Termination and normalisation under strategies–Proofs in ELAN

Hélène Kirchner, Isabelle Gnaedig

URL: http://www.elsevier.nl/locate/entcs/volume36.html 28 pages

Termination and normalisation under strategy Proofs in ELAN

Hélène Kirchner and Isabelle Gnaedig

LORIA-CNRS and INRIA-Lorraine
BP 239
54506 Vandœuvre-lès-Nancy Cedex, France
Email: Helene.Kirchner@loria.fr,gnaedig@loria.fr

Abstract

In this paper, we review several methods for proving termination of rewrite programs or answer questions about normal forms. We concentrate on the methods that have been designed in ELAN and which are useful for studying termination and normal forms of ELAN programs. We address several problems: general termination of a rewrite program, innermost termination on sets of ground terms, computation or approximation of the set of normal forms, modular termination of the union of rewrite systems, estimation of the number of normal forms for a non-deterministic rewrite program with strategies. For each point we briefly present one method, explain its scope and give references to other methods concerning the same problem, before describing the implementation in ELAN. We conclude with some open questions.

1 Introduction

The general context of the study of termination and normalisation problems presented here is given by the rule-based programming language ELAN [BKK+96,BKK+98], which provides an environment for specifying and prototyping deduction systems in a language based on rewrite rules controlled by strategies. ELAN is a framework for designing proof tools, supports the design of theorem provers, logic programming languages, constraint solvers and decision procedures, and offers a modular framework for studying their combination. So it is not surprising that most of the methods presented in this paper for proving termination of rewrite programs or answer questions about normal forms have been developed in ELAN itself. The reflection power of rewriting logic [CM96] gives the potentiality of applying them to ELAN programs. However the methods and techniques presented here go beyond ELAN and are applicable to rule-based languages such as ASF+SDF [Kli93], Cafe-OBJ [FN97] or Maude [CELM96]. The ELAN system is presented in Section 2 that also introduces the preliminary notions and notations on rewrite systems and termination.

In rule-based programming languages, a program is a set of rules, a query is an input expression, and computations are reduction of expressions to their normal forms. The result may be unique or not, and some computations may not terminate. In some cases, for instance for functional evaluation, termination is quite important since at least one result is expected. Termination is also important for checking that a computation has a unique result, via the local confluence property. Section 3 recalls how to prove termination with syntactic well-founded orderings. The implementation of a simplification ordering is described. With such orderings, it is sufficient to prove that every left-hand side of rules is greater than the right-hand side, to ensure the termination of the rewrite program.

Most functional languages have an implicit evaluation strategy, often built in its compiler or interpreter. This is the case for instance in ASF+SDF [Kli93] or in ELAN for unlabelled rules, where an innermost strategy is applied for reduction. Then, of course, there exist programs that terminate with this innermost evaluation strategy, but would not terminate in general. Innermost termination of systems which are not terminating in general is addressed in Section 4.

Moreover, proving termination on all terms is not always needed. In practice, a rewrite systems is often designed to rewrite terms from a subset E, for example logical formulas in disjunctive normal form, flattened lists, or well-typed terms. Again, some rewrite systems may terminate only on such subsets E. The automaton-based techniques presented is Section 5 are well-suited to this restriction. They allow to study descendants of terms in E, i.e. all intermediate results of the program at every step of its execution, and the set of R-normal forms of E, i.e. all possible results obtained by executing the program R on the set of possible given inputs E, when the program stops. Under the left-linearity hypothesis of the rules, each of these two sets is contained in a regular tree language described by an approximation automaton. Once built, this automaton may be used for ckecking properties of rewrite programs. For example, one can prove that some values representing a deadlock or an error are actually unreachable.

Putting together terminating rewrite programs does not ensure termination of their union. However, a hierarchical structure of rewrite programs can help and an interesting question is to study modular normalisation strategies consisting for instance in normalising first with one set of rules, and then by another one, and to prove termination of this process. This so-called sequential reduction strategy is useful for proving termination of programs combining different methods of termination proof. Modular termination of union of rewrite system is studied in Section 6.

In ELAN, a program is composed of a set of unlabelled rules which are applied with an innermost reduction strategy, and a set of labelled rules whose application is defined thanks to strategies, that are built from the rule labels and from strategy constructors available in the language (don't care, don't know, first, repeat, then). The termination of programs under these user-defined strategies is an open problem. However it is possible to perform a static analysis of the programs to detect whether it has no result, just one, or many. This technique is also useful to detect some cases of non-termination. The normal form analysis of programs with strategies is

explained in Section 7. Finally, open questions are stated in Section 8.

