Certification of Automated Termination Proofs*

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

[1] LRI, Université Paris-Sud, CNRS, INRIA Futurs, Orsay F-91405
[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

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 step by step, and fully automated tools which make use of (intricate) decision procedures.

Reducing the cost of formal proofs 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 answer of automated tools. Yet, since application fields include possibly critical sectors as security, code verification, cryptographic protocols, etc., *reliance on verification tools is crucial*.

Some proof assistants, like COQ [28], need to check mechanically the proof of each notion used. Among 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.

For instance, COQ or ISABELLE/HOL [26] have a small and highly reliable *kernel*. In COQ, the kernel type-checks a *proof term* to ensure the soundness of a proof. Certified-programming environments based on these proof assistants find here an additional guarantee. Yet, among the weaknesses of these assistants, one may regret the lack of automation in the proof discovery process. Automation is indeed difficult to obtain in this framework: the proof assistant has to check a property proven by an external

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

B. Konev and F. Wolter (Eds.): FroCos 2007, LNAI 4720, pp. 148–162, 2007.

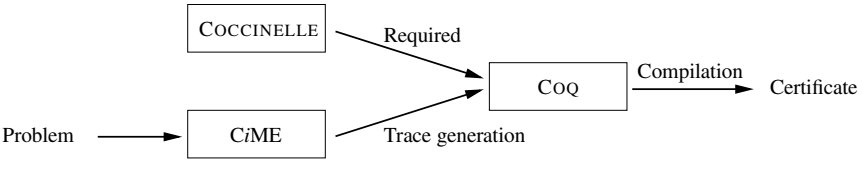
procedure before accepting it. Therefore, such a procedure has to return a *proof trace* checkable by the assistant.

We want to meet the important need of *proofs delegation* for some properties in the framework of rewriting techniques. We will focus on generic ways to provide reasonably-sized proof traces for complex properties, for instance *termination*.

Termination is the property of a program any execution 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 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.

The last decade has been very fertile w.r.t. automation of termination proofs, and yielded many efficient tools (APROVE [17], C*i*ME [8], JAMBOX [15], TPA [22], TTT [20] and others) referenced on the website of the Termination Competition [24]. Some of them 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 that do not. In the sequel, we aim at bridging this gap.

We present here a methodology for the particularly important challenge of automatically generating proof traces in the domain of *first order term rewrite systems* and in particular for termination proofs of such systems. We do not restrict to classical approaches of all-shallow embedding or, like Color [4], of all-deep embedding to model properties or techniques. Instead we use a mixed approach so as to get the best of both worlds. We implemented our principles and methodology within the rewrite tool box C*i*ME 2.99. This version uses parts of the termination engine of C*i*ME2.04; with our mixed approach it can certify (with COQ) termination proofs of more than 370 problems (i.e. approximately 34.5% of the TPDB 3.2 directory TRS excluding termination modulo equational theories), and using involved criteria. This is made possible thanks to a COQ library for rewriting (COCCINELLE) developed by E. Contejean in our project.



We make our notations precise and give some prerequisites about first order term rewriting and about the COQ proof assistant in Section 2. Then, in Section 3, we present our modelling of termination of rewriting in COQ, which mixes deep and shallow embeddings in order to take benefits of both. We briefly present the COQ library COCCINELLE developed in the project to that purpose. In section 4 we present the certification of proofs using involved criteria such as Dependency Pairs [1] with graphs refinement, mixing orderings based on polynomial interpretations [23] or RPO [10] with AFS [1]. We shall adopt the end-user point of view and provide some experimental

results of C*i*ME 2.99 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 [13, 2] and termination, in particular with the Dependency Pairs (DP) approach [1]. We recall usual notions, and give 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 subterm of t at position p and $t[u]_p$ for term t where $t|_p$ has been replaced by u. *Substitutions* are mappings from variables to terms and $t\sigma$ denotes the application of a substitution σ to a term t.

