ELAN *for Equational Reasoning in* Coq

CUIHTLAUAC ALVARADO
France Télécom Recherche & Développement, DTL/MSV/MFL
2, avenue Pierre Marzin, 22307 Lannion Cedex, France
(*e-mail:* Cuihtlauac.Alvarado@rd.francetelecom.fr)

QUANG-HUY NGUYEN
Loria – Inria Lorraine
615, rue du Jardin Botanique, BP 101, 54602 Villers-lès-Nancy Cedex, France
(*e-mail:* Quang-Huy.Nguyen@loria.fr)

Abstract

We describe an interface between Coq, an interactive theorem-prover based on type theory and ELAN, an automated deduction system based on rewriting logic. Our objective is to provide efficient tools to implement decision procedures using equational reasoning in Coq. ELAN is used as a rewrite engine. Reflection is used in Coq to translate back the delegated rewrite computations. A rewriting trace is returned from ELAN and replayed in Coq. Correctness is ensured by this replaying which builds a complete proof-term.

1 Introduction

Both automated deduction tools and proof assistants have strengths and weaknesses. Automated tools are fast and easy to use, but their logic is intrinsically limited. Proof assistants have a powerful logic (*i.e.* expressive) but require user interaction. Integration of these techniques is highly desirable. This work describes yet another interface between a pair of those systems, namely Coq and ELAN.

The Coq system (http://coq.inria.fr) is a proof-assistant based on the calculus of constructions, a typed lambda calculus with polymorphic and dependent types. The main extensions added in the calculus are: inductive and coinductive data types and fix point definitions. This logical framework is a constructive logic, assuming excluded middle is up to the user. The calculus of constructions can be used for program extraction: certified programs can be generated from proofs of their specification in Coq.

The ELAN system (Borovanský *et al.*, 1998) provides an environment for specifying and prototyping deduction systems in a language based on rewrite rules controlled by strategies. It offers a natural and simple logical framework for the combination of the computation and deduction paradigms, as it is backed up by the concepts of rewriting logic and rewriting calculus (Cirstea & Kirchner, 1999).

The rewriting technique is used since 1970 (Knuth & Bendix, 1970) to decide the equivalence between two terms in an equational theory. The main principle is to direct all equational axioms to get a *canonical* (confluent and terminating) rewrite system that represents the theory. In such a system, each term has an unique normal form. Checking equality between two terms is reduced to the identity of their normal forms. The derivation process from a term to its normal form is called *normalization*.

Coq keeps a proof-object for each reasoning. These proof-objects are first-class objects of the system (*i.e.* typed λ-terms), they store all the logical information needed to justify each reasoning step. This mechanism reduces Coq's soundness to the correction of its type-checking module and the user-assumed axioms. In the case of large equational reasoning type-checking becomes very time consuming.

The key ideas to address this issue are: delegate the normalization process to the automated deduction system, ELAN; replay using *reflection* the rewrite computation to build a reliable proof of equality in Coq. ELAN is used as a fast oracle which normalize terms. Reflection (internalization) permits making rewrite justifications once for all, at the rewrite rule level, without having to modify the core calculus. Hence, it enables a reasonably fast replaying using Coq's own computation mechanisms. The interface between the systems is the trace of the normalization process in the automated deduction system.

We advocate this approach by emphasizing the following points:

- the kernel of the proof-checker is unchanged;
- the rewriting trace is a simple interface mechanism between the systems;
- correctness is ensured because rewritings are replayed in Coq;
- reasonably good performance can be expected.

2 Related work

There are many attempts to integrate interactive theorem-provers and automated deduction tools. Two kinds of integration can be encountered: *trust* integration means the interactive prover accepts the result of the automated prover as a truth; *non-trust* integration means the proof is converted and replayed in the interactive prover to get a homogeneous proof (Calmet & Homann, 1996). Depending on the automated system, these works cover integration of:

- Model checking: various theorem-provers have an interface with a model checker. (Joyce & Seger, 1994) describes one of the firsts. Basically, model checking can't deal with infinite-state problems. In order to overcome this obstacle, a finite-state abstract model can be constructed, using a validity preservation property. The prover is used to establish and check this property (Rushby, 1999).
- Computer algebra: (Harrison & Théry, 1998) have embedded a computer algebra system (Maple) in a prover to solve purely mathematical problems. The key idea is to transfer the proof finding process to the computer algebra tool. The prover is only responsible for checking the result.
- Decision procedures: (Hurd, 1999; Ahrendt *et al.*, 1998; Boulton *et al.*, 1998; Bush, 1994) have implemented first-order decision procedure as tactics in various proof assistants. These tactics are activated when the prover encounters

a first-order formula. The formula is transfered to the automated prover that returns the proof.

The use of reflection in type-theories was proposed by H. Barendregt (Barthe *et al.*, 1996), it was called "two level approach". Independently S. Boutin successfully used reflection to implement in Coq a fast decision procedure based on normalization of ring terms (Boutin, 1997).

