Lot 5
Technologie de vérification
Coopération Coq/ELAN pour la recherche et la vérification de preuve de théorèmes équationels ou inductifs

Description :	Nous présentons nos travaux en cours sur la combinaison des méthodes de recherche de preuve automatique par réécriture et des assistants à la preuve. Nous élaborons les concepts nécessaires et montrons comment les mettre en œuvre tout en respectant une contrainte forte : assurer une coopération sûre entre les deux approches. En pratique, nous instancions l'étude théorique sur l'assistant à la preuve Coq et, en ce qui concerne la réécriture, sur le système ELAN, en nous concentrant tout d'abord sur les preuves équationnelles, puis sur les preuves inductives. Différents concepts, notamment le calcul de réécriture et la déduction modulo, sont utilisées pour définir et relier la recherche, la représentation et la vérification de preuves.
Auteur(s) :	Eric Deplagne, Claude Kirchner, Hélène Kirchner, Anamaria Martins-Moreira, Quang Huy Nguyen
Référence :	Averroes / Lot 5 / Fourniture 3.1 / V1.0
Date :	15 septembre 2003
Statut :	validé
Version :	1.0

Réseau National des Technologies Logicielles
Projet subventionné par le Ministère de la Recherche et des Nouvelles Technologies

CRIL Technology, France Télécom R&D, INRIA-Futurs, LaBRI (Univ. de Bordeaux – CNRS), LIX (École Polytechnique, CNRS) LORIA, LRI (Univ. de Paris Sud – CNRS), LSV (ENS de Cachan – CNRS)

Historique

15 septembre 2003	V 0.9	création du document
20 novembre 2003	V 1.0	mise au format

Table des matières

1 Introduction **3**

2 Making Coq and ELAN cooperating **4**

3 Construction of proof terms **5**

4 Proof search and proof check for equational proofs **6**
4.1 Tracing term normalisation by AC rewriting in ELAN . . . 6
4.2 Proof term translation . . . 7
4.3 Extension for proofs by structural induction . . . 8

5 Deduction modulo and the Noetherian induction principle **10**
5.1 Deduction modulo and $HOL_{\lambda\sigma}$ . . . 10
5.2 Deduction modulo for inductive proofs . . . 12
5.3 Proof search by narrowing . . . 13
5.4 Complete narrowing for proof search . . . 14

6 Translating induction proofs into Coq **15**
6.1 Structural induction . . . 15
6.2 Noetherian induction . . . 15

7 Coq-ELAN for Induction - future work **15**
7.1 Semi-automatic version . . . 16
7.2 Automatic version . . . 16

8 Conclusion **16**

1 Introduction

Although formalized since the beginning of mechanized deduction, the three concepts of proof search, proof representation and proof check are always of fundamental interest, both from the theoretical point of view and for their applications, in particular concerning computer security and dependability. These three notions have drawn independently much attention, but are closely related and this work addresses them as a whole, focusing first on equational and then on inductive proofs.

The conceptual bridges joining proof search, representation and check are the deduction modulo and the rewriting calculus. *Deduction Modulo* [DHK03] is a presentation of first-order logic allowing to identify the notions of computation and deduction and their interactions. This fundamental difference between computation and deduction has been identified since at least one century by Henri Poincaré and it plays a fundamental role in today proof theory, because the status of what we search for and what needs to be computed should be identified and treated appropriately, in order to get proofs where only useful (and often difficult) parts are described.

Indeed, we are today able to compute billion of numerical operations per second and symbolic systems like ELAN [KM01] are able to execute several tens of millions of computational rewrites per second. Building these computations as explicit parts of proofs is hopeless and, most important, uninteresting. Deduction modulo is a way to keep such computations into account.

The *rewriting calculus* [CK01], has been introduced to combine the capabilities of standard rewriting and of the lambda calculus. It relies on the matching paradigm as a fundamental way to capture the structural form of the considered objects. We are using it in this work as a powerful means to represent proofs in a compact way.

The necessity to check a proof as such, and not the proof search, has also been identified since a long time, and the now well established use of proof-assistants like Coq makes this routine. It allows us to (mostly) forget about the *believing way*, in which the proof assistant just accepts new statements produced by the proof search side and to switch to the skeptical or the autarkic ways (using the terminology of [BB02]). In the *skeptical way*, the proof search engine must provide with each of its results a proof object checkable by the deduction system. In the *autarkic way* the proof assistant learns to itself proof search, by incorporating for instance some verified term rewriting engine or model-checker in the system itself, using a reflection technique to be more efficient [Har95, Bou97].

This paper presents an on-going research project on theoretical and practical issues of combining rewriting based automated theorem proving and user-guided proof development, with the strong constraint of safe cooperation of both. In practice, we instantiate the theoretical study on the Coq proof assistant and the ELAN rewriting based system.

A first approach of cooperation between Coq and ELAN is described, where Coq delegates equational proofs by rewriting to ELAN that builds a proof term, while doing the proof. This proof term is then returned to Coq and checked. In the context of proof by structural induction performed by Coq, this technique is currently experimented for proving the base case and the induction case by rewriting. However proofs by Noetherian induction performed by rewriting are much more powerful than structural induction. A second approach, which is yet on-going work, is then presented. Based on the description at the proof theoretical level of proof by Noetherian induction provided by the deduction modulo framework, we show how to use narrowing in order to perform proof search for an inductive proof by rewriting, and give hints to build a proof term that can be checked by Coq. In order to check the proof, additional proof obligations to justify the Noetherian induction principle in Coq are also needed.

The structure of the paper is as follows: Section 2 gives the general context of the cooperation between Coq and ELAN as an instance of the skeptical approach. Section 3 presents the notion of proof terms, their representation in $\rho\sigma$-syntax and their association to rewriting derivations. Section 4 shows how Coq and ELAN actually cooperate in proof construction and proof checking, for rewriting proofs and for proofs by structural induction. Section 5 relates the deduction modulo framework and Noetherian induction, gives a proof theoretic presentation of induction by rewriting, and shows how to perform the induction step using an adequate notion of narrowing. Section 6

briefly sets what are the needed steps for making ELAN able to perform delegated proofs by induction and for making Coq able to check these proofs. Then related works and further goals are mentioned in the concluding Section 8.

2 Making Coq and ELAN cooperating

In [NKK02, Ngu02a], we consider an instance of this general problem that consists in using rewriting techniques to tackle some of the computations needed by a proof assistant. The approach was a contribution to the skeptic way to integrate computation and deduction. Its concrete realization was to make cooperating the Coq proof assistant, based on the calculus of constructions, with the ELAN deduction and computation system, based on the rewriting calculus. From a practical point of view, this amounts on one hand to providing for Coq a class of decision procedures using term rewriting techniques, and on the other hand to using Coq as a proof checker for ELAN.

Coq is a proof assistant based on the Calculus of Inductive Constructions, the Calculus of Constructions [CH88] with inductive data types [PM93]. Proofs in Coq are constructive and by the Curry-Howard isomorphism, logical propositions are interpreted as types. A proposition is provable if and only if, when interpreted as a type, it is inhabited by a term which is a proof term of that proposition. The proof terms generated in deduction steps are type checked by Coq kernel. This approach has strong advantages: correctness is ensured by the reliability of a tiny kernel, and a certified (functional) program can be extracted from the proof of its specification. However, this mechanism requires to keep all information concerning each deduction step in the proof term that is often huge.

ELAN provides an environment for specifying and prototyping deduction systems in a language based on rewrite rules controlled by strategies [BKKM02]. It offers a natural and simple logical framework for the combination of the computation and deduction paradigms. It supports the design of theorem provers, logic programming languages, constraint solvers and decision procedures and it offers a modular framework for studying their combination. ELAN's evaluation mechanism is based on rewriting. In ELAN a rewrite rule may be labeled, may have boolean conditions, and matching conditions. The evaluation mechanism also involves backtracking since in ELAN, an evaluation step may have zero, one or several results. One of the original aspects of the system is to provide a strategy language allowing the programmer to specify the control used for the rule applications. This is in contrast to many existing rewriting-based languages where the term reduction strategy is hard-wired and not accessible to the designer of an application. The strategy language offers primitives for sequential composition, iteration, deterministic and non-deterministic choices of elementary strategies that are labeled rules. From these primitives, more complex strategies can be expressed. ELAN's formal foundation is based on rewriting calculus [CK01], also called ρ-calculus, a common generalization of lambda-calculus and of term rewriting.