A *term rewriting system* (TRS for short) over a signature $\mathcal{F}$ is a set R of *rewrite rules* $l \to r$ with $l, r \in T(\mathcal{F}, X)$. A TRS R defines a monotonic relation $\to_R$ closed under substitution (aka a *rewrite relation*) in the following way: $s \to_R t$ (s *reduces to* t) if there is a position p such that $s|_p = l\sigma$ and $t = s[r\sigma]_p$ for a rule $l \to r \in R$ and a substitution σ. In the following, we shall omit systems and positions that are clear from the context. We denote the reflexive-transitive closure of a relation $\to$ by $\to^\star$. 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*.

A term is *R-strongly normalizing* (R-SN) if it cannot reduce infinitely many times for the relation defined by System R[1]. A rewrite relation *terminates* if any term is SN. Termination is usually proven with the help of *reduction orderings* [11] or *ordering pairs* with *dependency pairs*. The set of *unmarked* dependency pairs[2] of a TRS R, denoted $DP(R)$ is defined as $\{\langle u, v\rangle$ such that $u \to t \in R$ and $t|_p = v$ and $\Lambda(v)$ is defined$\}$. An *ordering pair* is a pair $(\succeq, >)$ of relations over $T(\mathcal{F}, X)$ such that: 1) $\succeq$ is a stable and monotonic quasi-ordering, i.e. reflexive and transitive, 2) $>$ is a stable strict ordering, i.e. irreflexive and transitive, and 3) $> \cdot \succeq \; = \; >$ or $\succeq \cdot > \; = \; >$. An ordering pair is *well-founded* if there is no infinite strictly decreasing sequence $t_1 > t_2 > \ldots$

2.2 The Coq Proof Assistant

The Coq proof assistant is based on *type theory* and features: 1) 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) [9]. λ-abstraction is denoted `fun x:T => t`, and application is denoted `t u`. 2) 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 $\to$ is right associative): the identity function

[1] When R is clear from the context, we shall write SN.

[2] For readability's sake we detail only unmarked DP, see Sec. 4.4 for how we deal with marks.

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 of natural numbers, O and S (successor) being the two constructors[3], and the property even of being an even natural number.

```
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 (S (S O)) (even_S O (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 on 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.
```

This inductive definition contains both the basis case (that is when an element has no predecessor w.r.t. the relation R) and the general inductive case. For example, in a relation R on bool defined by R true false, true is accessible because it has no predecessor, and so is false because its only predecessor is true. Hence Acc R true and Acc R false are provable, hence well-founded R is provable.

The usual ordering $<$ is not well-founded over the integers, there are infinite descending chains as for example $\ldots - (n+1) < -n < -(n-1) < \ldots < -1 < 0$. However, it is possible to reason by well-founded induction over the integer using a well-founded relation $<_{wf}$ defined by $x <_{wf} y$ if $|x| < |y|$ or ($|x| = |y|$ and $x > 0 > y$).
The relation is of the form: $0 <_{wf} 1 <_{wf} -1 <_{wf} 2 <_{wf} -2 <_{wf} \ldots$

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 $t \to^* u$) instead.

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

```
Theorem well_founded_R: well_founded R.
```

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

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* and *deep embedding*. When using shallow embedding, one defines *ad hoc translations* for the different notions, and proves criteria on the translation of each considered system. For instance TRSs will be inductive definitions with one constructor per rule.

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 usually leads (not always as we explain below) to simpler 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 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 sometimes deep embedding not worth the efforts it requires: some premises of generic lemmas, which have to be proven on each considered problem, are as hard (if not harder) to prove than the shallow lemmas themselves. 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 [6]. To start with, it 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 proofs of properties of these notions, for example that RPO is well-founded whenever the underlying precedence is.

Moreover Coccinelle is intended 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].

[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 when written in Objective Caml than in Coq, they just need to provide a *trace* for Coccinelle.

Translating functions and proving their full correctness obviously provide a certification of the underlying algorithm. Note that 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.