In HOL (Gordon & Melham, 1993) rewriting is handled using conversions, *i.e.* specialized tactics. This mechanism, which comes back from LCF (Paulson, 1983) is easy to use and expressive. Reasonable performance is achieved because of the abstract nature of the type of theorems (`thm`). Recording primitive inference steps is useless in HOL. Thus, a ML tactic can apply a large number of those steps at an affordable price.

Isabelle have even more powerful rewriting tactics (Paulson & Nipkow, 1999). Higher-order rewriting is possible, advanced control of the user defined term rewriting system is offered and performances seems to be good.

Our work is very similar with D. Howe's treatment of term-rewriting using partial reflection in Nuprl (Howe, 1988). In Nuprl the normalization process is implemented using a terminating function (*i.e.* a function computable by Nuprl, not a ML tactic) which approximates the LCF tactical `repeat`. We do not have such combinator, but it could be implemented easily, using the same trick as in Nuprl. Since we do not compute normalizations we only replays them it is useless.

Another way to improve the ease of use and performances of rewriting in Coq is Gilles Dowek's proposed "modulo" extension of the Calculus of Inductive Constructions (CiC, Coq's logical framework). This extension would allow the user to add rewrite rules to the standard $\beta\delta\iota$-reduction of the system.

In order to prove strong normalization for this extended system, reduction of the system (β, δ and ι) and user defined term-rewriting rules are handled in an unifying framework: higher-order rewriting. Strong normalization also have to be modular in order to allow dynamical loading of user defined rules. All these requirements leads to syntactical restrictions on user definable rules.

3 Tracing ELAN computation

The current version of ELAN includes an interpreter, a compiler, a library of ELAN modules, a user manual and examples of applications. Written in Java, the compiler generates the respective C code from an ELAN program. This C code is then compiled by a standard C compiler. In some applications, thanks to the compiler, the computational speed can reach several millions of rewriting steps per second.

The normalisation in ELAN is performed by unlabelled rules using a leftmost-innermost reduction strategy. However, the trace for Coq requires the labels of applied rules. So, a new normalisation using labelled rules is implemented in ELAN. The reduction strategy can be now parameterised: leftmost-innermost or leftmost-outermost. The traces are generated and stored in an efficient data structure during the computation.

3.1 Trace form for simple rewrite system

The trace built by ELAN during the normalisation of a term t is a list L_π of pairs (ω, ℓ) where ℓ is the label of applied rule and ω is the position of subterm. The top position of term is denoted by ϵ. The left-hand side and right-hand side of ℓ are respectively denoted by $lhs(\ell)$ and $rhs(\ell)$. Let us explain how L_π is built, first by considering an elementary rewrite step where only one rule is applied. There are two phases in an elementary rewrite step: pattern matching and replacement.

3.1.1 Pattern matching

Input: a term t and a set of rewrite rules $\mathcal{R}$
Output: a rule $\ell \in \mathcal{R}$, a position ω and a substitution σ, such that $t|_\omega = lhs(\ell)\sigma$ where $t|_\omega$ is the subterm of t at position ω and is called a redex.

In order to select a redex, the list of all pairs *(position, subterm at this position in t)* is recursively built. Each reduction strategy (leftmost-innermost, leftmost-outermost ...) (Klop, 1990) corresponds to a distinct list. All lists have the same set of elements but the orders between their elements are different.

Example 3.1
For the term $t = (+\ (+\ x\ y)\ z)$, the list

$$[(\epsilon, (+\ (+\ x\ y)\ z)); (.0, (+\ x\ y)); (.0.0, x); (.0.1, y); (.1, z)]$$

corresponds to leftmost-outermost strategy; whereas

$$[(.0.0, x); (.0.1, y); (.0, (+\ x\ y)), (.1, z); (\epsilon, (+\ (+\ x\ y)\ z))]$$

corresponds to leftmost-innermost strategy.

Then, each rule in $\mathcal{R}$ is tried with each subterm in the list until the first matching pair is found.

In summary, the result of a pattern matching step is composed of a rule $\ell \in \mathcal{R}$, a pair $(\omega, t|_\omega)$ and the respective matching substitution σ.

3.1.2 Replacement

This step builds the result $t' = t[rhs(\ell)\sigma]_\omega$ and generates a trace $\pi' = (\omega, \ell)$.

A normalisation process consists of repeating simple rewriting steps until a normal form is reached. This process returns the normal form and generates the trace $trace(normalise(t))$.

3.2 Trace form for conditional rewrite system

A conditional rewrite rule has the form: $\Gamma \Rightarrow l \to r$ where Γ is $s_1 = t_1 \wedge ... \wedge s_n = t_n$. In order to apply this rule on a term, the substitution σ found by pattern matching must satisfy all equations in Γ. In other words, after the matching step we need to verify if $s_1(\sigma) = t_1(\sigma) \wedge \ldots \wedge s_n(\sigma) = t_n(\sigma)$ holds before the replacement step.

