Certifying Term Rewriting Proof in ELAN

Quang-Huy Nguyen

Electronic Notes in Theoretical Computer Science 59 No. 4 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume59.html 18 pages

Certifying Term Rewriting Proofs in ELAN *(work in progress)*

Quang-Huy Nguyen

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. We consider the proof term of term rewriting and propose a formalism based on the rewriting calculus with explicit substitutions ($\rho\sigma$-calculus). This formalism is helpful in analysing and in debugging rule-based programs. Moreover, 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 term rewriting but also with term 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 certify the whole proof and to communicate it with other systems [22,13]. In some proof assistants (*e.g.* ALF, Coq, LEGO), the proof term of all deduction steps is explicitly stored. In other systems, proof term memorising was not given the first priority in its design but some attempts have been done to get this feature, for example, in Isabelle [5] or in HOL [29].

Term rewriting has been shown to be a good environment for programming and proving. In a proof assistant, term rewriting is very useful since it eases equational reasoning and considerably simplify proofs by abstracting the computational arguments [15]. However, in a system like Coq, 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 its proof term. This fact poses a serious space problem for Coq kernel in certifying these proofs.

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

Besides, the efficiency of term rewriting in Coq is also limited since it 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 a computation but not a proof. 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 [28], and the stored information cannot be seen as proof term of the computation.

Hence, proof assistants must trust in programming environments and consider these computations as axioms in its proof. This approach is not completely reliable since these computations are not certified. In other words, when correctness is crucial, rule-based programs should return the proof term of their computation.

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 behaviours (*e.g.* termination) of rule-based programs can be significantly improved. Recently, some systems (*e.g.* ELAN [25], STRATEGO [23]) 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, 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 in [2] since pattern matching is deterministic and not costly (linear in the size of pattern). 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 [4]) and hence, the trace 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 *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 of term rewriting) is desirable since this allows to analyse the program execution and facilitates the syntactic manipulations such as compacting or translating this trace.

The *rewriting calculus* [9,8] 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 [8] 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 term rewriting that is in first order 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 term rewriting 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 [18]. This proof term is then checked by Coq kernel 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 of term rewriting in this syntax. The proof term syntax of equality in Coq is presented in section 5. Section 6 describes the translation between two above syntaxes. We describe and discuss our current implementation in section 7 before giving some conclusions.

2 Preliminaries

We mostly use the notations introduced in [14] and [17]. A *signature* Σ consists of a set $\mathcal{V}$ of variables and a set $\mathcal{F}$ of function symbols. The arity of a function symbol $f \in \mathcal{F}$ is denoted by $arity(f)$. The set of variables of a term t is written $\mathcal{V}ar(t)$. Term t is called *ground* if $\mathcal{V}ar(t)$ is empty. A *position* within a term is represented by a sequence of natural numbers describing the path from the root of term to the head of the subterm at that position. The *root position* is an empty sequence and is denoted by ϵ. A subterm rooted at position p of a term t is denoted by $t|_p$. By $t[s]_p$ we denote the term t whose subterm at position p is replaced by a term s. A *substitution* is a mapping from the variables of $\mathcal{V}$ to terms. The identity substitution is denoted by $\mathbb{ID}$. By σ_x

we denote the term which instantiates the variable x in the substitution σ. If σ is a substitution, then $t\sigma$ denotes the result of applying σ on t. We write $t\{x \mapsto s\}$ the term t in which each occurrence of variable x is replaced by a term s.

A rewrite rule is written $l \to r$ 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 $l \to r$ 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 $t \to_{\mathcal{R}} s$ and the reflexive-transitive closure of the binary relation $\to_{\mathcal{R}}$ by $\to^*_{\mathcal{R}}$. 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 $l_1 \to r_1$ and $l_2 \to r_2$ 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 $l \to r$ **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 $c\sigma$ can be evaluated to $True$ where σ is the used substitution.