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.
```

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

Rewriting systems. TRSs provided as sets 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:

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

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 $\mathcal{R}$.

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

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 subterms 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*ME2 tool box, this prototype does not certify all its predecessor's termination power. For instance, modular criteria [29] and termination modulo equational theories are not supported yet. In the following, we restrict to (marked/unmarked) Dependency Pairs [1] 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 $R_i s$ are terminating, then so is R. For instance, this structure appears explicitly with the *processors* [18] 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:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\{\dots\langle t_{1,i}, u_{1,i}\rangle\dots\}}\,<_1 \text{ (poly. interp.)} \quad < \quad \cfrac{}{\{\dots\langle t_{2,1,i}, u_{2,1,i}\rangle\dots\}}\,<_2 \text{ (RPO)} \quad \cfrac{}{\{\dots\langle t_{2,2,i}, u_{2,2,i}\rangle\dots\}}\,<_3 \text{ (poly. interp.)}}{\{\dots\langle t_{2,i}, u_{2,i}\rangle\dots\}}\,<}{\{\dots\langle t_i, u_i\rangle\dots\}}\,\textsc{Sub-Graph}}{R_{dp} = \{\dots\langle t_i, u_i\rangle\dots\}}\,\textsc{Graph}}{R_{init} = \{\dots l_i \to r_i\dots\}}\,\text{DP}$$

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.
```

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

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 a very simple TRS $R = R_{\text{ack}} \cup R_{\text{add}}$ (over a signature $\mathcal{F}$) where R_{ack} computes the Ackerman function on Peano integers, and R_{add} computes addition on binary integers. *Digits* are denoted as postfix operators $(\_)0$ and $(\_)1$, whereas $\#$ is the constant 0 seen as a *number*, shared between binary and Peano integers.

$$R \begin{cases} R_{\text{ack}} \begin{cases} \text{ack}(\#, y) \to s(y) & \text{ack}(s(x), \#) \to \text{ack}(x, s(\#)) \\ \text{ack}(s(x), s(y)) \to \text{ack}(x, \text{ack}(s(x), y)) \end{cases} \\ R_{\text{add}} \begin{cases} (\#)0 \to \# & \# + x \to x & x + \# \to x \\ (x)0 + (y)0 \to (x + y)0 & (x)1 + (y)0 \to (x + y)1 \\ (x)0 + (y)1 \to (x + y)1 & (x)1 + (y)1 \to ((x + y) + (\#)1)0 \end{cases} \end{cases}$$

4.3 Generation of the TRS Definition

For sake of clarity we will use Coq notations that are different than in previous sections: `Term X (Y::Z::...::nil)` will now be denoted by `X(Y,Z...)`.

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 $\mathcal{F}$. Notice the module type constraint `<: Signature` making Coq check that definitions and properties of `SIGMA_F` comply with `Signature` as defined in Section 3.2.

```
Module SIGMA_F <: Signature.
  Inductive symb : Set := | # : symb | 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 | # => Free 0 | s => Free 1... end.
End SIGMA_F.
```

We define a module `VARS` for variables, apply functors building the term algebra and rewrite system on it, and then the rewriting system corresponding to R:

```
Module Import TERMS := term.Make(SIGMA)(VARS).
Module Import Rwr := rewriting.Make(TERMS).
Inductive R_rules : term → term → Prop :=
| R0 : ∀V1 : term, ack(#,V1) -[R_rules]> s(V1)...
Definition R := Rwr.one_step R_rules.
```

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 subterms 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 generated by C*i*ME are the following:

$\langle \mathrm{ack}(s(x),\#), \mathrm{ack}(x,s(\#))\rangle$
$\langle \mathrm{ack}(s(x),s(y)), \mathrm{ack}(x,\mathrm{ack}(s(x),y))\rangle \quad \langle \mathrm{ack}(s(x),s(y)), \mathrm{ack}(s(x),y)\rangle$
$\langle (x)1+(y)0, x+y\rangle \quad \langle (x)0+(y)1, x+y\rangle \quad \langle (x)0+(y)0, x+y\rangle \quad \langle (x)0+(y)0, (x+y)0\rangle$
$\langle (x)1+(y)1, x+y\rangle \quad \langle (x)1+(y)1, (x+y)+(\#)1\rangle \quad \langle (x)1+(y)1, ((x+y)+(\#)1)0\rangle$