The motivation of combining proof assistants based on constructive type theory and automated provers based on rewriting is related to the definition of equality in proof assistants. Definitional equality is not easy to extend since it is in general difficult to add new rules in the kernel of the proof assistant, while keeping the strong requirements of subject reduction and decidability of type checking. So often, logical equality is defined as a theory and equality proof terms are built and later checked by the kernel. However this second option also raises some problems: efficiency is low and the size of generated proof terms could be prohibitive. Moreover, this technique does not easily generalize to associative commutative (AC for short) theories, frequent in practice, but where the exponential complexity of AC-pattern matching is an additional obstacle to efficiency.

The approach followed for equational proofs relies on a normalization tactic in associative and commutative theories written in ELAN. It generates a proof term in the rewriting calculus, which is then translated into a proof term written in the Calculus of Constructions syntax that can finally be checked by Coq to get the proof of the normalization process. The advantages of this approach are to take benefit from the efficient (conditional AC) rewriting performed by the ELAN compiler, and to ease the size reducing transformations of the proof terms before sending them to Coq.

Actually this work goes beyond the specific use of Coq and ELAN. It raises the general problem

of incorporating equational reasoning, and more generally decision procedures, in proof assistants based on type theory, in a reliable and efficient way. Reliability is handled here through the concept of proof term, that contains all information about the proof and is exchanged between the two systems. Built by ELAN during the rewriting proof construction, it is then checked by Coq or by the proof assistant.

3 Construction of proof terms

In a proof term, the information needed for each proof step has to be recorded in the most compact way. While for syntactic rewriting, the rewrite proof term can be reduced to a simple trace of rewriting derivation, which may be represented as a list of pairs $\langle rulename, position_of_redex\rangle$, the situation is different for AC rewriting: in this case, the proof term needs to include the used substitutions, since AC-pattern matching is not unitary, and moreover the subterm position is no more appropriate to identify a redex. For example, given the rewrite system, $\mathcal{R} = \{\ (x + (-x)) + y \quad\rightarrow\quad y\ \}$ where the symbol $+$ is AC, the term $(a+b)+((-a)+(-b))$ is rewritten modulo AC by Peterson and Stickel's rewrite relation denoted $\rightarrow_{\mathcal{R},AC}$ [PS81, JK86]) at root position in two different terms:

$$\begin{array}{lll} (a+b)+((-a)+(-b)) & \rightarrow_{\mathcal{R},AC} & a+(-a) \\ & \rightarrow_{\mathcal{R},AC} & b+(-b) \end{array}$$

using respectively the first-order substitutions $\{x \mapsto b, y \mapsto a+(-a)\}$ and $\{x \mapsto a, y \mapsto b+(-b)\}$. The requirement of taking into account AC rewriting induces an additional complexity compared to pure rewriting.

The chosen proof term representation is based on the $\rho\sigma$-calculus [Cir00, CK01] that provides a syntax and a semantics to an appropriate notion of proof objects. The main idea of this calculus is to make all the basic ingredients of rewriting explicit objects, in particular the notions of rule *application* and *result*.

A rule application can be reduced to a singleton, but it may also fail and return the empty set, or it can be reduced to a set with more than one element. For example, if the symbol $+$ is assumed to be commutative then $x+y$ is equivalent modulo commutativity to $y+x$ and thus applying the rule $x+y \rightarrow x$ to the term $a+b$ results in $\{a, b\}$.

Moreover, the rewrite binary operator "$_ \rightarrow _$", being integrally part of the calculus syntax provides a powerful abstraction operator whose relationship with λ-abstraction [Chu40] gives a useful intuition: a λ-expression $\lambda x.t$ can be represented in the ρ-calculus as the rewrite rule $x \rightarrow t$. Indeed, the β-redex $(\lambda x.t\ u)$ is nothing else than $[x \rightarrow t](u)$ (i.e., the application of the rewrite rule $x \rightarrow t$ to the term u) which reduces to $\{\{x/u\}t\}$ (i.e., the application of the higher-order substitution $\{x/u\}$ to the term t). Of course we have to make clear what a substitution $\{x/u\}$ is and how it applies to a term. This is performed by a substitution mechanism that preserves the correct variable bindings via the appropriate α-conversion. For a general presentation of explicit substitution calculi, the reader is referred for example, to [ACCL90, CHL96].

Shortly speaking, in $\rho\sigma$-calculus, abstraction is handled via the arrow binary operator, matching is used as the parameter passing mechanism, substitution takes care of variable bindings and results sets are handled explicitly.

More formally, let $\Sigma = (\mathcal{F}, \mathcal{X})$ be a signature, with a set $\mathcal{F}$ of function symbols and a set $\mathcal{X}$ of variables. The $\rho\sigma$-expressions are divided into two sorts, one for terms and another for substitutions, which are defined by the following BNF notations ($x \in \mathcal{X}$ and $f \in \mathcal{F}$):

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

The set of terms contains the first-order terms in the signature $(\mathcal{T}(\mathcal{F}, \mathcal{X}))$ and several new constructs. The binary symbol $\rightarrow$ is used to represent ρ-abstractions. The ρ-application (of terms on terms) is represented by the binary operator $[_](_)$ where $_$ is the place holder. The application

of a substitution on a term is denoted by the binary operator $_\langle_\rangle$. The set construct is used to represent the result of a rewrite step which is in general non-deterministic.

The substitution syntax is composed of the identity substitution ($\mathbb{ID} = \{x/x, y/y, \ldots\}$), the *shift* ($\uparrow$), the composition operator ($\circ$), the operator *cons* of a term onto a substitution (.) and the *lift* ($\Uparrow$). The operators *shift* and *lift* update bound variable indices.

In order to better understand this notion of proof term, let us give an example of a rewrite derivation and its associated proof term in $\rho\sigma$-calculus.

Example 0.1 *Consider the rewrite system*

$$\mathcal{R} = \left\{ \begin{array}{lrcl} [r1] & x + (-x) & \rightarrow & 0 \\ [r2] & x + 0 & \rightarrow & x \end{array} \right.$$

where $+$ is an AC symbol. The term $(a+b)+(-a)$ can be normalized by the following derivation:

$$(a+b)+(-a) \quad \xrightarrow{r1}_{\mathcal{R},AC} \quad 0+b \quad \xrightarrow{r2}_{\mathcal{R},AC} \quad b$$

The associated $\rho\sigma$-proof term of this derivation is:

$$\pi = ((x+(-x))\langle x/a\rangle \rightarrow 0\langle x/a\rangle) + b \;;\; (x+0)\langle x/b\rangle \rightarrow x\langle x/b\rangle$$

The proof term for a rewrite step at root position $l\sigma \longrightarrow r\sigma$ is of the form $l\langle\sigma\rangle \rightarrow r\langle\sigma\rangle$. If the rewrite step is performed inside a term, then its context needs also to be included in the proof term. The proof term for a rewriting derivation is obtained by concatenating the proof terms for its steps.

4 Proof search and proof check for equational proofs

We are now ready to describe more precisely how ELAN and Coq collaborate in proof construction and proof check activities. For that, the ELAN compiler, i.e. the rewrite engine, has been extended by a proof term producer that builds the *rewriting proof term*, and by a proof term translator that transforms this formal trace of ELAN into the corresponding Coq proof term for checking. In this cooperation scheme, ELAN can be seen as a computing server and Coq proof sessions as its clients. At the moment, both syntactic and AC rewriting are supported. A version for conditional rewriting is being experimented where ELAN generates proof obligations for conditional rewrite steps, whose proofs are left for now to the Coq side.

4.1 Tracing term normalisation by AC rewriting in ELAN

As already mentioned in the previous section, building proof terms for AC rewriting raises a number of difficulties, besides the fact that multiple results and corresponding substitutions have to be taken into account. Let us mention below some other points related to specificities of the ELAN compiler.

The trace of an AC rewrite step contains the applied rule, the used substitution and the context. By AC rewriting, terms are normalised in ELAN using the leftmost-innermost strategy. A term is flattened in its canonical form before being reduced: the subterms of an AC symbol are sorted while identical subterms are put together and their count represented by a multiplicity exponent. For example, the term $f(a, f(a,b))$ is flattened in $f(a^2, b)$ if f is an AC symbol. In order to speed up AC-pattern matching, the ELAN compiler automatically transforms complex patterns into simpler ones.

