Certification of automated termination proofs *

Evelyne Contejean[1], Pierre Courtieu[2], Julien Forest[2], Olivier Pons[2], and Xavier Urbain[2]

[1] LRI – CNRS – Université Paris-Sud
[2] CÉDRIC – Conservatoire national des arts et métiers

Abstract. Nowadays, formal methods rely on tools of different kinds: proof assistants with which the user interacts to discover a proof step by step; and fully automated tools which make use of (intricate) decision procedures. But while some proof assistants can *check* the soundness of a proof, they lack automation. Regarding automated tools, one still has to be satisfied with their answers `Yes`/`No`/`Do not know`, the validity of which can be subject to question, in particular because of the increasing size and complexity of these tools.
In the context of rewriting techniques, we aim at bridging the gap between proof assistants that yield formal guarantees of reliability and highly automated tools one has to trust. We present an approach making use of both shallow and deep embeddings. We illustrate this approach with a prototype based on the C*i*ME rewriting toolbox, which can discover involved termination proofs that can be certified by the COQ proof assistant, using the COCCINELLE library for rewriting.

1 Introduction

Nowadays, formal methods play an increasingly important role when it comes to guaranteeing good properties for complex, sensitive or critical systems. In the context of proving, they rely on tools of different kinds: proof assistants with which the user interacts to discover a proof step by step; and fully automated tools which make use of (intricate) decision procedures.

Indeed, discovering a proof step by step has a heavy cost. Reducing this cost as much as possible amounts to using more and more automation. However, while some proof assistants can *check* the soundness of a proof, one still has to be satisfied with the answers of automated tools: `Yes`/`No`/`Do not know`. Yet, since application fields include (amongst others) sensitive and possibly critical sectors: security, code verification, cryptographic protocols, etc., **reliance on verification tools is crucial**.

Some proof assistants, like COQ [24], need to check mechanically the proof of each notion used. Amongst the strengths of these assistants are firstly a powerful specification language that can express both logical assertions and programs, hence properties of programs; and secondly a *highly reliable* procedure that checks the soundness of proofs.

COQ or ISABELLE/HOL [23], for instance, have a small highly reliable *kernel*. In COQ, type checking of a *proof term* is performed by the kernel to ensure a proof's soundness. Certified-programming environments based on these proof assistants find

* This work is partially supported by the A3PAT project of the French ANR (ANR-05-BLAN-0146-01).

here an additional guarantee. Yet, amongst the weaknesses of these assistants, one may regret the lack of automation in the proof discovery process.

Automation is actually difficult to obtain in this framework: in order to be accepted by the proof assistant, a property proven by an external procedure has to be *checked* by the proof assistant. Therefore, the procedure has to return a *trace* of the proof that is understood by the proof assistant. We want of course proofs that are *axiom free*.

We want to meet the important need to *delegate proofs* of some properties in the framework of rewriting techniques. We will focus on generic ways to provide reasonably-sized proof traces for complex properties. It would thus improve communication between proof assistants and powerful automated provers.

The present work focuses on proofs of *termination*: the property of a function/program any execution (computation) of which always yields a result. Fundamental when recursion and induction are involved, it is an unavoidable preliminary for proving many various properties of a program. Confluence of a rewriting system for instance, becomes decidable when the system terminates. More generally, proving termination is a boundary between *total* and *partial* correctness of functions and programs. Hence, automating termination is of great interest for provers like Coq, in which functions can be defined only if they are proven to be terminating. We restrict here to first order.

The last decade has been very fertile w.r.t. automation of termination proofs, and yielded many efficient tools (AProVE [15], C*i*ME [7], Jambox [13], TPA [19], TTT [17] and others) referenced on the Termination Competition's web site [21], some of which display nice output for *human* reading. However, there is still a clear gap between proof assistants that provide formal guarantees of reliability and highly automated tools one has to trust.

In the sequel, we aim at bridging this gap. We shall precise our notations and some prerequisites about first order term rewriting and about the Coq proof assistant in Section 2. Then, in Section 3, we shall address the problem of modelling termination of rewriting in Coq. Rather than relying on shallow or deep embedding, our approach tries to take the full benefit of *both*, depending on the addressed criterion (see Section 4). With the help of the Coccinelle library dedicated to that purpose, we enable the certification of proofs using involved criteria such as Dependency Pairs [1] with graphs refinement, and mixing orderings based on polynomial interpretations [20] or RPO [9] with AFS [1]. We illustrate our approach with a prototype based on the C*i*ME tool box, the proofs of which can be certified using the Coq proof assistant. We shall adopt the end-user point of view and provide some experimental results in Section 5. Eventually we briefly compare with related works and conclude in Section 6.

2 Preliminaries

2.1 Rewriting

We assume the reader familiar with basic concepts of term rewriting [2, 11] and termination, in particular with the Dependency Pairs (DP) approach [1]. We recall usual notions, and give our notations. A *signature* $\mathcal{F}$ is a finite set of *symbols* with arities. Let X be a countable set of *variables*; $T(\mathcal{F}, X)$ denotes the set of finite *terms* on $\mathcal{F}$

and X. $\Lambda(t)$ is the symbol at root position in Term t. We write $t|_p$ for the subterm of t at position p. A *substitution* is a mapping σ from variables to terms; we use postfix notation for their applications. A substitution can easily be extended to endomorphisms of $T(\mathcal{F}, X)$: $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$. Then, $t\sigma$ is called an *instance* of t.