Clearly, each equation $s_i(\sigma) = t_i(\sigma)$ can be checked by comparing the normal forms of $s_i(\sigma)$ and $t_i(\sigma)$ in the current theory. So, $2n$ normalisation are needed to check the condition. The traces of these normalisations must be included in the trace of conditional rewriting step:

$$\pi' = (\omega, \ell)[trace(normalise(s_i(\sigma)))||trace(normalise(t_i(\sigma)))] \text{ where } i = 1, \ldots, n.$$

The trace in the conditional part ($[\ldots]$) are called sub-trace. A sub-trace can be composed of several normalisation traces separated here by the symbol $||$. This format records the subcomputations that checks the satisfaction of conditions.

3.3 Compiling the normalisation strategies

A normalisation process is specified in ELAN by a rewrite system $\mathcal{R}$ and a reduction strategy (leftmost-innermost/leftmost-outermost). This specification is parsed and transformed into an internal format REF (Reduced ELAN Format). The ELAN compiler generates the C code from the input in format REF. In the case of a leftmost-innermost reduction strategy, the output consists of two procedures:

- $str\_norm(t, \omega)$ is described in figure 1. This procedure normalises in $\mathcal{R}$ a subterm t at position ω.
- $str\_visit(t)$ is described in figure 2. This procedure visits a term t and applies str_norm on each of its subterm.

Step 1: Apply pattern-matching automata of $\mathcal{R}$ on t
if $\exists \ell \in \mathcal{R}$ and σ, such that $lhs(\ell)\sigma = t$ **then** goto *Step 2* **else return** t
Step 2: $t' = rhs(\ell)\sigma$ and *add a pair*(ω, ℓ) *to the traces* L_π
if t' is a variable or a constant **then begin** $t'' = str\_norm(t', \omega)$; **return** t' **end**
else t' is $f(t'_1, \ldots, t'_n)$
1. **for** i **from** 1 **to** n **do** $t''_i = str\_norm(t'_i, \omega.i)$
2. $t'' = str\_norm(f(t''_1, \ldots, t''_n), \omega)$
3. **return** t''

Fig. 1. $str\_norm(t, \omega)$ procedure

In leftmost-outermost reduction strategy, a rewriting attempt must be done on top when the term is changed. The reduction procedure is described in figure 3.

3.4 Structure of traces

The space needed for the traces in practice is huge. When the term is big, the position strings (lists of natural numbers) become long and the number of rewriting steps required to obtain the normal form is also big. The space needed for normalisation traces is drastically increased. In case of conditional rewriting, the situation is even worse because we need space for subcomputations.

An analysis of the position strings in the traces shows that there are many shared

Step 1: $\omega = \epsilon$;
Step 2: **if** t is a variable or a constant **then return** $str\_norm(t, \omega)$
else t is $f(t_1, \ldots, t_n)$
1. $\omega' = \omega$
for i **form** 1 **to** n **do**
begin
$\omega = \omega.i$; $t'_i = str\_visit(t_i)$
end
2. $\omega = \omega'$; $t'' = str\_norm(f(t'_1, \ldots, t'_n), \omega)$
3. **return** t''

Fig. 2. $str\_visit(t)$ procedure

Step 1: $\omega = \epsilon$.
Rewrite eagerly on the top of term. In each rewriting step, *replace* the term by the rewriting result and *added* a pair (ϵ, ℓ) to the trace L_π where ℓ is the label of applied rule.
Step 2: After *Step 1*, any rewrite rule can be applied on top of term.
Visit the term from top to bottom and from left to right. *Update* ω is by the position of current subterm. In each subterm, the pattern-matching automata is applied to find out the first redex. If any redex can be found, *return* the current term. In other case, perform *only one rewriting step* on this redex, *replace* it by the rewriting result, *add* a pair (ω, ℓ) to the trace L_π and *go to Step 1*.

Fig. 3. Leftmost-outermost reduction procedure

parts between them. These parts can be factorised. A tree is a good data structure to do this factorisation. Each position string is stored in a branch of the tree. This tree is called a *position tree* and denoted by $\mathcal{T}$. The common part between different position strings becomes the shared edges of their respective branches.

The structure used for the set of rules $\mathcal{R}$ is a *table* or a *hash table* if the number of rules becomes huge. One rule corresponds to an entry in the table. The label stored in the trace identifies the rule.

The normalisation trace is a *list* whose each node is the trace of an elementary rewriting step. Such a node contains two pointers: the *first one* points to the entry of *applied rule* in the table of rules, the *second one* points to a *node* in tree $\mathcal{T}$.

Each node in $\mathcal{T}$ is labelled by an integer and consists of three pointers: the *first one* points to its ancestor, the *second one* points to the *next node* in the same level and the *last one* points to the *list of its descendants*. Three primitive operations in this tree are:

- **Initialisation**: allocate the root of tree. This node corresponds to the top position (ϵ).
- **Insertion**: insert a new position string to $\mathcal{T}$. Start at the root of tree and extract sequentially each position in the string. The first position is a descen-

dant of the root. The i^{th} position is a descendant of the $(i-1)^{th}$. Here is the principle for insertion:
At i^{th} position:

— If its ancestor has already a descendant whose the label is equal to its value, continue with the $(i+1)^{th}$ position.
— In other case, insert a new descendant to its ancestor. Label this node by its value. Update its three pointers. Continue with the $(i+1)^{th}$ position.

This process is repeated until the end of position string.