For example, if f, g are AC symbols, the rule $h(f(x_1,x_2), g(x_3,x_4)) \rightarrow r$ is transformed into $h(X, g(x_3,x_4)) \rightarrow r$ where $f(x_1,x_2) = X$, where X is a fresh variable. The pattern matching on $h(f(x_1,x_2), g(x_3,x_4))$ is hence decomposed into two simpler steps: the first one on $h(X, g(x_3,x_4))$ and the second one on $f(x_1,x_2)$. Both patterns are in the restricted class for which several optimizations of AC-pattern matching are valid. However, in order to get the used substitution for a

transformed rule, we need also to trace the pattern matching of the *local evaluations* by where. For example, for the rule above, the first pattern matching returns the instantiation for x_3 and x_4 while the second one yields the instantiation for x_1 and x_2.

On the other hand, for efficiency reason, ELAN simulates the rewriting relation with AC-equivalence class ($\longrightarrow_{\mathcal{R}/AC}$) by the rewriting modulo AC relation ($\longrightarrow_{\mathcal{R},AC}$) [PS81, JK86]. The completeness is ensured by adding the *extension rules*, if necessary, in the original rewrite system. That is, for each rule $l \to r$ such that the head symbol of l is an AC symbol f, an extension rule $f(l, X) \to f(r, X)$ is added, where X is a fresh variable called the *extension variable*. Because the extension rule notion is not known by Coq, we need to reconstruct the corresponding trace by the original rule. If $C[\Box]$ is the context and σ is the used substitution in a rewrite step by the extension rule, i.e. $C[f(l, X)\sigma] \longrightarrow C[f(r, X)\sigma]$, the context of the corresponding rewrite step by the original rule $l \to r$ is $C[f([\Box], \sigma_X)]$. The used substitution σ' is obtained from σ by eliminating the image of the extension variable X.

4.2 Proof term translation

The translation of ELAN rewriting proof terms into the calculus of constructions syntax is described in [NKK02]. The trace of ELAN rewriting is first formalized in $\rho\sigma$-syntax. This trace is then normalised into its compact canonical form before being translated into an immediate format, called Π-syntax, and finally, into Coq-syntax. The Π-syntax gives genericity to the translation since one can parameterise it by proof term syntaxes of proof checkers: actually, constructions of Π-syntax actually mimic the basic properties of equality: reflexivity, symmetry, transitivity, substitutivity and congruence. Two operators $\rho\sigma 2\Pi$ and $\Pi 2Coq$ are used to translate proof terms from $\rho\sigma$-syntax to Π-syntax and from Π-syntax to Coq proof term syntax. As discussed in Section 3, translating proof terms of AC rewriting has another technical problem: the equalities modulo AC in ELAN are implicit while they need an explicit proof in Coq. In [Ngu02b], an efficient method for proof search and proof check of equalities modulo AC is described. This method has been used for checking rewriting proof terms by Coq.

The soundness of the proof term translation is stated as follows.

Theorem 0.1 *[NKK02] If π is a $\rho\sigma$ proof term for the rewriting derivation $t \longrightarrow^*_{\mathcal{R}} s$ in* ELAN*, then $\Pi 2Coq(\rho\sigma 2\Pi(\pi))$ is a proof of the equality $t = s$ in* Coq*.*

The general scheme of cooperation between Coq and ELAN is described in Figure 1.

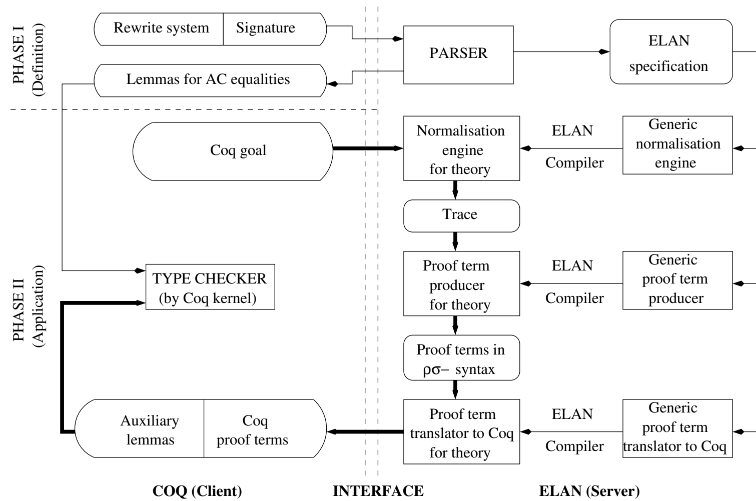

FIG. 1 – *Integration of automatic* ELAN *rewriting into* Coq *proofs*

In the first phase, the Coq user defines the client theory before calling the server that generates from this theory an ELAN specification and a set of lemmas needed for checking equalities modulo AC. The generic normalisation engine, proof term producer and proof term translator are three ELAN modules which generate from a specification its corresponding executable files using ELAN compiler. In the second phase, automatic rewriting can be used in any Coq proof. The bold arrows represent the data flow exchanged between the two systems during this phase. The trace of ELAN rewriting is processed in several steps before being able to be checked by the Coq kernel.

Example 0.2 *Let us come back to Example 0.1. Traditionally, the proof of the derivation* $(a+b)+(-a)\longrightarrow^*_{\mathcal{R},AC} b$ *in* Coq *must be manually performed using the associative and commutative properties of* $+$*, that are respectively called* +_assoc *and* +_commu*, as well as the two rules* r1 *and* r2*.* Rewrite *is a* Coq *tactic that replace equal by equal with a given axiom. The* Coq *script written by the user is as follows:*

```
Rewrite +_assoc;    (* rewrite (a+b)+(-a) to a+(b+(-a)) *)
Rewrite +_commu;    (* rewrite a+(b+(-a)) to a+((-a)+b) *)
Rewrite <- +_assoc; (* rewrite a+((-a)+b) to (a+(-a))+b by reverse
                       associativity *)
Rewrite r1;         (* rewrite (a+(-a))+b to 0+b *)
Rewrite +_commu;    (* rewrite 0+b to b+0 *)
Rewrite r2;         (* rewrite b+0 to b *)
```

This manual proof can be greatly improved now by automatically calling the ELAN *based rewriting tactic which performs AC rewriting and proofs of equalities modulo AC as follows:*

$$(a+b)+(-a) \quad \xrightarrow{r1}_{\mathcal{R},AC} \quad 0+b \quad \xrightarrow{r2}_{\mathcal{R},AC} \quad b$$

The tactic first provides to Coq *the proof of the equality modulo AC* $(a+b)+(-a) =_{AC} (a+(-a))+b$ *and then, translates the* $\rho\sigma-$*proof term* π *to* Coq *proof term syntax for checking the derivation* $(a+(-a))+b\longrightarrow^*_{\mathcal{R},AC} b$.

4.3 Extension for proofs by structural induction

A similar approach is currently experimented on the proofs by structural induction performed by Coq. In Coq, a default induction scheme for each inductive data type, called *elimination principle*, is generated by Coq when the data type is defined. This scheme is defined from the constructors of the data type. For instance, for a proof of a property `P` on the type `nat`, with constructors `0` and `S`, the elimination principle is Peano's induction principle:

```
nat_ind : (P:(nat->Prop))(P 0)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```

The Coq user must provide at least the recursion variable, that should be of an inductive type, and the system then generates the subgoals to be proved. Usually these subgoals need to be simplified and at some points the subgoals are simply proved by rewriting. Such parts of the proof can be delegated to ELAN as previously, especially in the case of AC theories.

For that, the ELAN based rewriting tactic has been enriched by the capability to dynamically add new rules (and so, new sorts and symbols) to an existing rewrite theory for ELAN. In a practical proof development, the Coq user often wants to enrich the rewrite system by newly obtained theorems or lemmas. Incremental definition of current rules for the tactics allows him to add such new hypotheses into a pre-defined rewrite engine. The new engine can then be used for simplifying any term in the later proofs. In particular, a proof by induction requires the use of an induction hypothesis, not known before, and that needs to be added to the rewrite engine during the proof of the induction case. More generally, the local context of a proof includes hypotheses that can also be used for simplifying the current goal, which is often very useful. The main difference in comparison with global context is that when the proof is finished, the local context is closed and all of its hypotheses are no more valid.

In Coq, the induction principle is generated from the definition of inductive data type. Given this induction principle, one can generate in ELAN the proof term of a proof by induction if

rewriting can solve alone the base case and the induction case. This amounts to an automatic tactic, that we are experimenting, which allows the Coq user to delegate simple structural induction proofs from Coq to ELAN.

Example 0.3 *We prensent here an example to show how AC rewriting is used to partially automatize structural inductions in* Coq*. We consider an extension of Peano arithmetic by exponential operator. The main lemma consists in proving that* $\forall x, n_1, n_2 \in \mathbb{N}. x^{n_1} * x^{n_2} = x^{n_1+n_2}$.