Most results presented in this paper have been obtained in collaboration with other members of the Protheo group, namely Olivier Fissore, Thomas Genet and Pierre-Etienne Moreau.

2 Preliminary notions and notations

Comprehensive surveys can be found in [DJ90,BN98,Kir99,KK99] for rewrite systems. Definitions and notations used in this paper are recalled in Section 2.1, and a brief survey on termination is presented in Section 2.2. Then the ELAN system is introduced in Section 2.3 and gives the general context for motivating this study.

2.1 Terms, substitutions, rewrite systems

Let $\mathcal{F}$ be a finite set of symbols associated with an arity function denoted by $ar : \mathcal{F} \mapsto \mathbb{N}$, $\mathcal{X}$ be a countable set of variables, $\mathcal{T}(\mathcal{F}, \mathcal{X})$ the set of terms, and $\mathcal{T}(\mathcal{F})$ the set of ground terms (terms without variables). A *context* is a term of $\mathcal{T}(\mathcal{F} \cup \{\Box\}, \mathcal{X})$ with only one occurrence of $\Box$, where $\Box$ is a special constant not occurring in $\mathcal{F}$. For any term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $C[t]$ denotes the term obtained after replacement of $\Box$ by t in the context $C[\,]$. Positions in a term are represented as sequences of integers. The set of positions in a term t, denoted by $\mathcal{P}os(t)$, is ordered by the lexicographic ordering on integers. The empty sequence ϵ denotes the top-most position. $\mathcal{R}oot(t)$ denotes the symbol at position ϵ in t. If $p \in \mathcal{P}os(t)$, then $t|_p$ denotes the subterm of t at position p and $t[s]_p$ denotes the term obtained by replacement of the subterm $t|_p$ at position p by the term s. For any term $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, we denote by $\mathcal{P}os_{\mathcal{F}}(s)$ the set of functional positions in s, i.e. $\{p \in \mathcal{P}os(s) \mid p \neq \epsilon \text{ and } \mathcal{R}oot(s|_p) \in \mathcal{F}\}$.

The set of variables of a term t is denoted by $\mathcal{V}ar(t)$. A set of variables $\{x_1, \ldots, x_n\}$ is also denoted by $\overline{x}$. A term is linear if any variable of $\mathcal{V}ar(t)$ has exactly one occurrence in t. A substitution is a mapping σ from $\mathcal{X}$ into $\mathcal{T}(\mathcal{F}, \mathcal{X})$, which can uniquely be extended to an endomorphism of $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Its domain $\mathcal{D}om(\sigma)$ is $\{x \in X \mid x\sigma \neq x\}$.

A rewrite system $\mathcal{R}$ is a set of *rewrite rules* $l \to r$, where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$, and $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. A rewrite rule $l \to r$ is *left-linear* (resp. *right-linear*) if the left-hand side (resp. right-hand side) of the rule is linear. A rule is linear if it is both left and right-linear. A TRS $\mathcal{R}$ is linear (resp. left-linear, right-linear) if every rewrite rule $l \to r$ of $\mathcal{R}$ is linear (resp. left-linear, right-linear).

The set of function symbols $\mathcal{F}$ occurring in a TRS $\mathcal{R}$ can be partitioned into the set of *defined symbols* $\mathcal{D} = \{\mathcal{R}oot(l) \mid l \to r \in \mathcal{R}\}$ and the set of *constructors* $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$. A *constructor term* is a ground term with no defined symbol. The set of constructor terms is denoted by $\mathcal{T}(\mathcal{C})$.

The relation $\to_{\mathcal{R}}$ induced by $\mathcal{R}$ is defined as follows: for any $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $s \to_{\mathcal{R}} t$ if there exist a rule $l \to r$ in $\mathcal{R}$, a position $p \in \mathcal{P}os(s)$ and a substitution σ such that $l\sigma = s|_p$ and $t = s[r\sigma]_p$. The subterm $s|_p$ is called a redex. The transitive (resp. reflexive transitive) closure of $\to_{\mathcal{R}}$ is denoted by $\to^+_{\mathcal{R}}$ (resp. $\to^*_{\mathcal{R}}$). A term s

is *reducible* by $\mathcal{R}$ if there exists t such that $s \to_{\mathcal{R}} t$.

