the same position. To this end, for each rewrite step (or derivation) at position p of term t, the following lemma is added and proved:

$$\textbf{Lemma}\ \mathsf{ctx\_t} : (\mathsf{x}, \mathsf{y} : \mathsf{A})\ \mathsf{x} = \mathsf{y} \to \mathsf{t}[\mathsf{x}]_\mathsf{p} = \mathsf{t}[\mathsf{y}]_\mathsf{p}$$

The proof of this lemma consists simply in applying the tactic **Rewrite** $\mathsf{x} = \mathsf{y}$ on t. After being proved, this lemma can be used as a new constant for building Coq proof terms. Let s and v be two terms of type A. If π is a proof of $s = v$, then $(\mathsf{ctx\_t}\ s\ v\ \pi)$ is a proof of $t[s]_p = t[v]_p$.

Rewrite steps at disjoint branches The root function symbol is the common part of the contexts and needs to be factorised. To this end, for each arity value n, the following lemma is added and proved:

$$\begin{aligned}\textbf{Lemma}\ \mathsf{eq\_concat\_n} : {} & (\mathsf{x}_1, ..., \mathsf{x}_\mathsf{n} : \mathsf{A}; \Phi : (\overbrace{\mathsf{A} \to \ldots \to \mathsf{A}}^{n \text{ times}} \to \textbf{Prop})) \\ & (\Phi\ \mathsf{x}_1 \ldots \mathsf{x}_\mathsf{n}) \to (\mathsf{y}_1 : \mathsf{A})\mathsf{x}_1 = \mathsf{y}_1 \to \ldots \to (\mathsf{y}_\mathsf{n} : \mathsf{A})\mathsf{x}_\mathsf{n} = \mathsf{y}_\mathsf{n} \\ & \to (\Phi\ \mathsf{y}_1 \ldots \mathsf{y}_\mathsf{n})\end{aligned}$$

This lemma also avoids copying all other branches in the context of each rewrite step performed in a branch. The proof of this lemma consists of n applications of the **Rewrite** tactic: **Rewrite** $\mathsf{x}_1 = \mathsf{y}_1 \ldots$ **Rewrite** $\mathsf{x}_\mathsf{n} = \mathsf{y}_\mathsf{n}$. This lemma provides a new constant for building Coq proof terms. Let $t_1, ..., t_n, s_1, ..., s_n$ be $2n$ terms of type A and $\Phi : \overbrace{\mathsf{A} \to \ldots \to \mathsf{A}}^{n \text{ times}} \to \textbf{Prop}$ be a predicate. If $\pi_1, ..., \pi_n$ are respectively a proof of $t_1 = s_1, ..., t_n = s_n$ and π is a proof of $(\Phi\ t_1\ \ldots\ t_n)$, then $(\mathsf{eq\_concat\_n}\ t_1\ \ldots t_n\ \Phi\ \pi\ s_1\ \pi_1\ \ldots\ s_n\ \pi_n)$ is a proof of $(\Phi\ s_1\ \ldots\ s_n)$.

6 Proof Term Translation

For building proof terms in Coq-syntax, the proof term of identity (**id**) in $\rho\sigma$-syntax is replaced by the terms on which it applies.

An auxiliary syntax (Π-syntax) is first introduced as a bridge between $\rho\sigma$-syntax and Coq-syntax. Π-syntax makes the translation more open in the sense that one can parameterise it by proof term syntaxes of proof checkers. Two operators $\rho\sigma 2\Pi$ and $\Pi 2 Coq$ which respectively translate proof terms from $\rho\sigma$-syntax to Π-syntax and from Π-syntax to Coq-syntax are then presented. Finally, a soundness proof of the whole translation process is described.

6.1 Π*-syntax*

We define Π-syntax as a simple syntax which represents proof terms in a compact format and which is straightforward to be translated into the syntaxes of proof checkers. In order to stay independent of proof checkers, we only use in $\Pi-$syntax the basic properties of equality, namely reflexivity, transitivity and substitutivity.

$$\text{terms } \; t ::= s \Big| CTX \Big| concat\_f(t, ..., t) \Big| trans(t, t) \Big| refl(t) \Big| ctx(t, t)$$

where s is any $\rho\sigma$-term which represents the proof term of a rewrite step (see section 4); CTX is a fresh constant representing the hole in a context; $concat\_f$ inherits the arity from $f \in \mathcal{F}$ and combines the proof terms at disjoint branches of f; $trans$ concatenates the proof terms of two consecutive derivations; $refl$ represents a proof by reflexivity; ctx represents the proof term of a non-root rewrite step. The introduction of ctx aims to separate the context from the proof term of a rewrite step and hence, to reduce the size of proof terms in Coq-syntax as described in section 5.3.