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. Replacing syntactic equality by equality modulo AC in the definition of rewrite step gives AC rewriting relation. The β-conversion relation is denoted by $=_\beta$. By $l \ll^?_T t$ we denote the matching problem between the term t and the pattern l in the equational theory T. A solution of this problem is a 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 $t : \mathsf{A}$ means that the type of t is A. The arrow symbol ($\to$) in a type expression denotes a functional type and is right associative. The application of the term f on the term t is denoted by $(f\ t)$. The term abstractor (λ) is written [], *e.g.* $[x, y : \mathsf{A}; z : \mathsf{B}]x$ denotes $\lambda xy : \mathsf{A}\lambda z : \mathsf{B}.x$. The type abstractor ($\Pi$) is written (), *e.g.* $(x, y : \mathsf{A}; z : \mathsf{B})x$ denotes $\Pi xy : \mathsf{A}\Pi z : \mathsf{B}.x$.

In this paper, the symbol $\twoheadrightarrow$ is used to give the specification of rewrite rule.

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 [1,11].

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

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

In the term syntax, $t \to u$ denotes a rewrite rule or a ρ-abstraction, $[t](u)$ represents the application of t on u, the application of the 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 $\rho\sigma$-calculus. The rule **F***ire* describes the application of a rewrite rule ($l \to r$) at root position of a term (t). This rewrite step replaces t by a set of instantiated RHS $r\sigma$ 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*	$[l \to r](t)$	$\mapsto\!\!\!\!\rightarrow \{r\langle\sigma\rangle\}$ where $\sigma \in Sol(l \ll^?_\emptyset t)$
(**C**)*ongruence*	$[f(s_1,...,s_n)](f(t_1,...,t_n))$	$\mapsto\!\!\!\!\rightarrow \{f([s_1](t_1),...,[s_n](t_n))\}$
(**C**)*ong_*(**f**)*ail*	$[f(s_1,...,s_n)](g(t_1,...,t_n))$	$\mapsto\!\!\!\!\rightarrow \emptyset$
(**D**)*istrib*	$[\{s_1,...,s_n\}](t)$	$\mapsto\!\!\!\!\rightarrow \{[s_1](t),...,[s_n](t)\}$
(**B**)*atch*	$[s](\{t_1,...,t_n\})$	$\mapsto\!\!\!\!\rightarrow \{[s](t_1),...,[s](t_n)\}$
(**S**)*witch*	$s \to \{t_1,...,t_n\}$	$\mapsto\!\!\!\!\rightarrow \{s \to t_1,\ldots,s \to t_n\}$
(**O**)*pOnSet*	$f(s_1,...,\{t_1,...,t_m\},...,s_n\}$	$\mapsto\!\!\!\!\rightarrow \{f(s_1,...,t_1,...,s_n),...,$ $f(s_1,...,t_m,...,s_n)\}$
(**Fl**)at	$\{s_1,...,\{t_1,...,t_m\},...,s_n\}$	$\mapsto\!\!\!\!\rightarrow \{s_1,...,t_1,...,t_m,...,s_n\}$

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 [8]. In this work, we only use the rule

$(\mathbf{I})d\ \ t\langle\mathbb{ID}\rangle \twoheadrightarrow\!\!\!\!\!\!\mapsto\ t$ which states that applying an identity substitution does not change a term. In the sequel, $\rightarrow_{\rho\sigma}$ denotes the reduction relation by the rules in figure 1 and $\mathbf{I}d$.

4 Proof Term of 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 $t \rightarrow^*_{\mathcal{R}} s$ if $[\pi](t) \rightarrow^*_{\rho\sigma} \{s\}$.

The proof term of the identity derivation $t \rightarrow t$ is given by the $\rho\sigma$-term $\mathbf{id} = [x \rightarrow x]$ since $\rho\sigma$-derivation: $[\mathbf{id}](t) = [x \rightarrow x](t) \rightarrow_{\mathbf{F}} \{t\}$ for all term t.