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

```
Inductive DPR : term → term → Prop :=
| DPR0: ∀x0 x1 V0, x0 -[R]*> s(V0) → x1 -[R]*> #
          → ack(x0,x1) -[DPR]> ack(V0,s(#))
```

The main lemma on DPs fits in the general structure we explained on Section 4.1:

```
Lemma wfR_if_wfDPR: well_founded DPR → well_founded R.
```

The proof follows a general scheme due to Hubert [21]. It involves several nested inductions instantiating the proof of the criterion in the particular setting of `DPR` and `R`.

Note that we can also prove this lemma in the case of an enhancement of DPs by Dershowitz [12] consisting in discarding DPs whose rhs is a subterm of the lhs[8].

Marked symbols. A refinement of the DP criterion consists in marking head symbols in lhs and rhs of dependency pairs 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 one 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 [19] 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

[8] Such DPs cannot occur in minimal chains. Thus they can be discarded.

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 found *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 the graph criterion on our example we may take the whole system R. C*i*ME detects two components ($\mathtt{sub}_0$ with some DPs of R_{add}, $\mathtt{sub}_1$ with some DPs of R_{ack}): we generate the two corresponding sub-relations of DPR.

```
Inductive DPR_sub0 : term → term → Prop :=
| DPR_sub00: ∀x0 x1 V0, x0 -[R]*> s(V0) → x1 -[R]*> #
→ ack(x0,x1)-[DPR_sub0]> ack(V0,s(#)) (*<ack(s(V0),#) , ack(V0,s(#))>*)...
Inductive DPR_sub1 : term → term → Prop := ...
```

The following lemma states the criterion and fits the general structure in Section 4.1.

```
Lemma wf_DPR_if_wf_sub0_sub1 : well_founded DPR_sub0 →
  well_founded DPR_sub1 → well_founded DPR.
```

The proof of these lemmas uses the idea that if we collapse each SCC into one node, they form a DAG on which we can reason by cases on the edges 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 by providing well-suited ordering pairs.

4.6 Orderings: Polynomial Interpretations

In our framework a polynomial interpretation is defined as a recursive function on terms. C*i*ME outputs an interpretation for the SCC $\mathtt{sub}_0$ (other symbols are mapped to 0):

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

From this interpretation we produce a measure: term → Z:

```
Fixpoint measure_DPR_sub0 (t:term) {struct t} : Z :=
match t with
| Var _ => 0 | # => 0
| 0(x0) => measure_DPR x0 + 1 | 1(x0) => measure_DPR x0 + 1
| plus (x0, x1) => measure_DPR x1 + measure_DPR x0 | _ => 0
end.
```

Notice that although our term definition is a deep embedding, the measure is defined as if we were in a shallow embedding[9]. Indeed it is defined by a direct recursive function on terms and does not refer to polynomials, substitutions or variables ($\mathtt{x}_0$ above

[9] In particular it is completely handled by the trace generation part of C*i*ME since our library COCCINELLE focuses on deep embedding.

is a COQ variable, it is not a rewriting variable which would be of the form `Var n`). This choice makes, once again, our proofs simpler to generate. 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 $\texttt{measure\_DPR\_sub}_0$ above can be proven by a trivial induction on `t`. So again the effort of a deep embedding is not worth this effort. The following lemma proves the well-foundedness of $\texttt{measure\_DPR\_sub}_0$:

```
Lemma Well_founded_DPR_sub0 : well_founded DPR_sub0.
```

which is equivalent to $\forall$ `x, Acc DPR x`. This is proven firstly by induction on the value of $\texttt{(measure\_DPR\_sub}_0\texttt{ x)}$, then by cases on each DP of $\texttt{DPR\_sub}_0$, finally by applying the induction hypothesis using the fact that each pair is decreases w.r.t. $\texttt{measure\_DPR\_sub}_0$. One concludes by polynomial comparison. It is well known that the comparison of non-linear polynomials on $\mathbb{N}$ is not decidable in general. We have a decision procedure for the particular kind of non linear polynomials C*i*ME produces.