Example 6.1 Consider the proof term $\pi_{\rho\sigma}$ in example 4.7. Its counterpart in $\Pi-$syntax is

$$\pi_{\Pi} = concat\_g(f(x)\langle x \mapsto a\rangle \to x\langle x \mapsto a\rangle \;,\; f(x)\langle x \mapsto b\rangle \to x\langle x \mapsto b\rangle)$$

But when considering the derivation $g(f(a), b) \to_{\mathcal{R}} g(a, b)$ we have the following proof term in $\rho\sigma$-syntax: $g(f(x)\langle x \mapsto a\rangle \to x\langle x \mapsto a\rangle \;,\; b)$ and its counterpart in Π-syntax:

$$ctx(g(CTX, b) \;,\; f(x)\langle x \mapsto a\rangle \to x\langle x \mapsto a\rangle)$$

6.2 $\rho\sigma 2\Pi$

Figure 3 describes the evaluation rules for $\rho\sigma 2\Pi$. The rule *Step* states that the proof term of a rewrite step at root position is not changed by $\rho 2\Pi$. The rule *Concat* concatenates the proof terms of n consecutive derivations. The rule *Refl* treats the case where no rewrite step is performed. This case corresponds to a proof by reflexivity. The rule *Disj*1 separates the context from the proof term of a derivation performed in *one branch* of a function symbol. The rule *Disj*2 deals with the case where rewrite steps are performed at disjoint branches of a function symbol.

Example 6.2 The translation of $\pi_{\rho\sigma}$ into π_{Π} by $\rho\sigma 2\Pi$ is described as follows:

$$\begin{aligned}
& \rho\sigma 2\Pi(g(f(x)\langle x \mapsto a\rangle \to x\langle x \mapsto a\rangle \;,\; f(x)\langle x \mapsto b\rangle \to x\langle x \mapsto b\rangle)) \\
\mapsto\!\!\!\twoheadrightarrow_{Disj2} \; & concat\_g(\rho\sigma 2\Pi(f(x)\langle x \mapsto a\rangle \to x\langle x \mapsto a\rangle) \;, \\
& \rho\sigma 2\Pi(f(x)\langle x \mapsto b\rangle \to x\langle x \mapsto b\rangle)) \\
\mapsto\!\!\!\twoheadrightarrow_{Step} \; & concat\_g(f(x)\langle x \mapsto a\rangle \to x\langle x \mapsto a\rangle \;,\; \rho\sigma 2\Pi(f(x)\langle x \mapsto b\rangle \to x\langle x \mapsto b\rangle)) \\
\mapsto\!\!\!\twoheadrightarrow_{Step} \; & concat\_g(f(x)\langle x \mapsto a\rangle \to x\langle x \mapsto a\rangle \;,\; f(x)\langle x \mapsto b\rangle \to x\langle x \mapsto b\rangle) = \pi_{\Pi}
\end{aligned}$$

$$\begin{array}{lll}
\mathit{Step} & \rho\sigma 2\Pi(t \to s) & \mapsto\!\!\!\!\twoheadrightarrow t \to s \\
\mathit{Concat} & \rho\sigma 2\Pi((t_1;\ldots;t_n)) & \mapsto\!\!\!\!\twoheadrightarrow \mathit{trans}(\rho\sigma 2\Pi(t_1), \rho\sigma 2\Pi(t_2;\ldots;t_n)) \\
\mathit{Refl} & \rho\sigma 2\Pi(t) & \mapsto\!\!\!\!\twoheadrightarrow \mathit{refl}(t) \;\text{ if } t \in \mathcal{T}(\mathcal{F},\mathcal{V}) \\
\mathit{Disj}1 & \rho\sigma 2\Pi(f(t_1,...,t_n)) & \mapsto\!\!\!\!\twoheadrightarrow \mathit{ctx}(f(t_1,...,t_i[CTX]_p,...,t_n), \rho\sigma 2\Pi(t_i|_p)) \\
 & & \qquad \text{if } \exists!\ i \text{ such that } t_i \text{ contains a rewrite step at position } p \\
\mathit{Disj}2 & \rho\sigma 2\Pi(f(t_1,...,t_n)) & \mapsto\!\!\!\!\twoheadrightarrow \mathit{concat\_f}(\rho\sigma 2\Pi(t_1),...,\rho\sigma 2\Pi(t_n)) \\
 & & \qquad \text{if } \exists\ i,j (i \neq j) \text{ such that } t_i, t_j \text{ contain a rewrite step}
\end{array}$$