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

Proof. By the following derivation:

$$[l\langle\sigma\rangle \rightarrow r\langle\sigma\rangle](l\sigma) \rightarrow_{\mathbf{F}} \{r\langle\sigma\rangle\langle\mathbb{ID}\rangle\} \rightarrow_{\mathbf{I}} \{r\langle\sigma\rangle\} \equiv \{r\sigma\}$$

□

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

Proof. By the following derivation:

$$\begin{array}{ll} [l\langle\sigma\rangle \rightarrow [True \rightarrow r\langle\sigma\rangle]([\pi_c](c\langle\sigma\rangle))](l\sigma) \rightarrow_{\mathbf{F}} & [True \rightarrow r\langle\sigma\rangle]([\pi_c](c\langle\sigma\rangle)) \\ \rightarrow^*_{\rho\sigma} [True \rightarrow r\langle\sigma\rangle](\{True\}) & \rightarrow_{\mathbf{B}} \{[True \rightarrow r\langle\sigma\rangle](True)\} \\ \rightarrow_{\mathbf{F}} \{\{r\langle\sigma\rangle\}\} & \rightarrow_{\mathbf{Fl}} \{r\langle\sigma\rangle\} \equiv \{r\sigma\} \end{array}$$

□

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

Proof. By the following derivation:

$$\begin{array}{ll} [f(\mathbf{id}, ..., \pi_i, ..., \mathbf{id})](f(t_1, ..., t_i, ..., t_n)) & \\ \rightarrow_{\mathbf{C}} \{f([\mathbf{id}](t_1), ..., [\pi_i](t_i), ..., [\mathbf{id}](t_n)\} & \rightarrow^*_{\rho\sigma} \{f(t_1, ..., \{s_i\}, ..., t_n)\} \\ \rightarrow_{\mathbf{O}} \{\{f(t_1, ..., s_i, ..., t_n)\}\} & \rightarrow_{\mathbf{Fl}} \{f(t_1, ..., s_i, ..., t_n)\} \end{array}$$

□

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:

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

□

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

Compacting the proof term The proof term needs to be concise so that it can efficiently be checked by a proof assistant. The rules in figure 2 allow to reduce the size of the proof term 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 term is strictly decreased after the application of each rule, and confluent since the only critical pair formed by Id_elim_l and Id_elim_r is trivial.

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

Fig. 2. Reduction rules of the proof term in $\rho\sigma$-syntax

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 [21,20]. Our formalism is more related to the representation proposed in Gadducci's work on *flat rewriting logic* [16] since we orient towards a practical implementation and hence, do not allow the nesting of 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 [10]. 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 with inductive data types). Proofs in Coq are constructive and assuming excluded middle is up to the user. By Curry-Howard isomorphism, proposition is interpreted as type and 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. The 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 by Coq kernel. This approach has some advantages: correctness is ensured by the consistency of a tiny kernel, a certified (functional) program can be extracted from the proof of its specification [24], etc. However, this mechanism requires to keep all information concerning each deduction step in the proof term and sometimes, poses a serious problem of space.

5.1 Equality

Coq uses Leibniz equality. Let A be a type and **Prop** be the type which represents the propositions, the equality between terms of type A ($=_{\mathsf{A}}$) is defined by the following logical rules:

$$\frac{x, y : \mathsf{A}}{x =_{\mathsf{A}} y : \mathbf{Prop}}\ \text{Formation} \qquad \frac{x : \mathsf{A}}{x =_{\mathsf{A}} x}\ \text{Introduction}$$

$$\frac{\Phi : \mathsf{A} \to \mathbf{Prop} \quad x, y : \mathsf{A} \quad x =_{\mathsf{A}} y \quad (\Phi\ x)}{(\Phi\ y)}\ \text{Elimination}$$

The elimination rule is used to simplify a goal by term rewriting. The proof term of the rewrite step $(\Phi\ t) \to (\Phi\ s)$ contains the rewriting context Φ, the instantiated LHS t, the instantiated RHS s and a proof of $t =_{\mathsf{A}} s$. Since the sequel of this paper is independent of A, we will use $=$ to denote equality instead of $=_{\mathsf{A}}$.