- **Extraction**: extract a position string from the tree. Start at a node and climb up to the root of tree by passing *from a node to its ancestor*. In each node, its label is concatenated to the position string. Note that the string found by this task is the inversion of the original position string and an inverse routine is needed here.

Example 3.2
Let us consider the groups. Table 1 shows the set of rewrite rules $[\ell] : l \to r$ returned from the completion procedure of Knuth-Bendix (ℓ is the label of rule).

[neutral_l] :	$(+\ 0\ x)\ \to\ x$	[Opp_Opp] :	$(-\ (-\ x))$	$\to$	x
[neutral_r] :	$(+\ x\ 0)\ \to\ x$	[assoc_invert_l] :	$(+\ (-\ x)\ (+\ x\ y))$	$\to$	y
[invert_l] :	$(+\ (-\ x)\ x)\ \to\ 0$	[assoc_invert_r] :	$(+\ x\ (+\ (-\ x)\ y))$	$\to$	y
[invert_r] :	$(+\ x\ (-\ x))\ \to\ 0$	[assoc] :	$(+\ (+\ x\ y)\ z)$	$\to$	$(+\ x\ (+\ y\ z))$
[Opp_Null] :	$(-\ 0)\ \to\ 0$	[Opp_Plus] :	$(-\ (+\ x\ y))$	$\to$	$(+\ (-\ y)\ (-\ x))$

Table 1. *Rewrite rules for groups*

The normalisation of term `(+ (- (- (- (+ (- c) (+ (+ (- b) b) b))))) (+ (- (- a)) (- (- a))))` using a leftmost-innermost strategy generates the trace in table 2. Figure 4 describes the structure used to store this trace.

1. invert_l [0;0;0;0;1;0]	5. Opp_Plus [0;0]	9. Opp_Opp [1;0]
2. neutral_l [0;0;0;0;1]	6. Opp_Opp [0;0;0]	10. Opp_Opp [1;1]
3. Opp_Plus [0;0;0]	7. Opp_Plus [0]	11. assoc []
4. Opp_Opp [0;0;0;0]	8. Opp_Opp [0;0]	

Table 2. *An example of trace*

4 Coq

Type-checking large equational reasoning proof-terms in Coq can be very time consuming. The need for efficient rewriting techniques in Coq was one of the main motivation of this work. *Reflection* takes advantage of Coq's computation mechanisms to improve type-checking performances of equational reasoning proof-terms which have been found by ELAN.

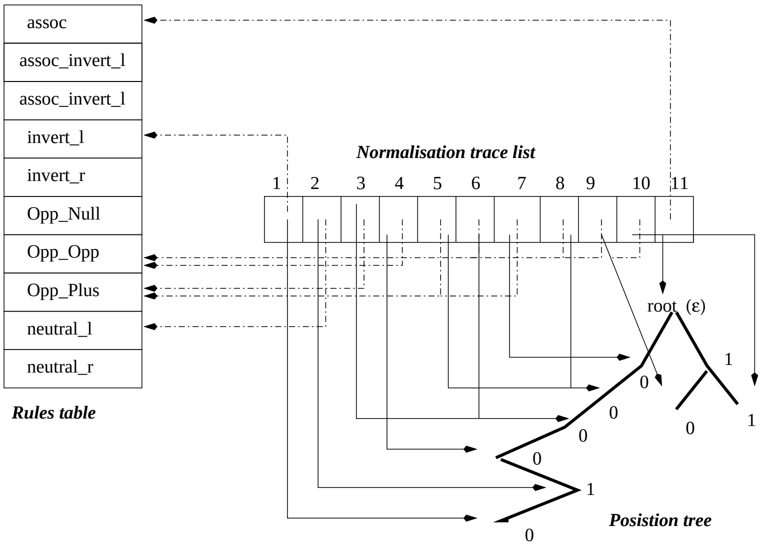

Fig. 4. An example of trace structure

4.1 Status of Equality

In the Calculus of Inductive Constructions, equality is defined as an inductive predicate. Let A be a type and **Prop** the kind of CIC-terms having a logical meaning, the equality of Coq ($=_{\mathsf{A}}$) is defined by the following rules:

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

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

The elimination rule of $=_{\mathsf{A}}$ is used to replace equals. This means each rewrite step in a proof (*i.e.* elimination of an equality) stores its logical justification: context of the rewrite (Φ), substitution (x, y) and equality ($x =_{\mathsf{A}} y$). These justifications can be done once for all using reflection.

4.2 First-Order Terms

We call *reflection* the ability for a formal language L_O (the object-level language) to be its own meta language. Reflection for L_O is defined by a triple $(L_M, \ulcorner\cdot\urcorner, \llcorner\cdot\lrcorner)$ where: $L_M \subset L_O$ is a set of Gödel codes; $\ulcorner\cdot\urcorner$ is an injective function ranging over L_O from L_M (encoding) and $\llcorner\cdot\lrcorner$ is the reverse function (decoding). The functions $\ulcorner\cdot\urcorner$ and $\llcorner\cdot\lrcorner$ are called, respectively: *reification* and *reflection*.