4.7 Orderings: RPO

The COCCINELLE library formalises RPO in a generic way, and proves it to be well-suited for ordering pairs. 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.
```

A module type for an RPO 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 an 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 in 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 to apply the functor `rpo.Make`. It should be noticed that the fact that the generic RPO uses a strict precedence and a comparison from left to right in the lexicographic case is not a restriction in practice: a simple translation from terms to terms mapping equivalent symbols onto the same symbol, and performing the wanted permutation over the subterms under a given lexicographic symbol is both monotonic and stable. Hence the relation defined by comparing the translations of terms by the generic RPO still has the desired properties.

The generated definition of the RPO used for proving well-foundedness of $\mathtt{sub}_1$ is:

```
Module precedence <: Precedence.
  Definition A : Set := symb.
  Definition prec (a b:symb) : Prop :=
    match a,b with | s,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.
```

And as previously: `Lemma Well_founded_DPR_sub1 : well_founded DPR_sub1.`

Argument filtering systems The use of Dependency Pairs allows a wide choice of orderings by dropping the condition of strict monotonicity. 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

C*i*ME 2.99 can be downloaded and tested from the A3PAT website[10]. Once the system is defined, we have to choose the termination criterion and the orderings. For instance, we may select DP with graphs refinement and both linear polynomials (bound 2) and RPO with AFSs, then ask C*i*ME to check termination and generate the proof trace:

```
CiME> termcrit "dp"; termcrit "nomarks"; termcrit "graph";
CiME> polyinterpkind {("linear",2);("rpo",1)}; termination R;
CiME> coq_certify_proof "example.v" R;
```

We used the *Termination Problems Data Base*[11] v3.2 as challenge. Until now we have produced a Coq certificate for 374 TRS that C*i*ME proves terminating without using modular technique or AC termination[12]; this number rises over 545 on TPDB v4.0. We will now give some details on our experiments. We give below, depending on the use of graphs, the average and max. sizes of compiled Coq proofs, as well as the *average* compilation time (together with the number of problem solved) using marks on a 2GHz, 1GB machine, running Linux. RPO + Pol. means that selected orderings for proof search are RPOs and polynomials.

	with graph				without graph			
	s. av.	s. max	t. av	(nb)	s. av.	s. max	t. av	(nb)
RPO	3.6MB	9.09MB	9.9s	(228)	3.64MB	5.31MB	7.8s	(196)
Linear Pol.	0.72MB	12.27MB	7.8s	(295)	0.42MB	4.89MB	2.5s	(225)
Simple Pol.	0.84MB	12.40MB	10.1s	(314)	0.45MB	1.26MB	6.3s	(264)
RPO + Pol.	1.20MB	12.30MB	10.6s	(374)	0.69MB	4.91MB	8.7s	(300)

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

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

[12] Not all the systems of the TPDB are terminating. Some are proven by the *full* termination engine of C*i*ME 2.04 using techniques for which C*i*ME 2.99 does not produce a certificate yet.

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 [14], based on tableaux, produces Coq proof terms as certificates. Elan enjoys techniques to produce Coq certificates for rewriting [25]. 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. Rainbow is a relatively efficient tool thanks to orderings built with *matrix interpretations* (which we don't handle yet). But it does not handle the following techniques: *enhanced* or *marked* dependency pairs, *complex graphs*, RPO with AFS. We think that adding these techniques to CoLoR/Rainbow will be hard, due to the pure deep embedding approach. There are currently 167 out of 864 termination problems in TPDB (v3.2) proven by TPA [22] and certified by CoLoR/Rainbow using polynomial interpretations and the webpage mentions 237 problems certified using matrix interpretations.

We presented a methodology to make automated termination tools generate *traces* in a proof assistant format. The approach is validated by a prototype generating 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 [7], 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. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theoretical Computer Science 236, 133–178 (2000)
2. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
3. Bezem, M., Hendriks, D., de Nivelle, H.: Automated proof construction in type theory using resolution. J. Autom. Reasoning 29(3-4), 253–275 (2002)
4. Blanqui, F., Coupet-Grimal, S., Delobel, W., Hinderer, S., Koprowski, A.: Color, a coq library on rewriting and termination. In: Geser and Sondergaard [16]
5. Contejean, E.: A certified AC matching algorithm. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 70–84. Springer, Heidelberg (2004)

6. Contejean, E.: Coccinelle (2005), http://www.lri.fr/~contejea/Coccinelle/coccinelle.html
7. Contejean, E., Corbineau, P.: Reflecting proofs in first-order logic with equality. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 7–22. Springer, Heidelberg (2005)
8. Contejean, E., Marché, C., Monate, B., Urbain, X.: Proving termination of rewriting with cime. In: Rubio [27], pp. 71–73, http://cime.lri.fr
9. Coquand, T., Paulin-Mohring, C.: Inductively defined types. In: Martin-Löf, P., Mints, G. (eds.) COLOG-88. LNCS, vol. 417, Springer, Heidelberg (1990)
10. Dershowitz, N.: Orderings for term rewriting systems. Theoretical Computer Science 17(3), 279–301 (1982)
11. Dershowitz, N.: Termination of rewriting. Journal of Symbolic Computation 3(1), 69–115 (1987)
12. Dershowitz, N.: Termination Dependencies. In: Rubio [27] Technical Report DSIC II/15/03, Univ. Politécnica de Valencia, Spain
13. Dershowitz, N., Jouannaud, J.-P.: Rewrite systems. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 243–320. North-Holland, Amsterdam (1990)
14. Doligez, D.: Zenon. http://focal.inria.fr/zenon/
15. Endrullis, J.: Jambox, http://joerg.endrullis.de/index.html.
16. Geser, A., Sondergaard, H. (eds.).: Extended Abstracts of the 8th International Workshop on Termination, WST'06 (August 2006)
17. Giesl, J., Schneider-Kamp, P., Thiemann, R.: Aprove 1.2: Automatic termination proofs in the dependency pair framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, Springer, Heidelberg (2006)
18. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and Improving Dependency Pairs. Journal of Automated Reasoning 37(3), 155–203 (2006)
19. Hirokawa, N., Middeldorp, A.: Automating the dependency pair method. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 32–46. Springer, Heidelberg (2003)
20. Hirokawa, N., Middeldorp, A.: Tyrolean termination tool. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 175–184. Springer, Heidelberg (2005)
21. Hubert, T.: Certification des preuves de terminaison en Coq. Rapport de DEA, Université Paris 7, In French (September 2004)
22. Koprowski, A.: TPA, http://www.win.tue.nl/tpa
23. Lankford, D.S.: 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
24. Marché, C., Zantema, H.: The termination competition 2006. In Geser and Sondergaard [16], http://www.lri.fr/~marche/termination-competition/
25. Nguyen, Q.H., Kirchner, C., Kirchner, H.: External rewriting for skeptical proof assistants. J. Autom. Reasoning 29(3-4), 309–336 (2002)
26. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. In: Nipkow, T., Paulson, L.C., Wenzel, M. (eds.) Isabelle/HOL. LNCS, vol. 2283, Springer, Heidelberg (2002)
27. Rubio, A., (ed.).: Extended Abstracts of the 6th International Workshop on Termination, WST'03, Technical Report DSIC II/15/03, Univ. Politécnica de Valencia, Spain (June 2003)
28. The Coq Development Team. The Coq Proof Assistant Documentation – Version V8.1, (February 2007), http://coq.inria.fr.
29. Urbain, X.: Modular and incremental automated termination proofs. Journal of Automated Reasoning 32, 315–355 (2004)