Leibniz equality is also reflexive and transitive. Two following pre-defined constants are used to build proofs by reflexivity and transitivity:

refl_equal : $(\mathsf{A} : \mathbf{Set}; x : \mathsf{A})x = x$.
If t is a term of type A, then (**refl_equal** A t) is a proof of $t = t$.

trans_equal : $(\mathsf{A} : \mathbf{Set}; x, y, z : \mathsf{A})x = y \to y = z \to x = z$.
Let t, s and v be three terms of type A. If π_1 and π_2 are respectively a proof of $t = s$ and $s = v$, then (**trans_equal** A $t\ s\ v\ \pi_1\ \pi_2$) is a proof of $t = v$.

Notice that these constants correspond to the case where the type of A is **Set**. The other case (A : **Prop**) requires two other constants but their usage is similar.

5.2 Rewrite rule

The rewrite rule $l \to r$ 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 may appear 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 step 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 the position p of the term t, the following lemma is added and proved:

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

The proof of this lemma consists simply in applying tactic **Rewrite** x = 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 (ctx_t s v π) 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:

$$\begin{aligned}\textbf{Lemma}\ \mathsf{eq\_concat\_n} : (x_1, ..., x_n : \mathsf{A}; \Phi : (\overbrace{\mathsf{A} \to \ldots \to \mathsf{A}}^{n \text{ times}} \to \textbf{Prop}))\\ (\Phi\ \mathsf{x_1} \ldots \mathsf{x_n}) \to (\mathsf{y_1} : \mathsf{A})\mathsf{x_1} = \mathsf{y_1} \to \ldots \to (\mathsf{y_n} : \mathsf{A})\mathsf{x_n} = \mathsf{y_n}\\ \to (\Phi\ \mathsf{y_1} \ldots \mathsf{y_n})\end{aligned}$$

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 **Rewrite** tactic: **Rewrite** $\mathsf{x_1} = \mathsf{y_1} \ldots$ **Rewrite** $\mathsf{x_n} = \mathsf{y_n}$. 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 (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 the $\rho\sigma$-syntax and the Coq-syntax in order to simplify the translation between them. Π-syntax makes the translation more open in the sense that one can parameterise it by the proof term syntax used in proof checker. Two operators $\rho\sigma 2\Pi$ and $\Pi 2Coq$ 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*

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

where s is a $\rho\sigma$-term; 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.

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 $Disj1$ separates the context from the proof term of a derivation performed in *one branch* of a function symbol. The rule $Disj2$ deals with the case where the rewrite steps are performed at disjoint branches of a function symbol.

6.3 $\Pi 2Coq$

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 figures 4.

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

rewrite step at root position The proof term in Coq-syntax is given by

$$
\begin{array}{lll}
Step & \rho\sigma 2\Pi(t \to s) & \mapsto\!\!\!\!\gg t \to s \\
Concat & \rho\sigma 2\Pi((t_1; \ldots; t_n)) & \mapsto\!\!\!\!\gg trans(\rho\sigma 2\Pi(t_1), \rho\sigma 2\Pi(t_2; \ldots; t_n)) \\
Refl & \rho\sigma 2\Pi(f(t_1, ..., t_n)) & \mapsto\!\!\!\!\gg refl(f(t_1, ..., t_n)) \\
 & & \text{if } \nexists\, i \ s.t. \ t_i \text{ contains some rewrite step} \\
Disj1 & \rho\sigma 2\Pi(f(t_1, ..., t_n)) & \mapsto\!\!\!\!\gg ctx(f(t_1, ..., t_i[CTX]_p, ..., t_n), \rho\sigma 2\Pi(t_i|_p)) \\
 & & \text{if } \exists!\ i \ s.t. \ t_i \text{ contains some rewrite step at position } p \\
Disj2 & \rho\sigma 2\Pi(f(t_1, ..., t_n)) & \mapsto\!\!\!\!\gg concat\_f(\rho\sigma 2\Pi(t_1), ..., \rho\sigma 2\Pi(t_n)) \\
 & & \text{if } \exists\ i, j \ s.t. \ t_i, t_j \text{ contain some rewrite step}
\end{array}
$$

Fig. 3. Rules for $\rho\sigma 2\Pi$

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

Fig. 4. Rules for In and Out

instantiating the corresponding axiom with the used substitution.