```
(* + and * are declared as AC operators *)
Parameter AC nat_plus nat.
Parameter AC nat_mult nat.

Parameter nat_exp : nat -> nat -> nat.
(* Axioms for arithmetic *)
Parameter plus_zero: (x:nat) (nat_plus x O) = x.
Parameter plus_succ: (x,y:nat) (nat_plus x (S y)) = (S (nat_plus x y)).
Parameter mult_zero: (x:nat) (nat_mult x O) = O.
Parameter mult_succ: (x,y:nat) (nat_mult x (S y)) =
                                    (nat_plus x (nat_mult x y)).
Parameter exp_zero: (x:nat) (nat_exp x O) = (S O).
Parameter exp_succ: (x,y:nat) (nat_exp x (S y)) =  (nat_mult x (nat_exp x y)).

Elan Sort A_dom_L [nat].
Elan Symbol A_fun_L [nat_plus nat_mult nat_exp O S].
Elan Rule A_axm_L [plus_zero plus_succ mult_zero mult_succ exp_zero exp_succ].

(* Definition of rewrite system *)
Elan Theory expo A_dom_L A_fun_L.
Elan Rewriting expo A_axm_L.

(* Connect to ELAN server *)
Connect expo "localhost".

Lemma exp_mult: (x,n1,n2:nat) (nat_mult (nat_exp x  n1) (nat_exp x n2)) =
                                      (nat_exp x (nat_plus n1 n2)).
Proof.
  Induction n1.
  (* Base case *)
  Intros.
  ElanRewrite expo.

  (* Induction case *)
  Intros.

  (* Add new symbols and induction hypothesis *)
  Elan Pose Symbol A_fun_L_add [x n1 n2 n].
  Elan Pose Rule A_axm_L_add [H].
  Elan AddTheory expo [] A_fun_L_add.
  Elan AddRewriting expo A_axm_L_add.

  (* Recompile the rewrite system *)
  Recompile expo "localhost".