Fig. 3. Evaluation rules for $\rho\sigma 2\Pi$

6.3 $\Pi 2\mathit{Coq}$

Two operators In and Out which return the input (the term to be reduced) and the output (the result) of a rewriting derivation are first described by the rules in figure 4.

$$\begin{array}{rlrl}
In(t \to s) & \mapsto\!\!\!\!\twoheadrightarrow t & In(\mathit{trans}(\pi_1,\pi_2)) & \mapsto\!\!\!\!\twoheadrightarrow In(\pi_1) \\
In(\mathit{refl}(t)) & \mapsto\!\!\!\!\twoheadrightarrow t & In(\mathit{ctx}(t[CTX]_p,\pi)) & \mapsto\!\!\!\!\twoheadrightarrow t[In(\pi)]_p \\
In(\mathit{concat\_f}(\pi_1,...,\pi_n)) & \mapsto\!\!\!\!\twoheadrightarrow f(In(\pi_1),...,In(\pi_n)) & & \\
 & & & \\
Out(t \to s) & \mapsto\!\!\!\!\twoheadrightarrow s & Out(\mathit{trans}(\pi_1,\pi_2)) & \mapsto\!\!\!\!\twoheadrightarrow Out(\pi_2) \\
Out(\mathit{refl}(t)) & \mapsto\!\!\!\!\twoheadrightarrow t & Out(\mathit{ctx}(t[CTX]_p,\pi)) & \mapsto\!\!\!\!\twoheadrightarrow t[Out(\pi)]_p \\
Out(\mathit{concat\_f}(\pi_1,...,\pi_n)) & \mapsto\!\!\!\!\twoheadrightarrow f(Out(\pi_1),...,Out(\pi_n)) & &
\end{array}$$

Fig. 4. Evaluation rules for In and Out

The evaluation rules for $\Pi 2\mathit{Coq}$ are then presented for all possible output of $\rho\sigma 2\Pi$ (see figure 3).

rewrite steps at root position The proof term in Coq-syntax is given by instantiating the corresponding axiom with the used substitution:

$$\Pi 2\mathit{Coq}(l\langle\sigma\rangle \to r\langle\sigma\rangle) \mapsto\!\!\!\!\twoheadrightarrow (\ell\ \sigma_{x_1}\ \ldots\ \sigma_{x_n})$$

where $\{x_1,...,x_n\} \equiv \mathcal{V}ar(l) \cup \mathcal{V}ar(r)$ and ℓ is the label of the axiom $l = r$ in

Coq.

trans The proof term in Coq-syntax is built by concatenating proof terms of two consecutive derivations using **trans_equal**:

$$\Pi2Coq(trans(\pi_1,\pi_2)) \mapsto\!\!\!\twoheadrightarrow (\textbf{trans\_equal}\ \mathsf{A}\ In(\pi_1)\ Out(\pi_1)\ Out(\pi_2)\ \Pi2Coq(\pi_1)$$
$$(\textbf{trans\_equal}\ \mathsf{A}\ Out(\pi_1)\ In(\pi_2)\ Out(\pi_2)\ \mathsf{lm\_eq}\ \Pi2Coq(\pi_2)))$$

where lm_eq is the following lemma: **Lemma** lm_eq : $Out(\pi_1) = In(\pi_2)$. This lemma states that the output of the first derivation is equal to the input of the second derivation *modulo the working equational theory* T. This statement is obvious in case of syntactic rewriting ($T \equiv \emptyset$) where no lemma is generated. However, in case of rewriting modulo a set of axioms, it is less obvious since the output of the first derivation can be syntactically transformed before being reduced in the second derivation. For instance, in AC rewriting, a term is usually put in AC-canonical form before any further reduction is applied on it. In this case, lm_eq needs to be added and proved since Coq has not reasoning modulo AC in its kernel.