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

where $\{x_1, ..., x_n\} \equiv \mathcal{V}ar(l) \cap \mathcal{V}ar(r)$ 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**.

$$
\begin{array}{rl}
\Pi 2Coq(trans(\pi_1, \pi_2)) \mapsto\!\!\!\!\gg & (\textbf{trans\_equal}\ \mathsf{A}\ In(\pi_1)\ Out(\pi_1)\ Out(\pi_2)\ \Pi 2Coq(\pi_1) \\
 & (\textbf{trans\_equal}\ \mathsf{A}\ Out(\pi_1)\ In(\pi_2)\ Out(\pi_2)\ \mathsf{Im\_eq}\ \Pi 2Coq(\pi_2)))
\end{array}
$$

where lemma_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 ($T \equiv \emptyset$) and 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**.

$$\Pi 2Coq(refl(t)) \mapsto\!\!\!\!\rightarrow (\textbf{refl\_equal}\ \mathsf{A}\ t)$$

ctx The proof term in Coq-syntax is built using the added lemma ctx_t.

$$\Pi 2Coq(ctx(t[CTX]_p, \pi)) \mapsto\!\!\!\!\rightarrow (\mathsf{ctx\_t}\ In(\pi)\ Out(\pi)\ \Pi 2Coq(\pi))$$

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

$$\begin{aligned}\Pi 2Coq(concat\_f(\pi_1, ..., \pi_n)) \mapsto\!\!\!\!\rightarrow (&\mathsf{eq\_concat\_n}\ \ In(\pi_1)\ \ \ldots\ \ In(\pi_n)\\ &[x_1, ..., x_n : \mathsf{A}]f(In(\pi_1), ..., In(\pi_n)) = f(x_1, ..., x_n)\\ &(\textbf{refl\_equal}\ \mathsf{A}\ f(In(\pi_1), ..., In(\pi_n)))\\ &Out(\pi_1)\ \ \Pi 2Coq(\pi_1)\ \ \ldots\ \ Out(\pi_n)\ \Pi 2Coq(\pi_n))\end{aligned}$$

6.4 Soundness of the translation

Lemma 6.1 *If* π *is a proof term in* Π*-syntax, then* $\Pi 2Coq(\pi)$ *is a proof of the equality* $In(\pi) = Out(\pi)$ *in* Coq.

Proof. By induction on $\Pi 2Coq$.

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

Proof. By induction on $\rho\sigma 2\Pi(\pi)$ (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 6.3 (Soundness) *If π is a proof term in $\rho\sigma$-syntax of the derivation $t \to^* s$, then $\Pi 2Coq(\rho\sigma 2\Pi(\pi))$ is a proof of the equality $t = s$ in Coq.*

Proof. By lemma 6.1, $\Pi 2Coq(\rho\sigma 2\Pi(\pi))$ 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 6.2. □

7 Implementation

In our implementation, ELAN and Coq work on the same theory $\mathcal{T}h$. The axioms of $\mathcal{T}h$ are specified in Coq: the AC symbols are expressed by adding the associative and commutative axioms. On the other hand, this theory is completed to get a *confluent* and *terminating* TRS $\mathcal{R}$ that is specified by an ELAN specification.

The aim of this implementation is to generate from each derivation $t \rightarrow^*_{\mathcal{R}} s$ in ELAN a *Coq*-term that is a proof term of the proposition $t =_{\mathcal{T}h} s$ 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 $t =_{\mathcal{T}h} s$. 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 reduces term in canonical form. A simple approach to implement this decision procedure is given by *ordered rewriting* proposed in [19]. 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}:

$$\begin{aligned} f_{AC}(x, f_{AC}(y, z)) &\rightarrow f_{AC}(y, f_{AC}(x, z)) \text{ if } (x >_{lpo} y) \\ f_{AC}(x, y) &\rightarrow f_{AC}(y, x) \text{ if } (x >_{lpo} y) \\ f_{AC}(f_{AC}(x, y), z) &\rightarrow f_{AC}(x, f_{AC}(y, z)) \end{aligned}$$

where $>_{lpo}$ 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 an Abelian group which is composed of the following ax-

ioms (+ being a AC function symbol):

$$x + y = y + x \qquad x + 0 = x$$
$$(x + y) + z = x + (y + z) \qquad x + (-x) = 0$$