  ElanRewrite expo.
Qed.
```

5 Deduction modulo and the Noetherian induction principle

However, by using sophisticated termination orderings, proofs by Noetherian induction performed by rewriting are much more expressive than structural induction. We explore in this section how deduction modulo can provide the description, at the proof theoretical level, of proof by Noetherian induction. A proof search system for induction is proposed, based on a main induction rule that relies on a narrowing process to choose both the induction variables and the instantiation schema.

5.1 Deduction modulo and $HOL_{\lambda\sigma}$

Proof search engines like Spike [BKR92] or RRL [KZ95] allow to find proof of inductive properties, but they do not build the proof object that results from this proof search. Because we are working on the cooperation scheme between Coq and ELAN we need to exhibit an explicit proof of a given inductive statement. Therefore, we need to provide a proof theoretic setting that gives a detailled account of a noetherian induction principle use.

Being a bit more formal (but not yet completely), if we assume given a noetherian relation R and a user defined theory Th_u, we are looking for a proof of the proposition P using a noetherian induction principle denoted $NoethInd$, i.e. a derivation of the sequent: $NoethInd(R), Th_u \vdash P$. Therefore, in this section, we represent our proofs in an appropriate sequent calculus. But since the noetherian induction principle is by essence a second order proposition, we need to encode this sequent in higher-order logic. The idea to use rewrite concepts and techniques leads to consider mainly first-order theories and therefore motivates the use a first-order presentation of higher-order logic called $HOL_{\lambda\sigma}$ [DHK01] which is based on deduction modulo [DHK03]. It is clearly out of the scope of this paper to explain in detail the full approach, and we only provide here the main ideas. The reader can refer to [Dep02] and to [DK03] for a detailled exposition.

In deduction modulo, terms but also propositions can be identified modulo a congruence. We use a congruence that can typically be defined by conditional equations and that takes into account the context of application to evaluate the conditions. Furthermore, since the congruence application should be controled closely, an appropriate notion of protective symbol is used, see [Dep02]: indeed the congruence is not allowed to act below a protective symbol. In deduction modulo, the notions of term and proposition are that of (many sorted) first-order logic. We consider theories formed with a set of axioms Γ and a congruence, denoted $\sim$, defined on terms and propositions. This congruence takes three arguments: the two objects to be compared and a set of axioms Γ called a local context. When we want to emphasize this, we denote the congruence $\sim^\Gamma$. The deduction rules of the sequent calculus take this equivalence into account. For instance, the right rule for the conjunction is not stated as usual

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}$$

but is formulated

$$\frac{\Gamma \vdash_\sim A, \Delta \quad \Gamma \vdash_\sim B, \Delta}{\Gamma \vdash_\sim D, \Delta} \text{ if } D \sim^\Gamma A \wedge B.$$

We recall in Figure 2, the definition of the *sequent calculus modulo*. It extends the usual sequent calculus by working modulo the congruence $\sim$. In these rules, Γ and Δ are finite multisets of propositions, P and Q denote propositions. When the congruence $\sim$ is simply identity, this sequent calculus collapses to the usual one. In that case sequents are written as usual with the $\vdash$ symbol.

Proof checking decidability for the sequent calculus modulo reduces to the decidability of the relation $\sim^\Gamma$, since we can check for each rule that the conditions of application are satisfied and we provide the needed information in the quantifier rules. When $\sim^\Gamma$ is not decidable, we still can use

$$\frac{}{\Gamma, P \vdash_\sim Q}\text{axiom}\ \text{if}\ P \sim^\Gamma Q \qquad \frac{\Gamma, P \vdash_\sim \Delta \quad \Gamma \vdash_\sim Q, \Delta}{\Gamma \vdash_\sim \Delta}\text{cut}\ \text{if}\ P \sim^\Gamma Q$$

$$\frac{\Gamma, Q_1, Q_2 \vdash_\sim \Delta}{\Gamma, P \vdash_\sim \Delta}\text{contr-l}\ \text{if}\ (\mathbf{A}) \qquad \frac{\Gamma \vdash_\sim Q_1, Q_2, \Delta}{\Gamma \vdash_\sim P, \Delta}\text{contr-r}\ \text{if}\ (\mathbf{A})$$

$$\frac{\Gamma \vdash_\sim \Delta}{\Gamma, P \vdash_\sim \Delta}\text{weak-l} \qquad \frac{\Gamma \vdash_\sim \Delta}{\Gamma \vdash_\sim P, \Delta}\text{weak-r}$$

$$\frac{\Gamma, P, Q \vdash_\sim \Delta}{\Gamma, R \vdash_\sim \Delta}\wedge\text{-l}\ \text{if}\ R \sim^\Gamma (P \wedge Q) \qquad \frac{\Gamma \vdash_\sim P, \Delta \quad \Gamma \vdash_\sim Q, \Delta}{\Gamma \vdash_\sim R, \Delta}\wedge\text{-r}\ \text{if}\ R \sim^\Gamma (P \wedge Q)$$

$$\frac{\Gamma, P \vdash_\sim \Delta \quad \Gamma, Q \vdash_\sim \Delta}{\Gamma, R \vdash_\sim \Delta}\vee\text{-l}\ \text{if}\ (\mathbf{B}) \qquad \frac{\Gamma \vdash_\sim P, Q, \Delta}{\Gamma \vdash_\sim R, \Delta}\vee\text{-r}\ \text{if}\ (\mathbf{B})$$

$$\frac{\Gamma \vdash_\sim P, \Delta \quad \Gamma, Q \vdash_\sim \Delta}{\Gamma, R \vdash_\sim \Delta}\Rightarrow\text{-l}\ \text{if}\ (\mathbf{C}) \qquad \frac{\Gamma, P \vdash_\sim Q, \Delta}{\Gamma \vdash_\sim R, \Delta}\Rightarrow\text{-r}\ \text{if}\ (\mathbf{C})$$

$$\frac{\Gamma \vdash_\sim P, \Delta}{\Gamma, R \vdash_\sim \Delta}\neg\text{-l}\ \text{if}\ R \sim^\Gamma \neg P \qquad \frac{\Gamma, P \vdash_\sim \Delta}{\Gamma \vdash_\sim R, \Delta}\neg\text{-r}\ \text{if}\ R \sim^\Gamma \neg P$$

$$\frac{}{\Gamma, P \vdash_\sim \Delta}\bot\text{-l}\ \text{if}\ P \sim^\Gamma \bot$$

$$\frac{\Gamma, Q\{t/x\} \vdash_\sim \Delta}{\Gamma, P \vdash_\sim \Delta}(Q, x, t)\ \forall\text{-l}\ \text{if}\ (\mathbf{D}) \qquad \frac{\Gamma \vdash_\sim Q\{y/x\}, \Delta}{\Gamma \vdash_\sim P, \Delta}(Q, x, y)\ \forall\text{-r}\ \text{if}\ (\mathbf{E})$$

$$\frac{\Gamma, Q\{y/x\} \vdash_\sim \Delta}{\Gamma, P \vdash_\sim \Delta}(Q, x, y)\ \exists\text{-l}\ \text{if}\ (\mathbf{F}) \qquad \frac{\Gamma \vdash_\sim Q\{t/x\}, \Delta}{\Gamma \vdash_\sim P, \Delta}(Q, x, t)\ \exists\text{-r}\ \text{if}\ (\mathbf{G})$$

$\mathbf{A} = P \sim^\Gamma Q_1 \sim^\Gamma Q_2$, $\mathbf{B} = R \sim^\Gamma (P \vee Q)$ $\mathbf{C} = R \sim^\Gamma (P \Rightarrow Q)$, $\mathbf{D} = P \sim^\Gamma \forall x\ Q$, $\mathbf{E} = P \sim^\Gamma \forall x\ Q, y$ fresh variable, $\mathbf{F} = P \sim^\Gamma \exists x\ Q, y$ fresh variable, $\mathbf{G} = P \sim^\Gamma \exists x\ Q$

FIG. 2 – *The sequent calculus modulo*

instances for which one can check the conditions of application (for instance a constraint solving algorithm can be used).

We can now introduce the fundamental notion of compatibility: a theory (a set of propositions) $\mathcal{T}$ is said to be compatible with a congruence $\sim$ when:

$$\mathcal{T}, \Gamma \vdash \Delta \text{ if and only if } \Gamma \vdash_\sim \Delta.$$

This property is modular: if $\mathcal{T}_1$ is compatible with a congruence C_1 and $\mathcal{T}_2$ is compatible with C_2 then $\mathcal{T}_1 \cup \mathcal{T}_2$ is compatible with $C_1 \cup C_2$.

Using the above equivalence, we can internalize propositions into the congruence, and call this operation "push". We can also recover them at the level of the logic, and call this operation "pop". Moreover, thanks to modularity, this can be done dynamically during the proof. This duality between computation and deduction is very conveniently reflected by the compatibility property. In [DHK03], internalization has been done statically and used to identify computation within the deduction process. Our aim here is to do internalization dynamically and to use it to design rules for induction by rewriting and an adequate strategy for noetherian induction.

In what follows, we consider congruences generated by conditional class rewrite systems denoted $\mathcal{RE}$ and composed of (conditional) term rewrite rules, (conditional) term equational axioms, (conditional) proposition rewrite rules, (conditional) proposition equational axioms. Moreover, we assume the left-hand side of a proposition rewrite rule and both sides of a proposition equational axiom have to be atomic propositions. Conditions may be arbitrary propositions. The (free) variables in the right-hand side and condition of a rule must occur in the left-hand side. In the case of equational axioms, variables in both sides have to be the same and (free) variables in the condition have to be a subset of those.