refl The proof term in Coq-syntax is built using **refl_equal** where A is the carrier type:

$$\Pi2Coq(refl(t)) \mapsto\!\!\!\twoheadrightarrow (\textbf{refl\_equal}\ \mathsf{A}\ t)$$

ctx The proof term in Coq-syntax is built using lemma ctx_t:

$$\Pi2Coq(ctx(t[CTX]_p,\pi)) \mapsto\!\!\!\twoheadrightarrow (\mathsf{ctx\_t}\ In(\pi)\ Out(\pi)\ \Pi2Coq(\pi))$$

concat_f This case corresponds to the derivations in different branches of function symbol f whose arity is n. The proof term in Coq-syntax is built using lemma eq_concat_n where A is the carrier type:

$$\Pi2Coq(concat\_f(\pi_1,...,\pi_n)) \mapsto\!\!\!\twoheadrightarrow (\mathsf{eq\_concat\_n}\ \ In(\pi_1)\ \ \ldots\ \ In(\pi_n)$$
$$[x_1,...,x_n : \mathsf{A}]f(In(\pi_1),...,In(\pi_n)) = f(x_1,...,x_n)$$
$$(\textbf{refl\_equal}\ \mathsf{A}\ f(In(\pi_1),...,In(\pi_n)))$$
$$Out(\pi_1)\ \ \Pi2Coq(\pi_1)\ \ \ldots\ \ Out(\pi_n)\ \Pi2Coq(\pi_n))$$

Example 6.3 Continue with our example. We translate π_Π into its counterpart in Coq-syntax as follows:

$$\Pi2Coq(concat\_g(f(x)\langle x \mapsto a\rangle \to x\langle x \mapsto a\rangle\ ,\ f(x)\langle x \mapsto b\rangle \to x\langle x \mapsto b\rangle))$$
$$\mapsto\!\!\!\twoheadrightarrow^* (\mathsf{eq\_concat\_2}\ \ f(a)\ \ f(b)\ \ [x_1,x_2 : \mathsf{A}]g(f(a),f(b)) = g(x_1,x_2)$$
$$(\textbf{refl\_equal}\ \mathsf{A}\ g(f(a),f(b)))\ a\ \ (\ell\ a)\ \ b\ (\ell\ b))$$

where ℓ is the label of the following Coq axiom:

$$(\mathsf{x} : \mathsf{A})\ \mathsf{f}(\mathsf{x}) = \mathsf{x}.$$

6.4 Soundness

Lemma 6.4 *If π is a proof term in Π-syntax, then $\Pi 2Coq(\pi)$ is a proof of equality $In(\pi) = Out(\pi)$ in Coq.*

Proof. By induction on the length of the derivation π represents. Consider different cases of $\Pi 2Coq$:

rewrite step at root position This is the basic case.

$\Pi 2Coq(\pi) = (\ell\ \sigma_{x_1}\ \ldots\ \sigma_{x_n})$ is a proof of $l\langle\sigma\rangle = r\langle\sigma\rangle\ \equiv\ In(\pi) =\ Out(\pi)$.

trans Lemma lm_eq is a proof of $Out(\pi_1) = In(\pi_2)$. By induction hypothesis, $\Pi 2Coq(\pi_2)$ is a proof of $In(\pi_2) = Out(\pi_2)$. Hence, by the definition of **trans_equal**, (**trans_equal** A $Out(\pi_1)$ $In(\pi_2)$ $Out(\pi_2)$ lm_eq $\Pi 2Coq(\pi_2)$) is a proof of $Out(\pi_1) = Out(\pi_2)$.

Furthermore, by induction hypothesis, $\Pi 2Coq(\pi_1)$ is a proof of $In(\pi_1) = Out(\pi_1)$ and so, $\Pi 2Coq(trans(\pi_1, \pi_2))$ is a proof of $In(\pi_1) = Out(\pi_2) \equiv In(trans(\pi_1, \pi_2)) = Out(trans(\pi_1, \pi_2))$.

refl $\Pi 2Coq(refl(t))$ is a proof of $t = t \equiv\ In(refl(t)) = Out(refl(t))$ by the definition of **refl_equal**.

ctx By induction hypothesis, $\Pi 2Coq(\pi)$ is a proof $In(\pi) = Out(\pi)$. By the definition of ctx_t, $\Pi 2Coq(ctx(t[CTX]_p, \pi))$ is a proof of $t[In(\pi)]_p = t[Out(\pi)]$
$\equiv\ In(ctx(t[CTX]_p, \pi)) = Out(ctx(t[CTX]_p, \pi))$.

concat_f Consider the predicate $\Phi\ \equiv\ \lambda x_1 \ldots x_n : \mathsf{A}.f(In(\pi_1), ..., In(\pi_n)) = f(x_1, ..., x_n)$, we have: (**refl_equal** A $f(In(\pi_1), \ldots, In(\pi_n))$) is a proof of