A term s is in $\mathcal{R}$-*normal form* (or is $\mathcal{R}$-*irreducible*) if s is not reducible by $\mathcal{R}$. A term s *has a normal form* if there exists a term t in $\mathcal{R}$-normal form such that $s \to_{\mathcal{R}}^* t$. The term t is also denoted by $s\!\downarrow$. The set of all ground terms in $\mathcal{R}$-normal form is denoted by $IRR(\mathcal{R})$, and $s \to_{\mathcal{R}}^* t$ with $t \in IRR(\mathcal{R})$ is denoted by $s \to_{\mathcal{R}}^! t$. The set of $\mathcal{R}$-descendants of a set of ground terms E is denoted by $\mathcal{R}^*(E)$ and $\mathcal{R}^*(E) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in E \ s.t. \ s \to_{\mathcal{R}}^* t\}$. The set of ground $\mathcal{R}$-normal forms of E is denoted by $\mathcal{R}^!(E)$ and $\mathcal{R}^!(E) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in E \ s.t. \ s \to_{\mathcal{R}}^! t\}$. Moreover, $\mathcal{R}^!(E) = \mathcal{R}^*(E) \cap IRR(\mathcal{R})$.

In general we consider conditional rewrite rules of the form $l \to r$ if c where c is a boolean term called the condition, written as a conjunction of the form $\bigwedge s \to t$. The rule applies if the condition, instantiated by the match from l to the term to be reduced, is satisfiable. The termination property of a conditional rewrite system can be reduced to the termination of an unconditional one, using the following transformation.

Every rule $l \to r$ if c with n conditions in c is transformed into $n+1$ unconditional rewrite rules using the following transformation U described in [OCM00]:

$$
\begin{aligned}
U(l \to r \text{ if } c) = \ & \{l \to r\} \text{ if } c \text{ is empty} \\
& \{l \to u(s, \overline{x})\} \cup U(u(t, \overline{x}) \to r \text{ if } c') \\
& \text{if } c = s \to t \wedge c' \\
& \text{where } u \text{ is a fresh function symbol and} \\
& \overline{x} = \mathcal{V}ar(l) \cap (\mathcal{V}ar(t) \cup \mathcal{V}ar(c') \cup \mathcal{V}ar(r))
\end{aligned}
$$

2.2 Termination

A rewrite system $\mathcal{R}$ is (1) *terminating* or *strongly normalising* if there exists no infinite derivation $s_1 \to_{\mathcal{R}} s_2 \to_{\mathcal{R}} \ldots$ where $s_1, s_2, \ldots \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, (2) *weakly normalising* (WN for short) if every s of $\mathcal{T}(\mathcal{F}, \mathcal{X})$ has a normal form, (3) *weakly normalising on* $E \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})$ (WN on E) if every $s \in E$ has a normal form, (4) *(strongly) innermost terminating* if there exists no infinite innermost derivation (i.e. a derivation in which redexes that are selected do not properly contain other redexes).

Existing methods for proving termination of rewrite systems (TRS in short) are essentially based on well-founded orderings containing the rewriting relation induced by the TRS. Among these methods, handling termination on free term algebras, we find syntactic and semantic methods, which provide a noetherian ordering $\gg$ via an analysis of the term structure in the TRS and ensure that, for all terms s and t, $s \to^* t$ implies $s \gg t$. Examples are the Path of Subterm Ordering [Pla78], the Lexicographic Path Ordering (LPO) [KL80], the Recursive Path Ordering (RPO) [Der82], polynomial interpretations [Lan79] [BCL87] and the General Path Ordering (GPO) [DH95,GG97], that generalizes previous ones.

Another approach consists in transforming the termination problem of a TRS R into the termination problem of another TRS R', provable with techniques of the previous category, and such that, for all terms s and t, $s \to_R^* t$ implies $s \to_{R'}^* t$.

Examples are transformation techniques [BL90] or semantic labelling [Zan95].