We assume here that we work with the axioms of equality. In this case, to any conditional class rewrite system $\mathcal{RE}$ is associated the theory denoted $T_{\mathcal{RE}}$ as follows: for each conditional rewrite rule ($l \to r$ if c) or equational axiom ($l \approx r$ if c) in $\mathcal{RE}$, $T_{\mathcal{RE}}$ contains the proposition:

- $\forall \overline{x}(c \Rightarrow (l \Leftrightarrow r))$ when l and r are propositions,

– $\forall \overline{x}(c \Rightarrow (l \approx r))$ when l and r are terms,

where all free variables $\overline{x}$ are universally quantified.

It is proved in [Dep02] that $T_{\mathcal{RE}}$ is compatible with the congruence generated by $\mathcal{RE}$ (see also [Dow99] and [DHK03]). This allows us to freely use the “pushing and popping” paradigm. This also ensures that deduction modulo a congruence represented by a conditional class rewrite system is not a proper extension of first-order logic, but only a different presentation of it.

5.2 Deduction modulo for inductive proofs

This short introduction to deduction modulo now allows us to give a proof theoretic understanding of induction by rewriting. In the context of deduction modulo, the induction hypotheses arising from equational goals can be (dynamically) internalized into the congruence. When doing this, the computational part of the deduction modulo appears to perform exactly induction by rewriting as done for instance by systems like Spike [BKR92] or RRL [KZ95].

The powerful principle of these approaches is to allow application of induction hypotheses, current conjectures and axioms of the theory, at any position of the current goal, provided that the applied formula is smaller in the Noetherian induction ordering than the current goal. When the ordering contains the relation induced by a terminating rewrite system, a smaller formula is obtained as soon as a rewrite step is performed. Moreover, in Spike for instance, the choice of the induction variables and instantiation schemas is done using pre-calculated induction positions and schemas called test-sets. In the approach described below, we propose to use narrowing to automatically perform these choices.

Given a property P and a relation R defined on a type τ, the Noetherian induction principle $NoethInd(P, R, \tau)$ is defined as follows:

$$\forall x\, ((x \in \tau \wedge \forall y\, ((y \in \tau \wedge R(x, y)) \Rightarrow P(y))) \Rightarrow P(x)) \Rightarrow \forall x\, (x \in \tau \Rightarrow P(x))$$

and we write $Noeth(R, \tau)$ to state that R is a Noetherian relation over τ.

For proving that P inductively holds in a user theory Th_u, denoted $Th_u \models_{Ind} P$, it is enough to derive the sequent:

$$\forall R\, \forall \tau\, (Noeth(R, \tau) \Rightarrow \forall P\, NoethInd(P, R, \tau)), Th_u \vdash P.$$

Of course to finish the proof, one should also provide a proof of $Noeth(R, \tau)$. Considering an equational goal Q of the form $\forall x\, (x \in \tau \Rightarrow t_1(x) \approx t_2(x))$, the whole problem is formalized in $HOL_{\lambda\sigma}$ the first-order presentation of higher-order logic using deduction modulo. The remainder of this section gives the main steps which are detailed in [Dep02]. We start from the sequent:

$$\forall R\, \forall \tau\, (Noeth(R, \tau) \Rightarrow \forall P\, NoethInd(P, R, \tau)), Th_u \vdash \forall x\, (x \in \tau \Rightarrow t_1(x) \approx t_2(x)).$$

Choosing a specific relation R (written $\prec$) and a type still denoted τ, we get:

$$Noeth(\prec, \tau) \Rightarrow \forall P\, NoethInd(P, \prec, \tau)), Th_u \vdash \forall x\, (x \in \tau \Rightarrow t_1(x) \approx t_2(x)).$$

From this, by the rule $\Rightarrow$-l of the sequent calculus, we get on the one hand the sequent $Th_u \vdash Noeth(\prec, \tau)$ corresponding to the proof that $\prec$ is indeed Noetherian, on the other hand the sequent

$$\forall P\, NoethInd(P, \prec, \tau)), Th_u \vdash \forall x\, (x \in \tau \Rightarrow t_1(x) \approx t_2(x))$$

corresponding to the use of the induction principle to prove our goal.

We instantiate P to get:

$$\begin{array}{l}\forall x\, ((x \in \tau \wedge \forall \underline{x}\, ((\underline{x} \in \tau \wedge \underline{x} \prec x) \Rightarrow t_1(\underline{x}) \approx t_2(\underline{x}))) \Rightarrow t_1(x) \approx t_2(x)) \\ \Rightarrow \forall x\, (x \in \tau \Rightarrow t_1(x) \approx t_2(x)), Th_u \vdash \forall x\, (x \in \tau \Rightarrow t_1(x) \approx t_2(x))\end{array}$$

where we have renamed y to $\underline{x}$ to emphasize that $\underline{x}$ is a smaller instance of x. A few easy steps of the sequent calculus later, we get:

$$Th_u \vdash \forall x\, ((x \in \tau \wedge \forall \underline{x}\, ((\underline{x} \in \tau \wedge \underline{x} \prec x) \Rightarrow t_1(\underline{x}) \approx t_2(\underline{x}))) \Rightarrow t_1(x) \approx t_2(x))$$

We then instantiate x by a fresh variable that we call X to emphasize this status, and we get:

$$Th_u \vdash (X \in \tau \wedge \forall \underline{x}\,((\underline{x} \in \tau \wedge \underline{x} \prec X) \Rightarrow t_1(\underline{x}) \approx t_2(\underline{x}))) \Rightarrow t_1(X) \approx t_2(X).$$

The $\Rightarrow$-r and $\wedge$-l rules of the sequent calculus lead to the discovery of the induction hypothesis:

$$Th_u, X \in \tau, \forall \underline{x}\,((\underline{x} \in \tau \wedge \underline{x} \prec X) \Rightarrow t_1(\underline{x}) \approx t_2(\underline{x})) \vdash t_1(X) \approx t_2(X).$$

Using what we have seen on compatible theories, this hypothesis, can now be internalized as a conditional equation denoted in general $\mathcal{RE}_{ind(Q)}$:

$$t_1(\underline{x}) \approx t_2(\underline{x}) \text{ if } \underline{x} \in \tau \wedge \underline{x} \prec X \tag{1}$$

Note that because of its status of free fresh variable, X behaves like a constant.

What is crucial in using the induction hypothesis (1) as an equation or a rewrite rule, is to check its condition. For any many-sorted theory, the $\underline{x} \in \tau$ part of the condition is trivial. More interestingly, the $\underline{x} \prec X$ condition is *always* satisfied provided the following hypotheses (called $\mathcal{H}$) are imposed:
(i) the theory Th_u can be oriented into a Noetherian rewrite system $\mathcal{R}$,
(ii) we choose for $\prec$ the reduction ordering induced by $\mathcal{R}$,
(iii) (1) is only applied on a subterm of the goal $t_1 \approx t_2$ or on a $\mathcal{R}$-reduced form of this goal.

Under these hypotheses, we are left to derive the sequent

$$Th_u, X \in \tau \vdash_{\mathcal{R}, t_1(\underline{x}) \approx t_2(\underline{x})} t_1(X) \approx t_2(X).$$

in the sequent calculus modulo. To be able to satisfy the (iii) part of the $\mathcal{H}$ hypotheses, we need in general to use the information that $X \in \tau$ in order to instantiate X by the free constructors of τ. This idea is exploited in the next section to provide the proof search strategy.

5.3 Proof search by narrowing

The rules of the proof system in Figure 3 apply on sequents modulo of the form $\Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} Q$, where Γ_1 is the deduction part of the definitions, $\mathcal{RE}_1$ is their computational part; Γ_2 is the deduction part for other statements, $\mathcal{RE}_2$ is their computational part; Q is an equational goal. The distinction between $\Gamma_1/\mathcal{RE}_1$ and $\Gamma_2/\mathcal{RE}_2$ is needed because in the *Induce* rule, only $\mathcal{RE}_1$ is used for narrowing. The initial Γ_2 may contain lemmas. $\mathcal{RE}_2$ receives the induction hypotheses. Sequents are gathered in a multiset structure modeled with the $\bullet$ operator that is an AC operator on sequents with $\diamond$ as a neutral element.