$f(In(\pi_1), ..., In(\pi_n)) = f(In(\pi_1), ..., In(\pi_n))\ \equiv (\Phi\ In(\pi_1)\ \ldots\ In(\pi_n))$ and by induction hypothesis, $\Pi 2Coq(\pi_1), ..., \Pi 2Coq(\pi_n)$ are respectively a proof of $In(\pi_1) = Out(\pi_1), \ldots, In(\pi_n) = Out(\pi_n)$. By the definition of eq_concat_n, $\Pi 2Coq(concat\_f(\pi_1, ..., \pi_n))$ is a proof of $(\Phi\ Out(\pi_1)\ \ldots\ Out(\pi_n))$
$\equiv\ f(In(\pi_1), ..., In(\pi_n)) = f(Out(\pi_1), ..., Out(\pi_n)) \equiv\ In(concat\_f(\pi_1, ..., \pi_n)) = Out(concat\_f(\pi_1, ..., \pi_n))$.

□

Lemma 6.5 *If π is a proof term in $\rho\sigma$-syntax of derivation $t \to^* s$, then $In(\rho\sigma 2\Pi(\pi)) = t$ and $Out(\rho\sigma 2\Pi(\pi)) = s$.*

Proof. By induction on the length of the derivation π represents. Consider different cases of $\rho\sigma 2\Pi$ (see figure 3):

Step This is the basic case: $In(\rho\sigma 2\Pi(t \to s)) = In(t \to s) = t$ while $Out(\rho\sigma 2\Pi(t \to s)) = Out(t \to s) = s$.

Concat $In(\rho\sigma 2\Pi(t_1;\ldots;t_n)) = In(trans(\rho\sigma 2\Pi(t_1), trans(t_2;\ldots;t_n)))$ $= In(\rho\sigma 2\Pi(t_1)) = t$ due to induction hypothesis. Similarly, $Out(\rho\sigma 2\Pi(t_1;\ldots;t_n))$ $= Out(trans(\rho\sigma 2\Pi(t_1), trans(t_2;\ldots;t_n))) = Out(trans(t_2;\ldots;t_n)) = \ldots =$ $Out(\rho\sigma 2\Pi(t_n)) = s$ due to induction hypothesis.

Refl In this case, no rewrite step has been performed in π and $t = s$.
Therefore, $In(\rho\sigma 2\Pi(refl(f(t_1,...,t_n)))) = f(t_1,...,t_n) = t$ while
$Out(\rho\sigma 2\Pi(refl(f(t_1,...,t_n)))) = f(t_1,...,t_n) = s$.

Disj1 By induction hypothesis, $In(ctx(f(t_1,...,t_i[CTX]_p,...,t_n), \rho\sigma 2\Pi(t_i|_p)))$
$= f(t_1,...,t_i[In(t_i|_p)],...,t_n) = t$.
Similarly, $Out(ctx(f(t_1,...,t_i[CTX]_p,...,t_n), \rho\sigma 2\Pi(t_i|_p)))$
$= f(t_1,...,t_i[Out(t_i|_p)],...,t_n) = s$.

Disj2 By induction hypothesis, $In(concat\_f(\rho\sigma 2\Pi(t_1),...,\rho\sigma 2\Pi(t_n)))$
$= f(In(\rho\sigma 2\Pi(t_1)),...,In(\rho\sigma 2\Pi(t_n))) = t$
Similarly, $Out(concat\_f(\rho\sigma 2\Pi(t_1),...,\rho\sigma 2\Pi(t_n)))$
$= f(Out(\rho\sigma 2\Pi(t_1)),...,Out(\rho\sigma 2\Pi(t_n))) = s$

□

Theorem 6.6 (Soundness) *If π is a proof term in $\rho\sigma$-syntax of derivation $t \to^* s$, then $\Pi 2Coq(\rho\sigma 2\Pi(\pi))$ is a proof of equality $t = s$ in Coq.*

Proof. By lemma 6.4, $\Pi 2Coq(\rho\sigma 2\Pi(\pi))$ is a proof of $In(\rho\sigma 2\Pi(\pi)) = Out(\rho\sigma 2\Pi(\pi))$ which is equivalent to $t = s$ due to lemma 6.5. □

7 Implementation

In our implementation, ELAN and Coq work on the same TRS $\mathcal{R}$. On one hand, the rules in $\mathcal{R}$ are specified as axioms in Coq: the AC function symbols require adding associative and commutative axioms. On the other hand, $\mathcal{R}$ is also specified as an ELAN specification. Generality is ensured since our implementation is completely independent of $\mathcal{R}$.