First-order, many-sorted terms are encoded in CIC using two mutually inductive datatypes: $\mathsf{fo.term}$ (terms) and $\mathsf{fo.term}^*$ (lists of terms). These types use datadependencies to restrict formation to well-sorted terms.

$$\frac{s\!:\!S}{(\mathsf{fo.term}\ s) : \mathbf{Set}} \qquad \frac{\bar{s}\!:\!S^*}{(\mathsf{fo.term}^*\ \bar{s}) : \mathbf{Set}}$$

$$\frac{x\!:\!X}{(\mathsf{fo.var}\ x) : (\mathsf{fo.term}\ \langle x\rangle)} \qquad \frac{}{\mathsf{fo.nil} : (\mathsf{fo.term}^*\ [\,])}$$

$$\frac{f\!:\!F \quad \bar{t} : (\mathsf{fo.term}^*\ \langle f\rangle^1)}{(\mathsf{fo.app}\ f\ \bar{t}) : (\mathsf{fo.term}\ \langle f\rangle^2)} \qquad \frac{s\!:\!S \quad \bar{s}\!:\!S^* \quad t\!:\!(\mathsf{fo.term}\ s) \quad \bar{t}\!:\!(\mathsf{fo.term}^*\ \bar{s})}{(\mathsf{fo.cons}\ s\ \bar{s}\ t\ \bar{t}) : (\mathsf{fo.term}^*\ s :: \bar{s})}$$

Here, **Set** is the kind of CIC-terms having a computational meaning; X is the type of first-order variable symbols; S is the type of sort symbols; $\langle x\rangle$ is the sort of x; $\langle f\rangle^1$ is the sorts (in a list) of the arguments of f and $\langle f\rangle^2$ is the sort of the result of f.

4.3 Reification & Reflection

Reification is performed by a meta-level function of Coq (*i.e.* a ML tactic). For the sake of simplicity the reification procedure is described for the calculus of construction (CC). CC-terms are defined by the following syntax:

$$\mathsf{cc.term} := (\mathsf{cc.var}\ x) | (\mathsf{cc.kind}\ k) | (\mathsf{cc.app}\ c\ \bar{c}) | (\mathsf{cc.prod}\ \bar{b}\ c) | (\mathsf{cc.lam}\ \bar{b}\ c)$$

where Y is the type of CC-variable symbols; K is the type of kind symbols; $x\!:\!X$; $c\!:\!\mathsf{cc.term}$, $\bar{c}\!:\!\mathsf{cc.term}^*$ and $\bar{b}\!:\!(Y \times \mathsf{cc.term})^*$.

Figure 5. describes the reification procedure. Computation of $(\mathsf{reify}\ \nu_0\ c)$ returns a pair (ν_1, t) where $t = \ulcorner c \urcorner$ and ν_1 is a CC-mapping of the variable symbols of t. CC-mapping of variables are functions of type $X \to \mathsf{cc.term}$. The mapping ν_1 is an extension of ν_0. We denote by $\llcorner F \lrcorner$ the set of CC-terms which are seen as reflection

$$\begin{array}{rcl}
\mathsf{reify} & : & \mathsf{cc.term} \to (X \to \mathsf{cc.term}) \times \mathsf{fo.term} \\
(\mathsf{reify}\ \nu\ (\mathsf{cc.app}\ f\ \bar{c})) & \triangleright & \mathbf{let}\ (\nu_{\bar{c}}, \bar{t}) = (\mathsf{reify}^*\ \nu\ \bar{c})\ \mathbf{in}\ (\nu_{\bar{c}}, (\mathsf{fo.app}\ \ulcorner f \urcorner\ \bar{t})) \\
(\mathsf{reify}\ \nu\ c) & \triangleright & \mathbf{if}\ c \in (\mathsf{range}\ \nu)\ \mathbf{then}\ (\nu, (\mathsf{fo.var}\ (\nu^{-1}\ c))) \\
 & & \mathbf{else\ let}\ x = (\mathsf{fresh}\ X\ \nu)\ \mathbf{in}\ (\nu \cup (x, c), (\mathsf{fo.var}\ x)) \\
\mathsf{reify}^* & : & \mathsf{cc.term}^* \to (X \to \mathsf{cc.term}) \times \mathsf{fo.term}^* \\
(\mathsf{reify}^*\ \nu\ [\,]) & \triangleright & (\nu, \mathsf{fo.nil}) \\
(\mathsf{reify}^*\ \nu\ c :: \bar{c}) & \triangleright & \mathbf{let}\ (\nu_c, t) = (\mathsf{reify}\ \nu\ c)\ \mathbf{in} \\
 & & \mathbf{let}\ (\nu_{\bar{c}}, \bar{t}) = (\mathsf{reify}^*\ \nu_c\ \bar{c})\ \mathbf{in} \\
 & & (\nu_{\bar{c}}, (\mathsf{fo.cons}\ \langle t\rangle\ \langle \bar{t}\rangle^*\ t\ \bar{t}))
\end{array}$$

Fig. 5. Reification procedure

(interpretation) of first-order function symbols, in Fig. 5. f is one of those and $\ulcorner f \urcorner$ is its Gödel code. The reflection procedure for fo.term objects

$$\mathsf{reflect} : (X \to \mathsf{cc.term}) \to \mathsf{fo.term} \to \mathsf{cc.term}$$

is only semantic interpretation (also denoted by $[\![\cdot]\!]$) of first-order terms into CIC.