Another powerful method for proving termination of rewrite systems is described in [AG00]. Looking at rewrite systems as programs, the idea is that infinite reductions originate from the fact that defined symbols are introduced by the right-hand sides of rules. By tracing the introduction of these defined symbols, information can be obtained about termination of R. This is the idea behind the dependency pairs criterion. If $f(t_1, \ldots, t_n) \to t[g(s_1, \ldots, s_m)]$ is a rule in R, where f and g are defined symbols, and t is some context term, then the pair $(f(t_1, \ldots, t_n), g(s_1, \ldots, s_m))$ is a dependency pair of R. The technique proposed in [AG97a,Art97] is the following: a dependency graph is built, whose vertices are labelled with the dependency pairs. There is an edge from (s, t) to (u, v) if there exists a substitution σ such that $t\sigma \xrightarrow{*}_R u\sigma$. Since this last property is in general undecidable, the dependency graph has to be approximated by a super-graph with the same cycles. Dependency pairs are then used to generate a set of inequalities. If these equalities can be satisfied by a suitable well-founded ordering, R is terminating. More precisely, if there exists a quasi-ordering $\geq$, well-founded, closed under context and substitution, such that $l \geq r$ for each rule $l \to r$ in R, $s \geq t$ for each dependency pair (s, t) on a cycle of the dependency graph, and $s > t$ for at least one dependency pair (s, t) on every cycle of the dependency graph, then R is terminating. This method can prove innermost termination of R too.

2.3 ELAN

ELAN is an environment for specifying and prototyping deduction systems in a language based on rewrite rules controlled by strategies. It offers a natural and simple logical framework for the combination of computation and deduction paradigms. ELAN has a clear operational semantics based on rewriting logic [BKKR01] and on the rewriting calculus [CK99]. Its implementation involves compiled matching and reduction techniques integrating associative and commutative functions. Non deterministic computations return several results whose enumeration is handled thanks to a few constructors of choice strategies. A strategy language is available to control rewriting. Evaluation of strategies and strategy application is again based on rewriting.

The language is close to the algebraic specification formalism but provides additional specificities that are worth being emphasized.

- First, the language allows rules to be non-terminating and non-confluent, but then their application has to be controlled. So, there are rules for computations, which are required to be confluent and terminating, in order to give a unique result, and rules for deductions, for which neither confluence nor termination is required.
- Rules and strategies are first-class objects in the language. A strategy language is provided to express control and derivation tree exploration. A few strategy constructors are offered and efficiently implemented, to allow the user to design his own strategies.
- Application of rule or a strategy on a term may give 0, 1, or several results.

This non-determinism related to the production of sets of results is operationaly handled by backtracking.

As a consequence of these features, the language allows different programming styles. Functional programs are naturally expressed with confluent and terminating rules, while the backtracking mechanism, used to handle the search process, gives a flavour of logic programming and allows to program non-deterministic computations. The main originality is the capability of strategy programming to express the control of programs in a declarative way. The language provides the following strategy constructors.

- A labelled rule is a primal strategy. The result of applying a rule labelled lab on a term t returns a multiset of terms. This primal strategy fails if the multiset of resulting terms is empty.
- Two strategies can be concatenated by the symbol ";", which means that the second strategy is applied on all results of the first one. $S_1; S_2$ denotes the sequential composition of the two strategies. It fails if either S_1 fails or S_2 fails. Its results are all results of S_1 on which S_2 is applied and gives some results.
- dc$(S_1, \ldots, S_n)$ chooses in the list one strategy S_i that does not fail, and returns all its results. This strategy may return more than one result, or fails if all sub-strategies S_i fail.
- first$(S_1, \ldots, S_n)$ chooses in the list the first strategy S_i that does not fail, and returns all its results. Again, this strategy may return more than one result, or fails when all sub-strategies S_i fail.
- dc_one$(S_1, \ldots, S_n)$ chooses in the list one strategy S_i that does not fail, and returns its first result. This strategy returns at most one result or fails if all sub-strategies fail.
- first_one$(S_1, \ldots, S_n)$ chooses in the list the first strategy S_i that does not fail, and returns its first result. This strategy returns at most one result or fails if all sub-strategies fail.
- dk$(S_1, \ldots, S_n)$ chooses all strategies given in the list of arguments and for each of them returns all its results. This multiset of results may be empty, in which case the strategy fails.
- The strategy id is the identity that does nothing but never fails.
- fail is the strategy that always fails and never gives any result.
- repeat(S) applies repeatedly the strategy S until it fails and returns the results of the last unfailing application. This strategy may return more than one result but can never fail because zero applications of S is possible: in this case the initial term is returned.
- The strategy iterate(S) is similar to repeat(S) but returns all intermediate results of repeated applications.

Some strategy constructors introduced here are quite close to other tactics languages used on proof systems designed in the LCF style [Plo77,GMW79], such as

for instance Isabelle [Pau94]. They have been chosen to express main control constructions: concatenation, iteration and search. All these constructors are part of the ELAN language, and have been useful to design in ELAN theorem proving and constraint solving tools.