The aim of this implementation is to generate from each derivation $t \to^*_{\mathcal{R}} s$ in ELAN a proof term of proposition $t = s$ in Coq. We first implement a tracing mechanism for syntactic rewriting and AC rewriting in ELAN. The generated trace includes the rewriting context, the used substitution and the applied rule in each rewrite step. We next transform the trace into a proof term in $\rho\sigma$-syntax by a module written itself in ELAN. This module also *normalises* this proof term in order to reduce its size by the rules in figure 2. A set of lemmas and their proofs in Coq is then generated from this normalised proof term via the translation described in section 6. These lemmas include the main claim which states that $t = s$. Finally, all generated lemmas are automatically checked in Coq.

Equality modulo AC When working with AC rewriting, a decision procedure for equality modulo AC is needed in Coq since ELAN works on terms in canonical form. An approach to implement this decision procedure is given by *ordered rewriting* proposed in [21]. Equality modulo AC between two ground terms can be decided by a TRS which includes three following syntactic (conditional) rewrite rules for each AC function symbol f_{AC} in the signature:

$$\begin{array}{lll}
f_{AC}(x, f_{AC}(y,z)) & \to f_{AC}(y, f_{AC}(x,z)) & \text{if } (x \succ_{lpo} y) \\
f_{AC}(x,y) & \to f_{AC}(y,x) & \text{if } (x \succ_{lpo} y) \\
f_{AC}(f_{AC}(x,y),z) & \to f_{AC}(x, f_{AC}(y,z)) &
\end{array}$$

where $\succ_{lpo}$ denotes the lexicographic path ordering. Intuitively, these rules implement a sorting algorithm (bubble sort) on the subterms of a ground term. Two ground terms are equal modulo AC if and only if their sorted forms are syntactically equal. The search for a derivation from a term to its sorted form is done in ELAN. Coq replays this derivation later by checking its proof term as described in this paper or by using the Coq/ELAN interface for syntactic rewriting described in [2].

Benchmarking We give here some performance data yielded by testing our implementation on the TRS given by completing an Abelian group and that is composed of the following rewrite rules (+ being a AC function symbol):

$$\begin{array}{rcc}
x + 0 \to x & -0 \to 0 & -(x+y) \to (-x) + (-y) \\
x + (-x) \to 0 & -(-x) \to x &
\end{array}$$

This TRS is expressed in a natural way by the ELAN specification in figure 5. Syntactic function symbols and their arity are described in section Ops. Section ACOps contains the AC function symbols (with arity 2). In section Type, we need to precise the carrier type used in Coq since this type is required in building Coq proof terms. Section Vars declares the variables used in defining rewrite rules. These rules are given in section Rules with their label in the following syntax: $[label]\ LHS \to RHS$.

The experiment consists in normalising randomly generated terms using the leftmost-innermost strategy of ELAN and checking the corresponding proof term in Coq to prove that every term is equal to its normal form.

We used a PC Pentium III 860 Mhz running Linux for these tests. Time is measured in second. Table 1 shows that the number of generated lemmas and the proof term size of the main theorem is linear in the number of rewrite steps. It is not easy to estimate the proof term size since the effectiveness of the optimisations in size reducing depends on the positions of contracted redexes. These optimisations generate some auxiliary lemmas (see section 5.3). The proofs of these lemmas in Coq are not expensive. On the contrary, the lemmas for equality modulo AC are costly since we presently use the Coq/ELAN

```
specification abelian_group
Ops
      Opp:1 Zero:0
ACOps
      Plus
Type
      A
Vars
      X_1 X_2
Rules
       [neutral] (Plus X_1 Zero) -> X_1
      [Opp_Null] (Opp Zero) -> Zero
       [Opp_Opp] (Opp (Opp X_1)) -> X_1
      [Opp_Plus] (Opp (Plus X_1 X_2)) ->
                                  (A_Plus (Opp X_1) (Opp X_2))
        [invert] (Plus X_1 (Opp X_1)) -> Zero
end of specification
```

Fig. 5. ELAN specification for Abelian groups

Size	Steps	Normalisation in ELAN	Proof term translation in ELAN	Proof checking in Coq		Number of lemmas	Proof term size
				$=_{AC}$	Total		
20	9	0	0.06	3	4	20	723
40	18	0	0.24	9	10	31	1608
35	33	0.02	0.37	16	18	68	3238
55	50	0.03	0.96	24	29	96	7266
70	82	0.04	1.89	35	45	203	13315

Table 1
Benchmark on Abelian groups

interface [2] to prove them and the replaying process in Coq is quite time consuming. In other words, term rewriting proofs in ELAN are not only *checked* but *partially replayed* in Coq.