The implemented tactic does not only perform the reification process as described above, it also disseminates it inside a CIC-term, as illustrated by figure 6. Let c

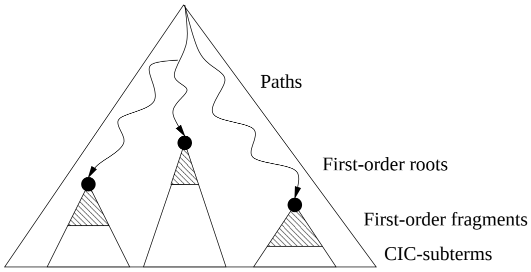

Fig. 6. Reification inside a CIC-term.

be a CIC-term, the set of first-order roots in c, denoted by R_c, is the set of paths leading to elements of $\llcorner F \lrcorner$ which are minimal for path ordering. For all $p \in R_c$ we replace $c|_p$ with $[\![\ulcorner c|_p \urcorner]\!]_\nu$, where ν is a common variable-mapping for all the reified sub-terms.

An equality $\ell = r$ is reified into a *rewriting function* γ of type

$$\mathsf{rewrite} := \forall s\!:\!S.(\mathsf{fo.term}\ s) \to (\mathsf{fo.term}\ s).$$

which computes the head reduction $(\gamma\ \ulcorner \ell \urcorner) \triangleright \ulcorner r \urcorner$. Since Coq functions must be total, f is the identity function on failure cases. When ℓ does not provide a linear pattern we use a conditional term rewriting encoding. For example $x - x \to 0$ is transformed into $x - y \to 0$ **if** $x = y$. These conditional rules can be computed since equality is decidable on fo.term (inductive datatype). A *context rewriting* function

$$\mathsf{inctx} : \mathsf{nat}^* \to \mathsf{rewrite} \to \mathsf{rewrite}$$

is also defined, it applies a rewrite at a given position in a term. Positions are defined by Dewey words (lists of natural numbers).

A rewriting function γ is reflected using a *stability* lemma

$$\forall x\!:\!(\mathsf{fo.term}\ s).[\![x]\!] =_{[\![s]\!]} [\![(\gamma\ x)]\!]$$

called π, where $[\![s]\!]$ is the interpretation (*i.e.* reflection) of the sort symbol s and $=_{[\![s]\!]}$ is Coq's equality on this domain. Stability is also defined for context rewriting.

4.4 Rewriting Traces

Let Γ be a set of Coq equalities one wants to use as a first-order term rewriting system. Let ℓ and r be terms such as $\ell \to_\Gamma^* r$. A *trace* for $\ell \to_\Gamma^* r$ is a list of triples

$$t = [(\gamma_1, \pi_1, w_1); \ldots; (\gamma_n, \pi_n, w_n)]$$

where γ_i is a rewriting function, π_i is a stability proof for γ_i and w_i is a position. Since t verifies

$$((\mathsf{inctx}\ \gamma_n\ w_n) \circ \cdots \circ (\mathsf{inctx}\ \gamma_1\ w_1))\ \ulcorner \ell \urcorner = \ulcorner r \urcorner$$

it can be used to rewrite ℓ in r using the function:

$$\mathsf{replay} : \mathsf{trace} \to \mathsf{rewrite},$$

where trace denotes the type of traces. Stability is extended to trace replaying using the stability proofs π_i. Figure 7. describes the construction of a proof for $\Gamma \vdash \ell = r$

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash r = r}\ \text{Reflexivity}}{\Gamma \vdash [\![\ulcorner r \urcorner]\!]_\nu = r}\ \text{Compute } [\![\cdot]\!]_\nu}{\Gamma \vdash [\![(\mathsf{replay}\ t\ \ulcorner \ell \urcorner)]\!]_\nu = r}\ \text{Compute } \mathsf{replay}}{\Gamma \vdash [\![\ulcorner \ell \urcorner]\!]_\nu = r}\ \text{Elim } (\mathsf{replay}\ t)\text{'s stability}}{\Gamma \vdash \ell = r}\ \text{Reify}$$

Fig. 7. Equational Reasoning using Reflection

in this setting.

5 Implementation & Interface

In this section we describe, the implementation of the Coq/ELAN interface. A decision procedure using this interface is implemented as a Coq tactic. Figure 8. shows an overview of such a tactic.

Communication between Coq and ELAN uses the TCP/IP sockets. This communication solution was preferred because we wanted to keep two systems cleanly separated. Both systems are rather complex and they should not be tangled too tight. This solution has a small impact on both systems. Hopefully the Coq/ELAN interface will be more maintainable and extensible this way. We didn't wanted to create a *two-headed monster*.