A *rewrite relation* is a binary relation $\rightarrow$ on terms which is monotonic and closed under substitution; $\rightarrow^\star$ will denote its reflexive-transitive closure. A *term rewriting system* (TRS for short) over a signature $\mathcal{F}$ and a set of variables X is a (possibly infinite) set $R(\mathcal{F})$ of *rewrite rules* $l \rightarrow r$. A TRS R defines a rewrite relation $\rightarrow_R$ in the following way: $s \rightarrow_R t$ if there is a position p such that $s|_p = l\sigma$ and $t = s[r\sigma]_p$ for a rule $l \rightarrow r \in R$ and a substitution σ. We say then that s *reduces to* t *at position* p *with* $l \rightarrow r$ denoted $s \xrightarrow[l \rightarrow r]{p} t$. Symbols occurring at root position in the left-hand sides of rules in R are said to be *defined*, the others are said to be *constructors*. The set of *unmarked dependency pairs* of a TRS R, denoted $DP(R)$ is defined as $\{\langle u, v\rangle$ such that $u \rightarrow t \in R$ and $t|_p = v$ and $\Lambda(v)$ is defined$\}$.

A term is R-*strongly normalizable* (SN) if it cannot reduce infinitely many times for the relation defined by system R. A rewrite relation is *terminating* or *terminates* if any term is SN. In the following, we shall omit signatures, systems and positions that are clear from the context, and we shall restrict to finite systems.

Termination is usually proven with the help of *reduction orderings* [10] or quasi-orderings with dependency pairs. We briefly recall what we need. An *ordering pair* is a pair $(\succeq, >)$ of relations over $T(\mathcal{F}, X)$ such that: 1) $\succeq$ is a quasi-ordering, i.e. reflexive and transitive, 2) $>$ is a strict ordering, i.e. irreflexive and transitive, and 3) $> \cdot \succeq = >$ or $\succeq \cdot > = >$. We will refer to these as *term orderings*. A term ordering is said to be *well-founded* if there is no infinite strictly decreasing sequence $t_1 > t_2 > \cdots$; *stable* if both $>$ and $\succeq$ are stable under substitutions; *weakly* (resp. *strictly*) *monotonic* if for all terms t_1 and t_2, for all $f \in \mathcal{F}$, if $t_1 \succeq$ (resp. $>$) t_2 then $f(\ldots, t_1, \ldots) \succeq$ (resp. $>$) $f(\ldots, t_2, \ldots)$; a term ordering $(\succeq, >)$ is called a *weak (resp. strict) reduction ordering* if it is well-founded, stable and weakly (resp. strictly) monotonic.

2.2 The COQ proof assistant

The COQ proof assistant is based on *type theory*. It consists of:

- A *formal language* to express objects, properties and proofs in a unified way; all these are represented as terms of an expressive λ-calculus: the *Calculus of Inductive Constructions* (CIC) [8].
- A *proof checker* which checks the validity of proofs written as CIC-terms. Indeed, in this framework, a term is a *proof* of its type, and checking a proof consists in typing a term. The tool's correctness relies on this type checker, which is a small kernel of 5 000 lines of OBJECTIVE CAML code.

For example the following simple terms are proofs of the following (tautological) types (remember that implication arrow `->` is right associative): the identity function `fun x:A => x` is a proof of `A->A`, and `fun (x:A) (f:A->B) => f x` is a proof of `A->(A->B)->B`.

A very powerful feature of Coq is the ability to define *inductive types* to express inductive data types and inductive properties. For example the following inductive types define the data type `nat` (itself of type `Set`) of natural numbers and the property `even` of being an even natural number. `O` and `S` (successor) being the two constructors of `nat`[3].

```
Inductive nat : Set := | O : nat | S : nat -> nat.
Inductive even : nat -> Prop := | even_O : even O
  | even_S : ∀n : nat, even n -> even (S (S n)).
```

Hence the term `even_S (even_S (even_O))` is of type `even (S (S (S (S O))))` so it is a proof that 4 is even.

2.3 Termination in Coq

We focus in this paper onto termination. This property is defined in Coq standard library as the well-foundedness of an *ordering*. Hence we model TRS as *orderings* in the following. This notion is defined using the *accessibility* predicate. A term $t : A$ is accessible for an ordering $<$ if all its predecessors are, and $<$ is well-founded if all terms of type A are accessible (`R y x` stands for $y < x$):

```
Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A) : Prop :=
  | Acc_intro : (∀ y : A, R y x -> Acc R y) -> Acc R x
Definition well_founded (A : Type) (R : A -> A -> Prop) :=
  ∀ a : A, Acc R a.
```

3 Modelling termination of rewriting in Coq

If `R` is the relation modelling a TRS $\mathcal{R}$, we should write `R` u t (which means $u < t$) when a term t rewrites to a term u. For the sake of readability we will use as much as possible the Coq notation: t `-[R]>` u (and t `-[R]*>` u for $t \rightarrow^* u$) instead.

The wanted final theorem stating that $\mathcal{R}$ is terminating has the following form:

```
Theorem well_founded_R: Well_founded R.
```

Since we want certified automated proofs, the definition of `R` and the proof of this theorem are discovered and generated in Coq syntax *with full automation* by our prototype. In order to ensure that the original rewriting system $\mathcal{R}$ terminates, the only things the user has to check is firstly that the generated relation `R` corresponds to $\mathcal{R}$ (which is easy as we shall see in Section 3.2), and secondly that the generated Coq files do compile.