8 Conclusions

We have described a representation of the proof term for term rewriting in $\rho\sigma$-calculus. This representation allows to communicate term rewriting proofs with other systems by translating proof terms into their syntaxes. One of these translations (from $\rho\sigma$-syntax into Coq-syntax) has been described and implemented. As a result, term rewriting proofs in ELAN can be checked in Coq. In this translation we only need two Coq constants which provide equality proofs by reflexivity (**refl_equal**) and transitivity (**trans_equal**). Most of proof checkers (*e.g.* ALF, LEGO) also offer these constants and hence, the translations into their proof term syntaxes can be given by adapting $\Pi 2 Coq$.

Translating logical proofs between theorem provers has been studied by several researchers [24,32]. This is not always a simple task since they need to bridge the gap between different logical foundations on which these systems are based. Our work is more restrictive since we only consider equational proofs. Therefore, we can avoid the semantics issues and concentrate on the syntax of proof terms. In [35], Wong traces HOL proofs in order to check them by an independent proof checker which has been developed itself by the author. On the contrary, as what is done in [7], we use Coq kernel as proof checker. This fact requires us to respect Coq-syntax when building proof terms. However, we think that Coq kernel is more reliable since its logical foundation has been well-studied. Moreover the kernel has formally been proved consistent itself [3]. In [31], Twelf [29] has been chosen for checking proofs. The authors have defined their own equality relation and its properties in this framework. We think that it is not difficult to adapt our translation process, namely the procedure $\Pi 2 Coq$, to generate proof terms for checking in Twelf. Moreover, since proof terms in Π-syntax are already compacted, the generated proof terms in Twelf-syntax will be more concise than in the format used in [31].

Some attempts [8,2] have been done in order to get efficient term rewriting in Coq using *reflection* method. This approach is adequate for syntactic rewriting but when considering AC rewriting it becomes less obvious since pattern matching modulo AC is not simple enough to be efficiently performed in Coq.

The next step of this work is to implement an interface in Coq which allows the user to send the equalities they want to prove to ELAN. This latter performs these proofs and sends back the corresponding proof terms for checking in Coq.

One main advantage of ELAN is its set of strategies which allows the user to control rewriting and hence, to implement sophisticated proof search procedures [30,4]. Generating proof terms of these strategies helps to check a proof done by ELAN and to export it to other systems. Presently, only proof terms of the normalisation strategies in ELAN are generated but it seems not difficult to extend the formalism (and the implementation) to the complete set of ELAN strategies.

Another use of proof terms is to analyse and to debug ELAN programs. To this end, not only the proof terms of successful rewrite steps but also that of failure should be stored in order to understand why a rule or a strategy failed to apply on some term. At this stage, we need to deal with the space problem due to the size of proof terms. Sharing gives a means to overcome this problem. Some public libraries like ATerm [34] offer a term representation with maximal sharing. Integrating ATerm in ELAN is being investigated.

Acknowledgements I am indebted to Claude Kirchner for numerous discussions and to some anonymous referees for their helpful remarks.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In ACM, editor, *Conf. Rec. 17th Symp. POPL*, pages 31–46, 1990.

[2] C. Alvarado and Q-H. Nguyen. ELAN for equational reasoning in Coq. In J. Despeyroux, editor, *Proc. of 2nd Workshop on Logical Frameworks and Metalanguages*. Institut National de Recherche en Informatique et en Automatique, ISBN 2-7261-1166-1, June 2000.

[3] B. Barras. *Auto-validation d'un système de preuves avec familles inductives.* PhD thesis, Université Paris VII, November 1999.

[4] E. Beffara, O. Bournez, H. Kacem, and C. Kirchner. Verification of timed automata using rewrite rules and strategies. In N. Dershowitz and A. Frank, editors, *Proc. BISFAI 2001*, June 2001.

[5] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3(1–2):203–216, 1987.

[6] S. Berghofer and T. Nipkow. Proof terms for simply typed higher order logic. In J. Harrison and M. Aagaard, editors, *Proc. 13th Int. Conf. TPHOL*, volume 1869 of *Lecture Notes in Computer Science*, pages 38–52. Springer-Verlag, 2000.

[7] M. Bezem, D. Hendriks, and H. de Nivelle. Automated proof construction in type theory using resolution. In D. McAllester, editor, *Proc. 17th Int. Conf. CADE*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 148–163. Springer-Verlag, 2000.