In our interface, ELAN takes the role of a server and Coq proof sessions work as its client. The ELAN server listens on a pre-defined TCP port. All Coq clients will open the different connections to this unique port. Each Coq client is served by a separated child process. This process generates and launches a rewriting engine for a given theory received from Coq. Then, the child process loops and waits for Coq terms accompanied by a reduction strategy. It passes these terms to the rewriting engine and retrieves the computational traces in the successful cases. These traces are then returned to Coq client for replaying. In other cases, when the term is not syntactically well-formed or when the normalisation process fails, Coq client will be informed by an error signal.

The generation of rewriting engine may take a little time due to two compilation process in ELAN and C. Therefore, a list of available theories whose the rewriting engine already exits in ELAN server is consulted and updated respectively before and after each generation process. For an existing theory, a simple call of its rewriting engine is sufficient.

This interface mechanism is completely transparent for Coq users. In a working session, the users first define their working theory by a set of axioms and then,

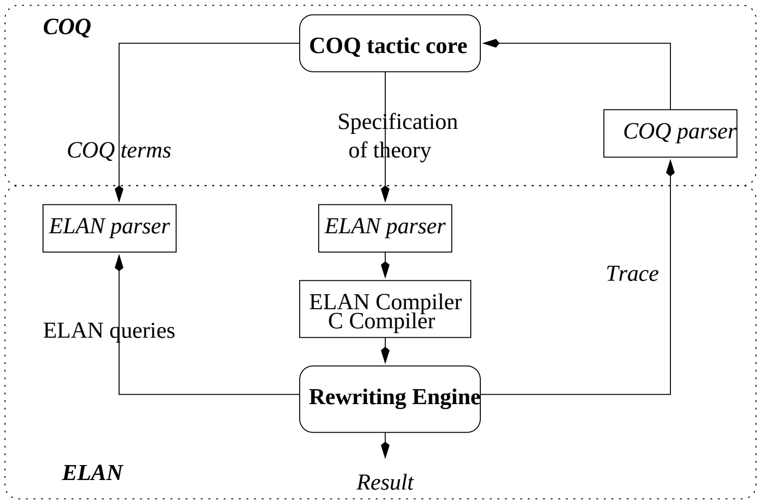

Fig. 8. Integration overview

apply the integrated tactic to check equality between two arbitrary terms in all their proofs. The client interface on Coq side first sends the theory to ELAN server to launch the respective rewriting engine. Then, it sends the terms and replays the returned traces to have an equational proof. At the end of session, Coq client sends a QUIT signal to stop the rewriting engine. Figure 9 shows some possible exchanged signals in a working session of our interface.

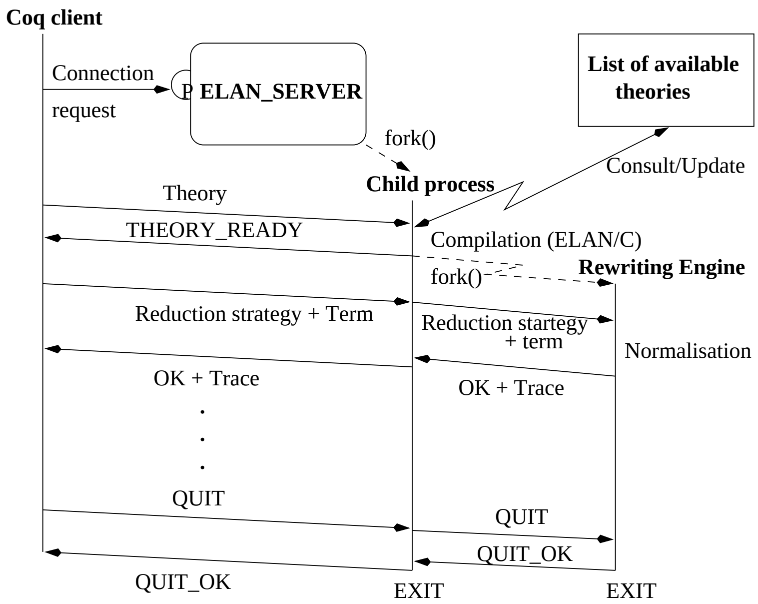

Fig. 9. An interface working session

6 Conclusion and future works

The Coq/ELAN interface only works with first-order, many-sorted term rewriting systems. Extensions towards derived formalisms are planned, *e.g.* conditional rewriting. These extensions will require extension of the trace structure as well as new Coq formalisations.

Some decision procedures involve search or tree exploration. Implemented in ELAN with the strategy language the search process can be done efficiently. One successful path can then be returned to Coq that has just to check it.

ELAN is also a powerful non-deterministic calculus engine. Presently, we only work with canonical rewriting systems but extensions towards non-confluent or non-terminating systems can be imagined. Nevertheless, tracing non-deterministic computation might raise space management problems. Adequate modifications of the trace structure might be sufficient to handle these systems.

People in the Coq community are working on extensions of the calculus of the constructions towards reasoning modulo sets of equational axioms (*e.g.* associativity or commutativity). These extensions might permit removing from the trace some information. It might also be useful to take advantage of the builtin AC-rewriting mechanism of ELAN.