3.1 Shallow vs deep embedding

In order to prove properties on our objects (terms, rewriting systems, polynomial interpretations...), we have to model these objects in the proof assistant by defining a theory of rewriting. There are classically two opposite ways of doing this: *shallow embedding*

[3] Note that this notion of constructors is different from the one in Section 2.1.

and *deep embedding*. When using shallow embedding, one defines *ad hoc* translations for the different rewriting notions, and then *instantiates* criteria on the particular encoding of those notions. When using deep embedding, one defines *generic* notions for rewriting and *proves* generic criteria on them, and then instantiates notions and criteria on the considered system. Both shallow and deep embedding have advantages and drawbacks. On the plus side of shallow embedding are: an easy implementation of rewriting notions, and the absence of need of meta notions (as substitutions or term well-formedness w.r.t. a signature). On the minus side, one cannot certify a *criterion* but only its *instantiation* on a particular problem, which often leads to large scripts and proof terms. Regarding deep embedding, it leads usually to smaller scripts and proof terms since one can reuse generic lemmas but at the cost of a rather technical first step consisting in defining the generic notions and stating and *proving* generic lemmas.

We present here an hybrid approach where some notions are deep (Σ-algebra, RPO) and others are shallow (rewriting system, dependency graphs, polynomial interpretations). The reason for this is mainly due to our *proof* concern which makes sometime deep embedding not worth the efforts it requires (some generic premises can be rather tricky to fulfil). We will show that using both embeddings in a single proof is not a problem, and moreover that we can take full benefit of both.

3.2 The COCCINELLE library

The deep part of the modelling is formalised in a public COQ library called COCCINELLE. To start with, COCCINELLE contains a modelling of the mathematical notions needed for rewriting, such as term algebras, generic rewriting, generic and AC equational theories and RPO with status. It contains also properties of these structures, for example that RPO is well-founded whenever the underlying precedence is so.

Moreover COCCINELLE is intented to be a mirror of the C*i*ME tool in COQ; this means that some of the types of COCCINELLE (terms, etc.) are translated from C*i*ME (in OBJECTIVE CAML) to COQ, as well as some functions (AC matching)[4]. Translating functions and proving their full correctness obviously provide a certification of the underlying algorithm. Moreover, some proofs may require that *all* objects satisfying a certain property have been built: for instance in order to prove local confluence of a TRS, one need to get all critical pairs, hence a unification algorithm which is complete[5].

Since module systems in OBJECTIVE CAML and COQ are similar, both C*i*ME and COCCINELLE have the same structure, except that C*i*ME contains only types and functions whereas COCCINELLE also contains properties over these types and functions.

Terms A signature is defined by a set of symbols with decidable equality, and a function `arity` mapping each symbol to its arity.

[4] It should be noticed that COCCINELLE is not a *full* mirror of C*i*ME: some parts of C*i*ME are actually search algorithms for proving for instance equality of terms modulo a theory or termination of TRSs. These search algorithms are much more efficient written in OBJECTIVE CAML than in COQ, they just need to provide a *trace* for COCCINELLE.

[5] Local confluence is not part of COCCINELLE yet.

The arity is not simply an integer, it mentions also whether a symbol is free of arity n, AC or C (of implicit arity 2) since there is a special treatment in the AC/C case.

```
Inductive arity_type : Set :=
  | Free : nat -> arity_type | AC : arity_type | C : arity_type.

Module Type Signature.
  Declare Module Export Symb : decidable_set.S.
  Parameter arity : Symb.A -> arity_type.
End Signature.
```

Up to now, our automatic proof generator does not deal with AC nor C symbols, hence in this work all symbols have an arity `Free n`. However, AC/C symbols are used in other parts of COCCINELLE, in particular the formalisation of *AC matching* [5].

A term algebra is a module defined from its signature `F` and the set of variables `X`.

```
Module Type Term.
  Declare Module Import F : Signature.
  Declare Module Import X : decidable_set.S.
```

Terms are defined as variables or symbols applied to lists of terms (lists are built from two constructors `nil` and `::`, and enjoy the usual `[ x ; y; ...]` notation).

```
Inductive term : Set :=
 | Var : variable -> term | Term : symbol -> list term -> term.
```

This type allows to share terms in a standard representation as well as in a canonical form; but this also implies that terms may be ill-formed w.r.t. the signature. The module contains decidable definitions of well-formedness. However, the rewriting systems we consider do not apply on ill-formed terms, so we will not have to worry about it to prove termination.

The term module type contains other useful definitions and properties that we omit here for the sake of clarity. The COCCINELLE library contains also a *functor* `term.Make` which, given a signature and a set of variables, returns a module of type `Term`. We will not show its definition here.

```
Module Make (F1 : Signature) (X1 : decidable_set.S) : Term.
```

Rewriting systems TRSs provided as a set of rewrite rules are not modelled directly in COCCINELLE. Instead, as explained in the introduction of this Section, we use orderings built from any arbitrary relation `R : relation term` (by definition `relation A` is `A -> A -> Prop`). The usual definition can be retrieved obviously from a list of rewrite rules (i.e. pairs of terms) $\mathcal{R}$ by defining `R` as:

$$\forall s,t \in T(\mathcal{F},X), s \texttt{ -[R]> } t \Longleftrightarrow (s \to t) \in \mathcal{R}$$

The COCCINELLE library provides a module type `RWR` which defines a reduction relation (w.r.t. the "rules" `R`) and its properties.

```
Module Type RWR.
  Declare Module Import T : Term.
```

The first step toward definition of the rewrite relation is the closure by instantiation:

```
Inductive rwr_at_top (R : relation term) : relation term :=
 | instance : ∀t1 t2 sigma, t1 -[R]> t2
  -> (apply_subst sigma t1) -[rwr_at_top R]> (apply_subst sigma t2).
```