From now on, let us consider that not only rules but also strategies can be applied on terms. $(S)t$ denotes the application of the strategy S on the term t that produces a multiset of results. The general form of ELAN rules is actually as follows:

$$[\ell]\ \ l \to r\ \textbf{where}\ \ p_1 := (S_1)c_1 \ldots\ \textbf{where}\ \ p_n := (S_n)c_n$$

- $l, r, p_1, \ldots, p_n, c_1, \ldots, c_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,
- $\mathcal{V}ar(p_i) \cap (\mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})) = \emptyset$,
- $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_n)$ and
- $\mathcal{V}ar(c_i) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})$.

In such expressions, **where** $true := c$ is usually written **if** c. The pattern p_i is often reduced to a variable x. S_i may be the identity strategy, which is written $()c_i$ or simply c_i. An expression **where** $p := (S)c$ may also be considered as **if** $(S)c \to p$ where p is an irreducible term.

An additional construction **choose try** allows factorizing common parts of rules with the same left-hand side. For instance the two rules

$$\begin{aligned}
[\ell]\ l \to r\ &\textbf{where}\ y_1 := ()u_1 \\
&\textbf{if}\ c_2'(l, y_1) \\
[\ell]\ l \to r\ &\textbf{where}\ y_1 := ()u_1 \\
&\textbf{if}\ c_2''(l, y_1)
\end{aligned}$$

are factorized into one rule:

$$\begin{aligned}
[\ell]\ l \Rightarrow r\ &\textbf{where}\ y_1 := ()u_1 \\
&\textbf{choose} \\
&\quad \textbf{try if}\ c_2'(l, y_1) \\
&\quad \textbf{try if}\ c_2''(l, y_1) \\
&\textbf{end}
\end{aligned}$$

In this factorized form, the term u_1 is normalized only once and the assignement to y_1 is performed also only once. Examples of ELAN rules are given in the next section.

The study of ELAN programs motivate a wide range of termination and normalisation problems. For unlabelled rewrite rules, innermost termination is relevant. Since the language is modular, modularity of termination is also a problem. It may be interesting to find modular evaluation strategies whose termination is relatively easy to check in a modular way with different termination methods. For labelled rules controlled by strategies, the question of termination is widely open. A first step in this direction is to help the programmer to detect some non-terminating

programs. Another question to address is the analysis of determinism of rewrite programs with strategies. In the following sections, we consider these questions and propose some answers. Since ELAN is also a framework for designing proof tools, most of these methods have been developed in ELAN itself. Thanks to the reflection power of rewriting logic, we expect to apply them in a near future to ELAN programs.

3 Proving termination with syntactic well-founded orderings

Syntactic orderings, recalled in Section 2.2, are the easiest but also most fundamental tools to prove termination of rewrite systems.

Let us consider a simple syntactic method to prove termination of a rewrite system, based on a simplification ordering, that is an irreflexive transitive relation $>$ on terms, closed under context and substitution, that contains the subterm ordering. A rewrite system R is simply terminating if there exists a simplification ordering $>$ such that for any rule $l \to r$ in R, $l > r$. When the set $\mathcal{F}$ of operator symbols is finite, a rewrite system R is terminating if R is simply terminating [Der82]. Simplification orderings can be built from a well-founded ordering on the function symbols $\mathcal{F}$ called a precedence. An example is the following *lexicographic path ordering.*

Definition 3.1 *Let $>_F$ be a precedence on F. The lexicographic path ordering $>_{lpo}$ is defined on terms by $s = f(s_1, .., s_n) >_{lpo} t = g(t_1, \ldots, t_m)$ if one at least of the following condition holds:*

(i) $f = g$ *and* $(s_1, \ldots, s_n) >^{lex}_{lpo} (t_1, \ldots, t_m)$ *and* $\forall j \in \{1, \ldots, m\}, s >_{lpo} t_j$

(ii) $f >_F g$ *and* $\forall j \in \{1, \ldots, m\}, s >_{lpo} t_j$

(iii) $\exists i \in \{1, \ldots, n\}$ *such that either* $s_i >_{lpo} t$ *or* $s_i = t$.

The lexicographic path ordering is a simplification ordering [KL80].