Acknowledgements

We sincerely thank Pierre Crégut and Helène Kirchner for their comments in the draft version of this paper. We also thank Claude Kirchner, Christine Paulin-Mohring, Benjamin Werner, Pierre-Étienne Moreau and Christophe Ringeissen for many fruitful discussions.

References

Ahrendt, Wolfgang, Beckert, Bernhard, Hähnle, Reiner, Menzel, Wolfram, Reil, Wolfgang, Schellhorn, Gerhard, & Schmitt, Peter. (1998). Integrating automated and interactive theorem proving. Bibel, W., & Schmitt, P. (eds), *Automated - a basis for applications*, vol. 1. Kluwer Academic Publishers.

Barthe, Gilles, Ruys, Mark, & Barendregt, Henk. 1996 (June). A two-level approach towards lean proof-checking. *Pages 16–40 of:* Berardi, Stefano, & Coppo, Mario (eds), *Types for proofs and programs, international workshop, proceedings, Types'95.* Lecture Notes in Computer Science, vol. 1158.

Borovanský, Peter, Kirchner, Claude, Kirchner, Hélène, Moreau, Pierre-Etienne, & Ringeissen, Christophe. (1998). An overview of ELAN. Kirchner, Claude, & Kirchner, Hélène (eds), *Proceedings of the second international workshop on rewriting logic and applications*, vol. 15, `http://www.elsevier.nl/locate/entcs/volume15.html`. Pont-à-Mousson (France): Electronic Notes in Theoretical Computer Science. Report LORIA 98-R-316.

Boulton, Richard, Slind, Konrad, Bundy, Alan, & Gordon, Mike. (1998). An interface between clam and hol. *Pages 87–104 of:* Grundy, J., & Newey, M. (eds), *Proceeding of the 11th international conference on theorem proving in higher order logics.* Lecture Notes in Computer Science, no. 1479. Springer-Verlag.

Boutin, Samuel. 1997 (Sept.). Using reflection to build efficient and certified decision procedures. *Pages 515–526 of: Theoretical aspects of computer software, 3rd international symposium, proceedings, TACS'97*. Lecture Notes in Computer Science, vol. 1281.

Bush, Holger. (1994). First-order automation for higher-order-logic theorem proving. *Pages 97–112 of:* Melham, T., & Camilleri, J. (eds), *Higher order logic theorem proving and its application: 7th international workshop*. Lecture Notes in Computer Science, no. 859. Springer-Verlag.

Calmet, Jacques, & Homann, Karsten. (1996). Classification of communication and cooperation mechanisms for logical and symbolic computation systems. *Pages 221–234 of:* Baader, F., & Schulz, K. U. (eds), *Frontiers of combining systems, 1st international workshop, proceedings, FroCos'96*. Applied Logic. Kluwer Academic Publishers.

Cirstea, Horatiu, & Kirchner, Claude. (1999). *An introduction to the rewriting calculus*. Rapport de recherche. `http://www.loria.fr/~cirstea/RoReport.ps.gz`.

Gordon, M. J. C., & Melham, T. F. (1993). *Introduction to HOL*. Cambridge University Press.

Harrison, John, & Théry, Laurent. (1998). A sceptic's approach to combining HOL and Maple. *Pages 279–294 of: Journal of automated reasoning*, vol. 21.

Howe, Douglas James. 1988 (Mar.). *Automatic reasoning in an implementation of constructive type theory*. Ph.D. thesis, Cornell University.

Hurd, Joe. (1999). Integrating Gandalf and HOL. *Pages 311–321 of:* Bertot, Y, Dowek, G, Hirschowitz, A, Paulin, C, & Thery, L (eds), *Theorem proving in higher order logics TPHOL'99*. Lecture Notes in Computer Science, no. 1690. Springer-Verlag.

Joyce, Jeffrey, & Seger, Carl. (1994). The hol-voss system: Model-checking inside a general-purpose theorem prover. *Pages 185–198 of:* Joyce, J.J, & Seger, C (eds), *Higher order logic theorem proving and its applications: 6th international workshop, hug'93*. Lecture Notes in Computer Science, no. 780. Springer-Verlag.

Klop, J. W. (1990). Term Rewriting Systems. *Chap. 6 of:* Abramsky, S., Gabbay, D., & Maibaum, T. (eds), *Handbook of logic in computer science*, vol. 1. Oxford University Press.

Knuth, Donald E., & Bendix, P. B. (1970). Simple word problems in universal algebras. *Pages 263–297 of:* Leech, J. (ed), *Computational problems in abstract algebra*. Oxford: Pergamon Press.

Paulson, Laurence C. (1983). A Higher-Order Implementation of Rewriting. *Science of Computer Programming*, **3**, 119–149.

Paulson, Laurence C., & Nipkow, Tobias. (1999). *Simplification*. Computer Laboratory, University of Cambridge. Chap. 10.

Rushby, John. (1999). Integrated formal verification: Using model checking with automated abstraction, invariant generation, and theorem proving. *Pages 311–321 of:* Dams, D, Gerth, R, Leue, S, & Massinek, M (eds), *Theoretical and practical aspects of spin model checking*. Lecture Notes in Computer Science, no. 1680. Springer-Verlag.