Then we define a rewrite step as the closure by context of the previous closure. Notice the use of mutual inductive relations to deal with lists of terms.

```
(** One step at any position. *)
Inductive one_step (R : relation term) : relation term :=
| at_top : ∀t1 t2, t1 -[rwr_at_top R]> t2 -> t1 -[one_step R]> t2
| in_context : ∀f l1 l2, l1 -[one_step_list R]> l2
                  -> (Term f l1 -[one_step R]> Term f l2)
with one_step_list (R : relation term): relation (list term) :=
 | head_step : ∀t1 t2 l, t1 -[one_step R]> t2
              -> (t1 :: l -[one_step_list R]> t2 :: l)
 | tail_step : ∀t l1 l2, l1 -[one_step_list R]> l2
                    -> (t :: l1) -[one_step_list R]> (t :: l2).
```

This module type contains properties declared using the keyword `Parameter`. This means that to build a module of this type, one must prove these properties. For instance it contains the following property stating that if $t_1 \to t_2$ then $t_1\sigma \to^+ t_2\sigma$[6] for any substitution σ.

```
Parameter rwr_apply_subst :
 ∀R t1 t2 sigma, t1 -[rwr R]> t2 ->
   (apply_subst sigma t1 -[rwr R]> apply_subst sigma t2).
```

The library contains a functor `rewriting.Make` building a module of type `RWR` from a module `T` of type `Term`. This functor *builds* in particular *the proof of all properties* required by `RWR`. For an `R` representing the rules of the TRS under consideration, the final theorem we want to generate is:

```
Theorem Well_founded_R: Well_founded (one_step R).
```

To ensure that `one_step R` corresponds to the original TRS $\mathcal{R}$, it *suffices for the user* to perform the easy check that `R` corresponds to the set of rules defining $\mathcal{R}$.

Note that since the datatype `term` represents any Σ-algebra (via application of the functor `Make`), we can say that terms are represented in a deep embedding. However, to simplify proofs, we avoid using substitutions by quantifying on sub-terms as much as possible. That makes our use of the type `term` slightly more shallow on this point.

4 Generation of proof traces

We will illustrate our approach by presenting proof generation techniques at work on a small example in our prototype, namely C*i*ME 2.99. While being based on the C*i*ME 2 tool box, this prototype does not certify all its predecessor's termination power. For instance, modular criteria [25] and termination modulo equational theories are not supported yet. In the following, we restrict to (marked/unmarked) Dependency Pairs [1]

[6] the transitive closure of `one_step` is defined as `rwr` in COCCINELLE.

with/without graphs refinements. The orderings we deal with include strictly the orderings that C*i*ME generates: (non-linear) polynomial interpretations (section 4.6) and RPO with status[7] (section 4.7).

4.1 Global structure of a generated proof

A close look at different termination tools reveals a common underlying methodology which we use as the skeleton of our generated proofs. It consists in deriving recursively from a relation R a set of other relations R_i such that if all R_is are terminating, then so is R. For instance, this structure appears explicitly in the output format of APROVE.

This recursive decomposition is done using *termination criteria*, like DP criteria, (complex) graph criteria, modular criteria, etc. Some tools may use some backtracking but if the procedure succeeds, it means that an implicit tree was built:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\{\dots\langle t_{1,i}, u_{1,i}\rangle\dots\}} <_1 \text{ (poly. interp.)} \quad \cfrac{\cfrac{}{\{\dots\langle t_{2,1,i}, u_{2,1,i}\rangle\dots\}} <_4 \text{ (RPO)} \quad \cfrac{}{\{\dots\langle t_{2,2,i}, u_{2,2,i}\rangle\dots\}} <_3 \text{ (poly. interp.)}}{\{\dots\langle t_{2,i}, u_{2,i}\rangle\dots\}} <}{\text{SUB-GRAPH}}}{\text{GRAPH}}}{R_{dp} = \{\dots\langle t_i, u_i\rangle\dots\}}\ \text{DP}}{R_{init} = \{\dots l_i \to r_i\dots\}}$$

This tree is rooted by the initial problem, namely the initial rewriting system and the first termination criterion used. Each intermediate node is also labelled by a relation and the termination criterion used to decompose the node into its children. Finally each leaf must be labelled by a relation R_i and a well-founded ordering which includes it.

The tree structure is reflected in the generated file. Indeed, for each criterion step (R replaced by sufficient conditions $\{R_i\}$), we will generate a lemma of the form:

```
Lemma wf_R_if_wf_Ri : well_founded R1 -> well_founded R2 ...
 -> well_founded R.
```

The proof of this lemma depends on the termination criterion used (Sec. 4.4 and 4.5).

Each time a leaf is proven using an ordering, we generate a lemma of the form:

```
Lemma wf_Ri : well_founded Ri.
```

The proof is made by induction on the ordering built by the automated tool. Once all leaves have been proven this way, one can easily build the proof of the initial termination property by applying lemmas from leaves to the root:

```
Lemma final: well_founded R.
Proof. apply (wf_R_if_wf_Ri wf_R1 wf_R2 ...). Qed.
```

4.2 The running example

We illustrate our method with two TRSs `R_ack` and `R_add` below in C*i*ME syntax. The first one is the Ackerman function on integers and the second one is the classical addition on binary integers. We will use them separately or together to show how several criteria and orderings may be used in the same proof.