The main rule is *Induce* as it performs the induction step. It uses narrowing to choose both the induction variable(s) and the instantiation schema. The other rules are *Trivial* which eliminates a trivial equation. *Push* pushes an equational hypothesis from the deduction part to the computational part, *Orient* orients an equation in the computational part into a rewrite rule, according to the term ordering, *Rewrite* rewrites using a rule or an equation and orients the step using an ordering on equations built upon the term ordering. *Push*, *Orient* and *Rewrite* are duplicated because of the $\Gamma_1/\mathcal{RE}_1$ and $\Gamma_2/\mathcal{RE}_2$ distinction.

Here is a simple example of proof by induction on natural numbers using our proof system:

	$x+0 \approx x, x+s(y) \approx s(x+y) \mid \vdash_{\mid} 0+x \approx x$	
$\rightarrowtail$	$x+0 \approx x \mid \vdash_{x+s(y)\approx s(x+y)\mid} 0+x \approx x$	Push$_1$
$\rightarrowtail$	$\mid \vdash_{x+0\approx x, x+s(y)\approx s(x+y)\mid} 0+x \approx x$	Push$_1$
$\rightarrowtail$	$\mid \vdash_{x+0\approx x, x+s(y)\rightarrow s(x+y)\mid} 0+x \approx x$	Orient$_1$
$\rightarrowtail$	$\mid \vdash_{x+0\rightarrow x, x+s(y)\rightarrow s(x+y)\mid} 0+x \approx x$	Orient$_1$
$\rightarrowtail$	$\mid \vdash_{x+0\rightarrow x, x+s(y)\rightarrow s(x+y)\mid 0+\underline{x}\approx\underline{x}} 0 \approx 0$ $\mid \vdash_{x+0\rightarrow x, x+s(y)\rightarrow s(x+y)\mid 0+\underline{x}\approx\underline{x}} s(0+y) \approx s(y)$	Induce
$\rightarrowtail$	$\mid \vdash_{x+0\rightarrow x, x+s(y)\rightarrow s(x+y)\mid 0+\underline{x}\approx\underline{x}} s(0+y) \approx s(y)$	Trivial
$\rightarrowtail$	$\mid \vdash_{x+0\rightarrow x, x+s(y)\rightarrow s(x+y)\mid 0+\underline{x}\rightarrow\underline{x}} s(0+y) \approx s(y)$	Orient$_2$
$\rightarrowtail$	$\mid \vdash_{x+0\rightarrow x, x+s(y)\rightarrow s(x+y)\mid 0+\underline{x}\rightarrow\underline{x}} s(y) \approx s(y)$	Rewrite$_2$
$\rightarrowtail$	$\diamond$	Trivial

Induce $\quad \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} Q[t]_\omega \rightarrowtail \bullet_{\substack{R\diamond E \in \mathcal{RE}_1 \\ \sigma \in Unif(t,l) \\ \sigma(l) > \sigma(r)}} \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2 \cup \mathcal{RE}_{ind(Q)}} \sigma(Q[r]_w)$

$(R\diamond E = l \rightarrow r \text{ or } R\diamond E = l \approx r \text{ or } R\diamond E = r \approx l), \omega \in GP_{\mathcal{RE}_1}(Q)$

Push$_1$ $\quad \Gamma_1, l \approx r|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} Q \rightarrowtail \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1 \cup \{l \approx r\}|\mathcal{RE}_2} Q$

Orient$_1$ $\quad \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1 \cup E|\mathcal{RE}_2} Q \rightarrowtail \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1 \cup \{l \rightarrow r\}|\mathcal{RE}_2} Q$

$E = l \approx r \text{ or } E = r \approx l$

$l > r$

Push$_2$ $\quad \Gamma_1|\Gamma_2, l \approx r \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} Q \rightarrowtail \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2 \cup \{l \approx r\}} Q$

Orient$_2$ $\quad \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2 \cup E} Q \rightarrowtail \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2 \cup \{l \rightarrow r\}} Q$

$E = l \approx r \text{ or } E = r \approx l$

$l > r$

Rewrite$_1$ $\quad \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1 \cup R\diamond E|\mathcal{RE}_2} Q[\sigma(l)]_\omega \rightarrowtail \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1 \cup R\diamond E|\mathcal{RE}_2} Q[\sigma(r)]_\omega$

$R\diamond E = l \rightarrow r \text{ or } R\diamond E = l \approx r \text{ or } R\diamond E = r \approx l$

$Q[\sigma(l)]_\omega \succ Q[\sigma(r)]_\omega$

Rewrite$_2$ $\quad \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2 \cup R\diamond E} Q[\sigma(l)]_\omega \rightarrowtail \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2 \cup R\diamond E} Q[\sigma(r)]_\omega$

$R\diamond E = l \rightarrow r \text{ or } R\diamond E = l \approx r \text{ or } R\diamond E = r \approx l$

$Q[\sigma(l)]_\omega \succ Q[\sigma(r)]_\omega$

Trivial $\quad \Gamma_1|\Gamma_2 \vdash_{\mathcal{RE}_1|\mathcal{RE}_2} t \approx t \rightarrowtail \diamond$

'$\bullet$' is an AC operator with neutral element '$\diamond$'.

FIG. 3 – *SIADM: A simple proof-search system for induction as deduction modulo*

5.4 Complete narrowing for proof search

To make precise the use of narrowing in the induction process, let us introduce a few notations. Let $\mathcal{R}$ be a term rewriting system. The signature Σ is partitioned into a set of free constructors and a set of defined symbols. Free constructors are constructors which are not related with each other by any rule. A constructor term is a term built only with constructor symbols. A ground substitution is a substitution mapping each variable to a ground term, i.e. a term without variables. The set of positions of a term t is denoted $Dom(t)$, the subterm of t at position ω is denoted $t_{|\omega}$ and the symbol at position ω in t by $t(\omega)$. The notation $t[u]_\omega$ means that the term t contains the subterm u at position ω. These notations extend to goals.

A rewrite system is said to be *ground convergent* if it is confluent and terminating over the set of ground terms. It is said to be *sufficiently complete* if any ground term can be reduced into a (ground) constructor term.

A goal Q is narrowed into Q' at a position ω with the rule $l \rightarrow r$ and the substitution σ, if σ is the most general unifier of l and $Q_{|\omega}$, and $Q' = \sigma(Q[r]_\omega)$. The narrowing relation is denoted by $Q \rightsquigarrow_{l \rightarrow r, \omega, \sigma} Q'$.

We partition $\mathcal{R}$ into subsets $\mathcal{R}_f = \{l \rightarrow r \in \mathcal{R} | l(\varepsilon) = f\}$ for each defined symbol. When a variable is involved in an induction process, it must be instantiated with all possible values, in order to cover all possible cases. Since the system is sufficiently complete, the rules in $\mathcal{R}_f$ do cover all cases for f. This idea leads to the following notion of good positions, which states that *all* rules in $\mathcal{R}_f$ *do* narrow at these positions. The set of *good positions* in a goal Q is defined by:
$GP_{\mathcal{R}}(Q) = \{\omega \in Dom(Q) | \forall l \rightarrow r \in \mathcal{R}_{Q(\omega)}, \exists Q' : Q \rightsquigarrow_{l \rightarrow r, \omega} Q'\}$

For an equational goal Q of the form $t_1 \approx t_2$, we can show that when $\mathcal{R}$ is ground convergent

and sufficiently complete, if $Q \rightarrowtail Q_1 \bullet \cdots \bullet Q_n$ by the *Induce* rule, then $\forall i, \mathcal{R} \models_{Ind} Q_i$ implies $\mathcal{R} \models_{Ind} Q$.

We then get the correctness of the proof search system:

Theorem 0.2 *Let $\mathcal{R}$ be ground convergent and sufficiently complete. If $Q \rightarrowtail^+ \diamond$, then $\mathcal{R} \models_{Ind} Q$.*

6 Translating induction proofs into Coq

6.1 Structural induction

Our goal is to translate automatic (or semi-automatic) structural induction proofs by ELAN to Coq. The detection of induction variable can be done in ELAN using automatic proof search techniques (i.e. narrowing or test-sets [BKR92]) by user interaction.

Proof terms in ELAN An induction step in ELAN is represented by the induction variable x. The proof term for ELAN is a n-tuple $\langle ind, x, \pi_b, \pi_i \rangle$ where ind is the induction principle, π_b and π_i are respectively the proof terms for the base case and the induction case.

Proof term in Coq

$$\lambda \vec{X}\ (ind\ \lambda n : T.P\{x/n\}\ (\Pi 2Coq(\rho\sigma 2\Pi(\pi_b))\ \ \vec{X})\ (\Pi 2Coq(\rho\sigma 2\Pi(\pi_i))\ \ \vec{X}))$$

where $\vec{X}$ is the sequence of universially quatified varibles, ind is the Coq induction principle for the type T; P is the proposition to prove; x is the induction variable.

6.2 Noetherian induction

Translating Noetherian inductions into Coq seems to be less straightforward. Given the description of the proof search system in section 5.3, the next step is to implement in ELAN the described rules and to design an adequate strategy for proof search. In case the system finds a proof, it is represented as a $\rho\sigma$-term in which the branching step involved in the *Induce* rule is expressed using sets of proof terms.

It remains to translate this $\rho\sigma$-term into a proof term in Coq. Using Noetherian induction induced by the rewrite system needs a powerful principle, already present in Coq.

Given a Noetherian relation R on the type A:

```
well_founded_ind:
  (A:Set; R:(A->A->Prop)) (well_founded A R)
  ->(P:(A->Prop))((x:A)((y:A)(R y x)->(P y))->(P x))->(a:A)(P a)
```

Indeed using this principle requires a proof that the relation R is well-founded. So a full formalization of implicit induction proofs will need formal proofs of termination of the rewrite systems. Standard techniques of proving termination in term rewriting, for example based on precedence on function symbols, such as the lexicographic path ordering (LPO), are already available. In [GL01], a constructive proof of termination is given for a generalization of path orderings that applies to any kind of structure with a well-founded notion of immediate substructure. Path orderings (LPO, RPO, MPO) are then easy to obtain as generalizations. In [Lec95], it is proved that the multiset path ordering (MPO) terminates, and that it enjoys the properties required to prove termination of rewrite systems, i.e. that it is a simplification ordering.

Summarizing, one of the main interest of this approach is that ELAN provides to Coq not only the proof term but the proof principle under which the proof can be done. We therefore bridge in an elegant and safe way rewriting techniques with proof assistant capabilities.

7 Coq-ELAN for Induction - future work

We draw in this section some ideas on the future ELAN based tactics for induction in Coq. Two different version can be considered depending on the level of proof automation.

7.1 Semi-automatic version

In this version, no search is needed in ELAN. The induction variables and priciples are provided by Coq user. The rewrite system used for simplifying goals is also given. In this case, ELAN needs to apply the induction principle on the right positions and to try to prove automatically the generated subgoal using rewriting. In case of failure, these subgoals are left to Coq user.

Two possible syntaxes:

$$\text{ElanInduction } \mathcal{X}\ R\ \pi_R^N\ th\ machine$$

or

$$\text{ElanInduction } \mathcal{X}\ P\ th\ machine$$

where

- $\mathcal{X}$ is the list of induction variables used in the proof
- P is the used induction principle
- R is the used well-founded ordering
- π_R^N is a proof of the well-founded property of R
- th is the rewrite system used for simplifying goals
- *machine* is the machine where the ELAN server is launched

For example :

ElanInduction `x lt lt_wf peano "localhost"`.

or

ElanInduction `x nat_ind2 peano "localhost"`.

7.2 Automatic version

In this version, ELAN searches for the induction variables and principles and applies them to prove the current goal. This process is semi-decidable and an exit point is needed:

$$\text{ElanInduction } th\ machine$$

8 Conclusion

Checking automatic proof in proof assistants has been studied by numerous researchers. To this end, the external proof can be translated either (1) into a script to be re-executed in the proof assistants or (2) into a low-level proof term. The first approach has been experimented on several interactive theorem provers such as HOL, Isabelle or PVS. In this vein, model checkers [JS94, Rus99], computer algebra systems [HT98] or automatic first-order theorem provers [Hur99, ABH+98] [BSBG98, Pau99] were combined to interactive theorem provers in order to improve the automatic level of proof search.

The second approach can only be applied to proof assistant using explicit proof terms but is more secure since we only need to trust on a small kernel of these proof assistants. The proof checking, that consists only in type checking the proof term, seems to be also more efficient. Several experimentations on checking imported proof terms has been done in Coq and Twelf [PS99]. To speed up proof checking, optimizing the generated proof term has attracted much attention. In [BHdN02], the smartly represented resolution proofs are translated into proof terms to be checked by Coq. In [SD02, NL98], proofs are checked in LF with appropriate optimizations to speed up proof checking.

Towards a more scalable framework, there are Necula's work on *proof-carrying code* (PCC) [Nec97] and Appel and Felty's work [AF00, App01] on *foundational proof-carrying code*. The key idea of these frameworks is to associate a proof of safety properties to machine-language programs generated by the code producer. On the other side, the code consumer uses an as small and trustworthy as possible proof checker for verifying the associated proof before executing these programs.

The originality of the work presented here relies on the proposition of performing proof of equational and inductive theorems by rewriting techniques that can be delegated to an automated prover, while being able to build a proof term to be checked by a proof assistant relying on constructive type theory. However in order to get there, the notion of proof term and the underlying $\rho\sigma$-calculus, as well as deduction modulo, are needed to set up an adequate theoretical background. Also mandatory to develop this work were mastering the compilation techniques for AC rewriting and strategies, the rewriting based induction techniques, and the practice of a proof assistant based on the calculus of inductive constructions.

Much work is yet needed to achieve our initial goal to understand theoretical and practical aspects of proof construction and verification. An ambitious goal concerns the design of a development framework for certified and modular software, whose security properties must be formally asserted. The proofs would be done by an incorporated proof builder, based upon deduction modulo and combining different provers specialized to given theories. When completed, the proof could be checked by the proof checker. Once achieved and checked, the proof could be recorded to be offered to any a posteriori verification purpose.

At the proof level, the general framework of deduction modulo is quite relevant to keep at the deduction level only the true deduction steps like modus ponens and to delegate all computational steps on propositions or terms to specialized provers using equational and rewriting techniques. Then, some parts of the proofs can be deferred to aside computations while the true skeleton of the proof is being built. At the checking level, the experiences described here of translating equational and inductive proofs to proof terms for Coq should be quite useful.

Références

[ABH+98] W. Ahrendt, B. Beckert, R. Hähnle, W. Menzel, W. Reil, G. Schellhorn, and P. Schmitt. Integrating automated and interactive theorem proving. In W. Bibel and P. Schmitt, editors, *Automated - A Basis for Applications*, volume 1. Kluwer Academic Publishers, 1998.

[ACCL90] 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.

[AF00] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. of 27th POPL*, pages 243–253, January 2000.

[App01] Andrew W. Appel. Foundational proof-carrying code. In *Proc. 16th LICS*, pages 247–258, June 2001.

[BB02] Henk Barendregt and Erik Barendsen. Autarkic computations in formal proofs. *Journal of Automated Reasoning*, 28(3):321–336, April 2002.

[BHdN02] M. Bezem, D. Hendriks, and H. de Nivelle. Automated proof construction in type theory using resolution. *Journal of Automated Reasoning*, 29(3-4):253–275, 2002.

[BKKM02] Peter Borovansky, Claude Kirchner, Hélène Kirchner, and Pierre-Etienne Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, (285):155–185, July 2002.

[BKR92] Adel Bouhoula, E. Kounalis, and M. Rusinowitch. Spike: An automatic theorem prover. In *Proceedings of the 1st International Conference on Logic Programming and Automated Reasoning, St. Petersburg (Russia)*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 460–462, July 1992.

[Bou97] S. Boutin. Using reflection to build efficient and certified decision procedures. In M. Abadi and T. Ito, editors, *Proc. TACS'97*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529, 1997.

[BSBG98] R. Boulton, K. Slind, A. Bundy, and M. Gordon. An interface between clam and HOL. In J. Grundy and M. Newey, editors, *Proc. of the 11th Int. Conf. TPHOL*, volume 1479 of *Lecture Notes in Computer Science*, pages 87–104, 1998.

[CH88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76, 1988.

[CHL96] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, March 1996.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Cir00] H. Cirstea. *Calcul de réécriture : fondements et applications.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, October 2000.

[CK01] Horatiu Cirstea and Claude Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.

[Dep02] Eric Deplagne. *Système de preuve modulo récurrence.* Thèse de doctorat, Université Nancy 1, November 2002.

[DHK01] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. HOL-$\lambda\sigma$ an intentional first-order expression of higher-order logic. *Mathematical Structures in Computer Science*, 11(1):21–45, 2001.

[DHK03] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 2003. To appear. See also `ftp://ftp.inria.fr/INRIA/publication/RR/RR-3400.ps.gz`.

[DK03] Eric Deplagne and Claude Kirchner. Induction as deduction modulo. research report, Loria, 2003. in preparation.

[Dow99] G. Dowek. *La part du Calcul.* 1999. Mémoire d'habilitation, Université de Paris 7.

[GL01] Jean Goubault-Larrecq. Well-founded recursive relations. In *Proc. 15th Int. Workshop Computer Science Logic (CSL'2001)*, LNCS 2142, pages 484–497. Springer, Paris, France, 2001.

[Har95] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, UK, 1995. Available at `http://www.cl.cam.ac.uk/users/jrh/papers/reflect.dvi.gz+`.

[HT98] J. Harrison and L. Théry. A sceptic's approach to combining hoal and mapple. In *Journal of Automated Reasoning*, volume 21, pages 279–294, 1998.

[Hur99] J. Hurd. Integrating gandalf and HOL. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Proc. of TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 311–321, 1999.

[JK86] J.-P. Jouannaud and Hélène Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986.

[JS94] J. Joyce and C. Seger. The HOL-Voss system: Model-checking inside a general-purpose theorem-prover. *Lecture Notes in Computer Science*, 780:185–198, 1994.

[KM01] Hélène Kirchner and Pierre-Etienne Moreau. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 11(2):207–251, 2001. Report LORIA A01-R-007.

[KZ95] Deepak Kapur and Hantao Zhang. An overview of rewrite rule laboratory (rrl). *J. Computer and Mathematics with Applications*, 29(2):91–114, 1995.

[Lec95] François Leclerc. Termination proof of term rewriting systems with the multiset path ordering: A complete development in the system coq. In *Proc. 2nd Int. Conf. on Typed Lambda Calculi and Applications (TLCA-95)*, LNCS 902, pages 312–327. Springer, Edinburgh, UK, 1995.

[Nec97] George C. Necula. Proof-carrying code. In *Proc. of 24th POPL*, pages 106–119, January 1997.

[Ngu02a] Q-H. Nguyen. *Calcul de réécriture et automatisation du raisonnement dans les assistants de preuve.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, Octobre 2002.

[Ngu02b] Q.-H. Nguyen. A constructive decision procedure for equalities modulo AC. In C. Ringeissen, C. Tinelli, R. Treinen, and R.M. Verma, editors, *Proc. of 18th UNIF*, pages 59–63, 2002.

[NKK02] Q.-H. Nguyen, C. Kirchner, and H. Kirchner. External rewriting for skeptical proof assistants. *Journal of Automated Reasoning*, 29(3-4):309–336, 2002.

[NL98] G.C. Necula and P. Lee. Efficient representation and validation of logical proofs. In *Proc. LICS'98*, pages 93–104, June 1998.

[Pau99] L.C. Paulson. A Generic Tableau Prover and its Integration with Isabelle. *Journal of Universal Computer Science*, 5(3):73–87, March 1999.

[PM93] 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.

[PS81] G. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28:233–264, 1981.

[PS99] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proc. of CADE 16*, volume 1632 of *Lecture Notes in Computer Science*, July 1999.

[Rus99] J. Rushby. Integrated formal verification: Using model checking with automated abstraction, invariant generation, and theorem proving. In D Dams, R Gerth, S Leue, and M Massinek, editors, *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*, pages 311–321, 1999.

[SD02] A. Stump and D. L. Dill. Faster proof checking in the Edinburgh Logical Framework. In *Proc. of CADE 18*, volume 2392 of *Lecture Notes in Computer Science*, pages 391–406, 2002.