[8] S. Boutin. Using reflection to build efficient and certified decision procedures. In M. Abadi and T. Ito, editors, *Proc. of the 3rd Int. Symp. TACS*, number 1281 in Lecture Notes in Computer Science, pages 515–529. Springer-Verlag, 1997.

[9] H. Cirstea. *Calcul de réécriture : fondements et applications.* PhD thesis, Université Henri Poincaré - Nancy I, October 2000.

[10] H. Cirstea and C. Kirchner. The rewriting calculus — Part I *and* II. *Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.

[11] H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In F. Honsell, editor, *Proc. of FOSSACS*, volume 2030 of *Lecture Notes in Computer Science*, pages 166–180, April 2001.

[12] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76, 1988.

[13] P-L Curien, T Hardin, and J-J Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, March 1996.

[14] E. Denney. A prototype proof translator from HOL to Coq. In M. Aagaard and J. Harrison, editors, *Proc. 13th Int. Conf. TPHOL*, volume 1869 of *Lecture Notes in Computer Science*, pages 108–125. Springer-Verlag, 2000.

[15] N. Dershowitz and J-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990.

[16] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. Rapport de Recherche 3400, Institut National de Recherche en Informatique et en Automatique, April 1998. `ftp://ftp.inria.fr/INRIA/publication/RR/RR-3400.ps.gz`.

[17] F. Gadducci. *On the Algebraic Approach to Concurrent Term Rewriting.* PhD thesis, Università di Pisa, January 1996.

[18] Claude Kirchner, Hélène Kirchner, and M. Vittek. Designing CLP using Computational Systems. In P. Van Hentenryck and S. Saraswat, editors, *Principles and Practice of Constraint Programming.* The MIT press, 1995.

[19] J.W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 1, pages 1–117. Oxford University Press, Oxford, 1992.

[20] LogiCal/INRIA. The Coq homepage. `http://coq.inria.fr`.

[21] U. Martin and T. Nipkow. Ordered rewriting and confluence. In M.E. Stickel, editor, *Proc. 10th Int. Conf. Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 366–380. Springer-Verlag, 1990.

[22] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[23] J-Y. Moyen. System presentation: an analyser of rewriting systems complexity. In Mark van den Brand and Rakesh Verma, editors, *Prceedings of the second RULE workshop*, volume 59. Elsevier Science Publishers B. V. (North-Holland), 2001. In this volume.

[24] P. Naumov, M-O. Stehr, and J. Meseguer. The HOL/NuPRL proof translator: A practical approach to formal interoperability. In R.J. Boulton and P.B. Jackson, editors, *Proc. 14th Int. Conf. TPHOL*, volume 2152 of *Lecture Notes in Computer Science*, pages 329–345. Springer-Verlag, 2001.

[25] University of Utrecht. The STRATEGO homepage. `http://www.stratego-language.org`.

[26] C. Paulin-Mohring. Extracting ω's programs from proofs in the calculus of constructions. In ACM, editor, *Proc. 16th Symp. POPL, January 11–13, 1989, Austin, TX*, pages 89–104. ACM Press, 1989.

[27] C. Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J. F. Groote, editors, *Proc. of the 1st Int. Conf. TLCA*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345, Berlin, 1993. Springer-Verlag.

[28] PROTHEO/LORIA. The ELAN homepage. `http://elan.loria.fr`.

[29] Carnegie Mellon School of Computer Science. The twelf project. `http://www-2.cs.cmu.edu/~twelf/`.

[30] J. Stuber. Experiments with an implementation of Extended Narrowing And Resolution in the rewriting language ELAN (system description). Available at `http://www.loria.fr/~stuber/software`, December 2000.

[31] A. Stump and D. L. Dill. Generating proofs from a decision procedure. In *Proc. of Workshop on Run-Time Result Verification*, July 1999. Available at `http://sprout.Stanford.EDU/~stump`.

[32] T. Tammet and J.M. Smith. Optimized encodings of fragments of type theory in first-order logic. *JLC: Journal of Logic and Computation*, 8, 1998.

[33] R. Verma and S. Senanayake. LR^2: A laboratory for rapid term graph rewriting. In P. Narendran and M. Rusinowitch, editors, *Proc. 10th Int. Conf. RTA*, volume 1631 of *Lecture Notes in Computer Science*, pages 252–255. Springer-Verlag, 1999.

[34] Centrum voor Wiskunde en Informatica (CWI). The ATerm homepage. `http://www.cwi.nl/projects/MetaEnv/aterm/`.

[35] W. Wong. Validation of HOL proofs by proof checking. *Formal Methods in System Design: An International Journal*, 14(2):193–212, 1999.