[7] To date, C*i*ME can discover polynomials and LPO with AFS.

```
let F_ack = signature " # : constant; s : unary; ack : binary; ";
let R_ack = TRS F_ack X "ack(s(x), #) -> ack(x, s(#));
ack(#, y) -> s(y);     ack(s(x), s(y)) -> ack(x, ack(s(x),y)); ";

let F_add = signature
    " # : constant ; 0,1 : postfix unary ; + : infix binary ; ";
let R_add = TRS F_add X "
   (#)0 -> #;              # + x -> x;              x + # -> x;
   (x)0 + (y)0 -> (x+y)0;                  (x)0 + (y)1 -> (x+y)1;
   (x)1 + (y)0 -> (x+y)1;          (x)1 + (y)1 -> (x+y+(#)1)0; ";
```

4.3 Generation of the TRS definition

The generation of the Σ-algebra corresponding to a signature in the automated tool is straightforward. We show here the signature corresponding to the Σ-algebra of `F_ack`. Notice the module type constraint `<: Signature` making Coq check that definitions and properties of `SIGMA_F_ack` comply with `Signature` as defined in Section 3.2.

```
Module SIGMA_F_ack <: Signature.
  Inductive symb : Set := | sig_sharp : symb | sig_s : symb ...
  Module Export Symb.
    Definition A := symb.
    Lemma eq_dec : ∀f1 f2 : symb, {f1 = f2} + {f1 <> f2}...
  End Symb.
  Definition arity (f:symb) : arity_type :=
    match f with | sig_sharp => Free 0 | sig_s => Free 1... end.
End SIGMA_F_ack.
```

A module `VARS` for variables is also defined and functors defining the term algebra and rewrite system on it are applied:

```
Module Import TERMS := term.Make(SIGMA)(VARS).
Module Import Rwr := rewriting.Make(TERMS).
```

Now we can define the rewriting system corresponding to `R_ack`:

```
Inductive R : term -> term -> Prop :=
| R0 : ∀V_1 : term,                              (* ack(#,V_1) -> s(V_1) *)
  Term sig_ack [Term sig_sharp nil;V_1] -[R]> Term sig_s [V_1;nil]...
```

Notice that from now on notation `T -[R]> U` denotes that `T` rewrites to `U` in the sense of section 2.1, i.e. there exists two sub-terms t and u at the same position in respectively `T` and `U`, such that `R` u t (see the definition of `one_step` in section 3.2).

4.4 Criterion: Dependency Pairs

The (unmarked) dependency pairs of `R_ack` generated by C*i*ME are the following:

```
< ack(s(x),#) , ack(x,s(#)) >
< ack(s(x),s(y)) , ack(x,ack(s(x),y)) >
< ack(s(x),s(y)) , ack(s(x),y) >
```

An inductive relation representing the *dependency chains* [1] is built automatically. A step of this relation models the (finite) reductions by `R_ack` in the strict subterms of DP instances ($x_0 \to^\star s(V_0), \ldots$) *and* one step of the relevant dependency pair ($\langle \mathrm{ack}(sx, \#), \mathrm{ack}(x, s\#)\rangle$ with $\sigma = \{x \mapsto V_0\}$):

```
Inductive DPR : term -> term -> Prop :=
| DPR_0: ∀x_0 x_1 V_0,                    (* < ack(s(x),#) , ack(x,s(#)) > *)
  x_0 -[one_step R]*> Term sig_s [V_0] ->
  x_1 -[one_step R]*> Term sig_sharp nil ->
  Term sig_ack [x_0 ; x_1] -[DPR]>
   Term sig_ack [V_0 ; Term sig_s [term sig_sharp nil]] ...
```

The main lemma on dependency pairs is the following and fits in the general structure we explained on section 4.1:

```
Lemma wfR_if_wfDPR: well_founded DPR -> well_founded (one_step R).
```

The proof follows a general scheme due to Hubert [18]. It involves several nested inductions instantiating the proof of the criterion in the particular setting of `DPR` and `one_step R`.

Marked symbols A refinement of the DP criterion consists in marking head symbols of dependency pairs's lhs and rhs in order to relax ordering constraints. We simply generate the symbol type with two versions of each symbol and adapt the definition of orderings. The proof strategy needs no change.

4.5 Criterion: Dependency Pairs with graph

Not all DPs can follow another in a dependency chain: one may consider the graph of possible sequences of DPs (*dependency graph*). This graph is not computable, so one uses graphs containing it. We consider here Arts & Giesl's simple approximation [1].

The graph criterion [1] takes benefit from working on the (approximated) graph. In its weak version, it consists in providing *for each strongly connected component* (SCC) an ordering pair that decreases strictly for all its nodes, and weakly for all rules. In its strong version, it considers *cycles*:

Theorem 1 (Arts and Giesl [1]). *A TRS $\mathcal{R}$ is terminating iff for each cycle $\mathcal{P}$ in its dependency graph there is a reduction pair $(\succeq_\mathcal{P}, \succ_\mathcal{P})$ such that: (1) $l \succeq_\mathcal{P} r$ for any $l \to r \in \mathcal{R}$, (2) $s \succeq_\mathcal{P} t$ for any $\langle s, t\rangle \in \mathcal{P}$, and (3) $s \succ_\mathcal{P} t$ for at least one pair in $\mathcal{P}$.*

In practice, our tool uses a procedure due to Middledorp and Hirokawa [16] which splits recursively the graph into sub-components using different orders. The proof uses shallow embedding. One reason for this choice is that a generic theorem for a complex graph criterion is not easy to prove since it involves a substantial part of graph theory (e.g. the notion of cycle). Moreover, verifying the premises of such a theorem amounts to checking that all SCCs found by the prover are really SCCs and that they are terminating, but also to *proving* that it founds *all* SCCs of the graph. That is tedious. On the contrary, using shallow embedding we use these facts *implicitly* by focusing on the termination proof of each component.