The canonical TRS returned by completing this theory is given by the following rewrite rules (+ being a AC function symbol):

$$x + 0 \to x \qquad -0 \to 0 \qquad -(x + y) \to (-x) + (-y)$$
$$x + (-x) \to 0 \qquad -(-x) \to x$$

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

Rewrite step	Normalisation in ELAN	Translation in ELAN	Proof checking in Coq	Number of lemmas	Proof term size of main theorem
9		0.06	4	20	723
18		0.24	10	31	1608
33	0.02	0.37	18	68	3238
50	0.03	0.96	29	96	7266
82	0.04	1.89	45	203	13315

Table 1
Benchmark on Abelian groups

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 very costly since we presently use the Coq/ELAN 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 of term rewriting in $\rho\sigma$-calculus. The formal notion of proof terms facilitates its communication between different systems by translating between 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 $\Pi 2Coq$.

Translating logical proofs between theorem provers has been studied by several researchers [22,6,27,29]. 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.

Some attempts [7,2] have been done in order to get efficient term rewriting in Coq using *reflection* method. This approach seems adequate for syntactic rewriting but when considering AC rewriting it becomes less obvious since pattern matching modulo AC is not trivial 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 searches for the proof and sends back the corresponding proof term in Coq-syntax to Coq for checking. At this stage, we also need completeness beside soundness. That is, if two terms are equal modulo the theory $\mathcal{T}h$ in Coq, then this equality is always provable in ELAN by rewriting with respect to $\mathcal{R}$. This statement is obvious if $\mathcal{R}$ is obtained by completing $\mathcal{T}h$.

One main advantage of ELAN is its set of strategies which allows the user to control rewriting and hence, to implement the sophisticated proof search procedures [26,3]. 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 term of successful rewrite step but also that of failure should be stored in order to understand why a rule or a strategy failed to apply on a 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 [12] 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 *Conf. Rec. 17th Symp. PPL*, 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] 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.

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

[5] 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 *LNCS*, page 38. Springer-Verlag, 2000.

[6] 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 *LNAI*. Springer-Verlag, 2000.

[7] S. Boutin. Using reflection to build efficient and certified decision procedures. In M. Abadi and T. Ito, editors, *Proc. of the 11th Int. Conf. TPHOL*, number 1281 in LNCS. Springer-Verlag, 1997.

[8] H. Cirstea. *Calcul de réécriture : fondements et applications*. PhD thesis, Université Henri Poincaré - Nancy I, 2000. October 25.

[9] 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.

[10] H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In F. Honsell, editor, *Proc. of FOSSACS*, volume 2030 of *LNCS*, pages 166–180, April 2001.

[11] 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.

[12] CWI. The ATerm homepage. http://www.cwi.nl/projects/MetaEnv/aterm/.

[13] 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 *LNCS*. Springer-Verlag, 2000.

[14] 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.

[15] 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.

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

[17] 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.

[18] LogiCal/INRIA. The Coq homepage. INRIA, http://coq.inria.fr.

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

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

[21] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Proceedings of CONCUR'96*, August 1996.

[22] P. Naumov, M-O. Stehr, and J. Meseguer. The HOL/NuPRL proof translator: A practical approach to formal interoperability. In *Proc. 14th Int. Conf. TPHOL*, 2001. To appear.

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

[24] 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.

[25] PROTHEO/LORIA. The ELAN homepage. LORIA, http://elan.loria.fr.

[26] 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.

[27] 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.

[28] R. Verma and S. Senanayake. LR^2: A laboratory for rapid term graph rewriting. In *Proc. 10th Int. Conf. RTA*, volume 1631 of *LNCS*, pages 252–255. Springer-Verlag, 1999.

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