Weak version The first thing we generate is the definition of each component as computed by C*i*ME. To illustrate graph criterion on our example we have to take a rewrite system containing rules of `R_ack` and `R_add`. C*i*ME detects two components: we generate the two corresponding inductive relations which are sub-relations of `DPR`.

```
Inductive DPR_sub_0 : term -> term -> Prop :=
| DPR_sub_0_0 : ∀x_0 x_1 V_0,        (* < ack(s(V_0),#) , ack(V_0,s(#)) > *)
  x_0 -[one_step R]*> Term sig_s [V_0] ->
  x_1 -[one_step R]*> Term sig_sharp nil ->
  Term sig_ack [x_0 ; x_1] -[DPR_sub_0]>
    Term sig_ack [V_0 ; Term sig_s [term sig_sharp nil]] ...
Inductive DPR_sub_1 : term -> term -> Prop := ...
```

We formalise the graph criterion by a lemma, fitting the general structure explained in Section 4.1. It is of the form:

```
Lemma wf_DPR_if_wf_sub0_sub1 : well_founded DPR_sub_0 ->
  well_founded DPR_sub_1 -> well_founded DPR.
```

The proof of this kind of lemmas is based on the idea that if we collapse each SCC into one node, they form a DAG. Thus we can reason by cases on the edges of this DAG in a depth-first fashion.

Strong version In addition, when the strong version of the criterion is used, the termination of each sub-component may itself be proven from the termination of smaller components, each one with a different ordering. Due to lack of space, we will not go into the details of this methodology.

It remains to conclude using orderings.

4.6 Orderings: Polynomial interpretations

In our framework a polynomial interpretation is defined as a recursive function on terms. Here is the interpretation as output by C*i*ME for `R_add` dependency pairs:

```
[#]= 0; [0](X0)= X0 + 1; [1](X0)= X0 + 2; [+](X0,X1)= X1 + X0;
```

From this interpretation we produce a measure: `term` $\longrightarrow$ `Z`:

```
Fixpoint measure_DPR (t:term) {struct t} : Z := match t with
| Var _ => 0 | Term sig_sharp nil => 0
| Term sig_0 [x0] => measure_DPR x0 + 1
| Term sig_1 [x0] => measure_DPR x0 + 2
| Term sig_plus [x0; x1] => measure_DPR x1 + measure_DPR x0
end.
```

Notice that despite our term definition is a deep embedding, the measure is defined as if we were in a shallow embedding. Indeed it is defined by a direct recursive function on terms and does not refer to polynomials, substitutions or variables (`x0` above is a Coq variable, it is not a rewriting variable which would be of the form `Var n`). This choice is, once again, because our proofs are simpler to generate this way. In a deep embedding we would need a theory for polynomials, and a generic theorem stating that

a polynomial on positive integers with positive factors is monotonic. But actually this property instantiated on `measure_DPR` above can be proven by a trivial induction on `t`. So again the effort of a deep embedding is not worth it. The final lemma is:

```
Lemma Well_founded_DPR : well_founded DPR.
```

which is equivalent to `∀ x, Acc DPR x`. This is proven firstly by induction on the value of `(measure_DPR x)` then by cases on each DP, finally by applying the induction hypothesis using the fact that each pair is decreasing by `measure_DPR`. One concludes by polynomial comparison. It is well known that the comparison of non linear polynomials on $\mathbb{N}$ is not decidable in general. Since C*i*ME generates non linear polynomials, we have a decision procedure for the particular kind of non linear polynomials it produces.

4.7 Orderings: RPO

The COCCINELLE library formalises RPO in a generic way. Once again we use the module system. RPO is defined using a precedence (a decidable strict ordering `prec` over symbols) and a *status* (multiset/lexicographic) for each symbol.

```
Inductive status_type: Set := Lex:status_type | Mul:status_type.
Module Type Precedence.
  Parameter (A: Set)(prec: relation A)(status: A -> status_type).
  Parameter prec_dec : ∀a1 a2 : A, {prec a1 a2} + {∼prec a1 a2}.
  Parameter prec_antisym : ∀s, prec s s -> False.
  Parameter prec_transitive : transitive A prec.
End Precedence.
```

We define a module type describing what should contain a RPO. First it should be built from a term algebra and a precedence.

```
Module Type RPO.
  Declare Module Import T : term.Term.
  Declare Module Import P : Precedence with Definition A:= T.symbol.
```

The library contains a functor `rpo.Make` building a RPO from two modules of type `Term` and `Precedence`. It also builds among other usual properties of RPO, the proof that if the precedence is well-founded, then so is the RPO. This part of the library is on a deep embedding style, proofs of termination using RPOs are very easy to generate as it is sufficient to generate the precedence, the proof that it is well-founded and apply the functor `rpo.Make`.

Here is the generated definition of the RPO used for proving termination of dependency chains of `R_ack`.

```
Module precedence <: Precedence.
  Definition A : Set := symb.
  Definition prec (a b:symb) : Prop :=
    match a,b with | sig_s,sig_ack => True | _,_ => False end.
  Definition status: symb -> status_type:= fun x => Lex.
  Lemma prec_dec: ∀a1 a2: symb, {prec a1 a2}+{∼ prec a1 a2}. ...
  Lemma prec_antisym: ∀s, prec s s -> False. ...
  Lemma prec_transitive: transitive symb prec. ...
End precedence.
```

Argument filtering systems The use of Dependency Pairs allows a wide choice of orderings by dropping the condition of strict monotoncity. Regarding path orderings, this can be achieved using argument filtering systems (AFS) [1]. We define AFSs as fixpoints and apply them at comparison time. This does not affect the (COQ) proof scheme.

5 Results and benchmarks

5.1 The user point of view

Our C*i*ME prototype is publicly available from the A3PAT project website[8]. To illustrate the trace production we consider the Ackerman function as defined in section 4.2. The system being defined we have to choose the termination criterion and the orderings. Here, we may select (marked) Dependency Pairs without graphs refinement and RPO with AFSs, then ask C*i*ME to check termination of the defined TRS.

```
CiME> termcrit "dp"; termcrit "marks";termcrit "nograph";
CiME> polyinterpkind {("rpo",1)}; termination R;
```

Since the system is found to be terminating, C*i*ME displays a human readable sketch of the termination proof. We may now ask for its COQ trace:

```
CiME> coq_certify_proof "example.v" R;
```

This produces a COQ scripts `example.v` which ends with a proof of the lemma:

```
Lemma well_founded_R : well_founded R.
```

Now, if as COQ user we want to prove the termination of a given relation, we first have to translate this relation into C*i*ME then to process as explained above, to compile the file, and eventually to `Require` the generated COQ module. Note that the relation should be syntactically equal to the one produced by C*i*ME.

5.2 Results

We used the *Termination Problems Data Base*[9] V3.2 as challenge. Until now we have produced a COQ certificate for 100% of the 358 TRS that C*i*ME proves terminating without using modular technique or AC termination[10]. We will now give some details on our experiments. We give below the average and maximal sizes of compiled COQ proofs, as well as the *maximal* compilation time (on a 2GHz, 1GB machine, running Linux). We omit average compilation time: most of the files compile in less than 1s.

	with graph	without graph
RPO	2.76MB / 4.7MB / 54s	2.6MB / 4.7MB / 68s
Interpretations	0.2MB/ 3.3MB / 23.25s	0.2MB/ 24.7MB / 544.79s
RPO+Interpretations	2.8MB / 9.4MB / 54.3s	1.4MB / 6.2MB / 41.57s

[8] `http://www3.ensiie.fr/~urbain/a3pat/pub/index.en.html`

[9] `http://www.lri.fr/~marche/tpdb`

[10] Remark that not all the systems of the database are terminating, and that some of them are proven by C*i*ME 2 using modular techniques or AC termination for which our prototype does not produce certificate yet.

Note that criteria do not affect sizes with RPO. Regarding polynomials, one interesting remark is that the small overhead (in the script) due to involved criteria may results in simpler interpretations and thus in dramatically smaller compiled proofs.

6 Related works and conclusion

There are several works to be mentioned w.r.t. the communication between automated provers and COQ. Amongst them, the theorem-prover ZÉNON [12], based on tableaux, produces COQ proof terms as certificates. ELAN enjoys techniques to produce COQ certificates for rewriting [22]. Bezem describes an approach regarding resolution [3]. However, these systems do not tackle the problem of termination proofs.

To our knowledge the only other approach to generate termination certificates for rewriting systems relies on the CoLoR/Rainbow libraries [4]. In this approach, term algebras and TRSs are handled via an embedding even deeper than in COCCINELLE, since a TRS is given by a set of pairs of terms. Notice that the RPO in CoLoR is weaker than the usual one, since the underlying equivalence used to compare terms is not $\leq_{RPO} \cap \geq_{RPO}$, but the syntactic equality: this means that CoLoR's RPO contains less pairs of terms than the usual RPO (as in COCCINELLE for instance). Further note that this RPO is not used in Rainbow for termination certificates: only polynomial interpretations are used as indicated on the Rainbow web page. There are currently 167 out of 864 termination problems in TPDB proven by TPA [19] and certified by CoLoR/Rainbow.

We presented a methodology to make automated termination tools generate *traces* in a proof assistant format. We illustrate this approach using our C*i*ME 2.99 prototype which generates COQ traces. The performances of the prototype on the examples of the TPDB database are promising. Our approach is easy to extend, in particular because extensions may be done in deep or shallow embedding.

To apply this methodology on different tools and targeted proof assistants, one needs a *termination trace language*. An ongoing work in the A3PAT group is to define a more general language that can even tackle proofs of various rewriting properties such as termination, confluence (which needs termination), equational proofs [6], etc. We think that a good candidate could be based on the tree structure we explained on Section 4.1.

One particularly interesting follow-up of this work is the possibility to plug automated termination tools *as external termination tactics* for proof assistants. Indeed termination is a key property of many algorithms to be proven in proof assistants. Moreover, in type theory based proof assistants like COQ, one cannot define a function without simultaneously proving its termination. This would allow to define functions whose termination is not obvious without the great proof effort it currently needs.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
3. M. Bezem, D. Hendriks, and H. de Nivelle. Automated proof construction in type theory using resolution. *J. Autom. Reasoning*, 29(3-4):253–275, 2002.

4. F. Blanqui, S. Coupet-Grimal, W. Delobel, S. Hinderer, and A. Koprowski. Color, a coq library on rewriting and termination. In Geser and Sondergaard [14].
5. E. Contejean. A certified AC matching algorithm. In V. van Oostrom, editor, *15th International Conference on Rewriting Techniques and Applications*, volume 3091 of *Lecture Notes in Computer Science*, pages 70–84, Aachen, Germany, June 2004. Springer-Verlag.
6. E. Contejean and P. Corbineau. Reflecting proofs in first-order logic with equality. In *20th International Conference on Automated Deduction (CADE-20)*, number 3632 in Lecture Notes in Artificial Intelligence, pages 7–22, Tallinn, Estonia, July 2005. Springer-Verlag.
7. E. Contejean, C. Marché, B. Monate, and X. Urbain. Proving termination of rewriting with CiME. In A. Rubio, editor, *Extended Abstracts of the 6th International Workshop on Termination, WST'03*, pages 71–73, June 2003. `http://cime.lri.fr`.
8. T. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.
9. N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, Mar. 1982.
10. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1):69–115, Feb. 1987.
11. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. North-Holland, 1990.
12. D. Doligez. Zenon. `http://focal.inria.fr/zenon/`.
13. J. Endrullis. Jambox. `http://joerg.endrullis.de/index.html`.
14. A. Geser and H. Sondergaard, editors. *Extended Abstracts of the 8th International Workshop on Termination, WST'06*, Aug. 2006.
15. J. Giesl, P. Schneider-Kamp, and R. Thiemann. Aprove 1.2: Automatic termination proofs in the dependency pair framework. In U. Furbach and N. Shankar, editors, *Third International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, Seattle, USA, Aug. 2006. Springer-Verlag.
16. N. Hirokawa and A. Middeldorp. Automating the dependency pair method. In F. Baader, editor, *19th International Conference on Automated Deduction (CADE-19)*, volume 2741 of *Lecture Notes in Computer Science*, pages 32–46, Miami Beach, FL, USA, July 2003. Springer-Verlag.
17. N. Hirokawa and A. Middeldorp. Tyrolean termination tool. In J. Giesl, editor, *16th International Conference on Rewriting Techniques and Applications (RTA'05)*, volume 3467 of *Lecture Notes in Computer Science*, pages 175–184, Nara, Japan, Apr. 2005. Springer-Verlag.
18. T. Hubert. Certification des preuves de terminaison en Coq. Rapport de DEA, Université Paris 7, Sept. 2004. In French.
19. A. Koprowski. TPA. `http://www.win.tue.nl/tpa`.
20. D. S. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Mathematics Department, Louisiana Tech. Univ., 1979. Available at `http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html`.
21. C. Marché and H. Zantema. The termination competition 2006. In Geser and Sondergaard [14]. `http://www.lri.fr/~marche/termination-competition/`.
22. Q. H. Nguyen, C. Kirchner, and H. Kirchner. External rewriting for skeptical proof assistants. *J. Autom. Reasoning*, 29(3-4):309–336, 2002.
23. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
24. The Coq Development Team. *The Coq Proof Assistant Documentation – Version V8.1*, Feb. 2007. `http://coq.inria.fr`.

25. X. Urbain. Modular and incremental automated termination proofs. *Journal of Automated Reasoning*, 32:315–355, 2004.

A RPO definition

The RPO makes use of an equivalence generated by symbols having multi-set status.

```
Inductive equiv : term -> term -> Prop :=
  | Eq : ∀t, equiv t t
  | Eq_lex : ∀f l1 l2, status f = Lex -> equiv_list_lex l1 l2
           -> equiv (Term f l1) (Term f l2)
  | Eq_mul : ∀f l1 l2, status f = Mul -> permut equiv l1 l2
           -> equiv (Term f l1) (Term f l2)
with equiv_list_lex : list term -> list term -> Prop :=
   | Eq_list_nil : equiv_list_lex nil nil
   | Eq_list_cons : ∀t1 t2 l1 l2, equiv t1 t2 -> equiv_list_lex l1 l2
            -> equiv_list_lex (t1 :: l1) (t2 :: l2).
```

It also makes use of the notion of permutations modulo the equivalence defined above. Permutations are defined in another module.

```
Declare Module Import LP : list_permut.S
  with Definition EDS.A:=term with Definition EDS.eq_A := equiv.
```

Then RPO is defined as the following mutual inductive relation :

```
Inductive rpo : term -> term -> Prop :=
  | Subterm : ∀f l t s, mem s l -> rpo_eq t s -> rpo t (Term f l)
  | Top_gt : ∀f g l l', prec g f
           -> (∀ s', mem s' l' -> rpo s' (Term f l))
           -> rpo (Term g l') (Term f l)
  | Top_eq_lex : ∀f l l', status f = Lex -> rpo_lex l' l
              -> (∀ s', mem s' l' -> rpo s' (Term f l))
              -> rpo (Term f l') (Term f l)
  | Top_eq_mul : ∀f l l', status f = Mul -> rpo_mul l' l
              -> rpo (Term f l') (Term f l)
with rpo_eq : term -> term -> Prop :=
  | Equiv : ∀t t', equiv t t' -> rpo_eq t t'
  | Lt : ∀s t, rpo s t -> rpo_eq s t
with rpo_lex : list term -> list term -> Prop :=
  | List_gt : ∀s t l l', rpo s t -> length l = length l'
             -> rpo_lex (s :: l) (t :: l')
  | List_eq : ∀s s' l l', equiv s s' -> rpo_lex l l'
             -> rpo_lex (s :: l) (s' :: l')
with rpo_mul : list term -> list term -> Prop :=
  | List_mul : ∀a lg ls lc l l',
      permut l' (ls ++ lc) -> permut l (a :: lg ++ lc) ->
      (∀ b, mem b ls -> ∃a', mem a' (a :: lg) /\ rpo b a') ->
      rpo_mul l' l.
```

The module also contains generic proofs of the fact that

- `equiv` is an equivalence relation,
- `rpo_eq` is an ordering over the equivalence classes of `equiv`,
- `rpo` is strict part of `rpo_eq`,

- `equiv`, `rpo` and `rpo_eq` monotonic and stable under substitution,
- if the precedence `prec` is well-founded, then so is `rpo`.