This ordering can be used to show the termination of the rewrite system defining Ackermann's function. Assuming the precedence $ack >_F succ$, it amounts to show that for each rule, the left-hand side is greater than the right-hand side, i.e.:

$$\begin{aligned} ack(0, y) &>_{lpo} succ(y) \\ ack(succ(x), 0) &>_{lpo} ack(x, succ(0)) \\ ack(succ(x), succ(y)) &>_{lpo} ack(x, ack(succ(x), y)). \end{aligned}$$

In order to illustrate in the same time the ELAN syntax, let us show a few rules from the program implementing the ordering.

```
//------------ lpo ---------------------------------------------------------------
    lpo(@,@) : (term term) bool;
    @ >lex @ : (list[term] list[term]) bool;
    lpo(@,@) : (list[term] term) bool;
    lpo(@,@) : (term list[term]) bool;
    lpo1(@,@) : (term term) bool;
    lpo2(@,@) : (term term) bool;
    lpo3(@,@) : (term term) bool;
//--------------------------------------------------------------------------------
```

```
// Case : f(s1,...,sn) >lpo f(t1,...,tn)
// if s >lpo t1,...,tn et s1,...,sn >lex t1,...,tn
//
    [] lpo(s,t) => true
                if not(isvar(s)) and not(isvar(t))
                choose
                try
                        if head(s) ==sig head(t)
                        where b:= () lpo3(s,t)
                        if b
                try
                        if head(s) >sig head(t)
                        where b:= () lpo2(s,t)
                        if b
                end
    end
    [] lpo3(s,t) => b
                choose
                try
                        if list_subterm(s) >lex list_subterm(t)
                        where b := () lpo(s,list_subterm(t))
                try
                        where b := () false
                end
    end
//
// Case : f(s1,...,sn) >lpo g(t1,...,tn) if f >sig g  and  s >lpo t1,...,tn
//
    [] lpo2(s,t) => lpo(s,list_subterm(t))
    end
//
// Case :  si >lpo t
//
    [] lpo(s,t) => lpo1(s,t)
    end
    [] lpo1(s,t) => lpo(list_subterm(s),t)
    end
```

This program is used in ELAN applications, such as completion processes or the most complex program for proving termination presented in the next section. Other syntactic orderings (RPO,GPO) have been similarly implemented in ELAN.

4 Proving innermost termination

Until now, the problem of proving innermost termination, for possibly non terminating systems, has only been addressed with the dependency pairs method [AG97b] mentioned above. In [GKF00], another approach is proposed and uses an explicit induction based on a well-founded ordering on ground terms, which is closed under context and contains the subterm ordering. The ordering relation is incrementally constrained during the proof by adding restrictions corresponding to ground terms that have to be compared. The proof method works as follows: assuming that $\forall t' < t$, t' is innermost terminating, we prove that t is innermost terminating. This technique is used either to prove termination on ground terms or to determine which ground instances of generic patterns $f(x_1, ..., x_m)$, where f is a defined symbol, are innermost terminating.

The method is based on the development of a finite number of trees, each of them analysing a pattern $t_{ref} = g(x_1, \ldots, x_n)$ where g is a defined function symbol

of the signature and $x_1, ...x_n$ are distinct variables. The trees are built by applying three different mechanisms.

- The first one abstracts the current term $f(u_1, ...u_m)$ at the top-most position, while possibly setting some constraints. First, constraints $t_{ref} > u_{i_1}, ..., u_{i_p}$ are put in a set of constraints C, where the u_{i_j} are those among the u_i that are not already ground terms in normal form. Assuming by induction hypothesis, the existence of irreducible forms for $u_{i_1}, ..., u_{i_p}$, these subterms are abstracted by so-called NF-variables $X_{i_1}, ...X_{i_p}$, that are fresh variables in a set $\mathcal{N}$, and can only be instantiated by ground terms in normal forms. This is expressed by stating constraints $X_{i_1} = u_{i_1} \downarrow ...X_{i_p} = u_{i_p} \downarrow$ in a set A. We call that step the *abstraction* step. If the *abstraction* step cannot be applied because the constraints $t > u_{i_1}, ..., u_{i_p}$ cannot be proved to be satisfiable, the process stops with failure.
- The second step narrows the resulting term $f(U_1, ...U_m)$ at position ϵ in all possible ways in one step, with all possible rewrite rules of the rewrite system R, and all possible substitutions, into terms v, provided that the narrowing substitution is compatible with all constraints. This is the *narrowing* step. If the current term u is not narrowable, any of its ground instantiations is in normal form. Then u terminates, and the process stops with success in that branch.
- Before the previous steps, we can test for the current term u and the current set of ordering constraints C, whether A and C imply $t_{ref} > u$. In this case, by induction hypothesis, u terminates, and the process stops with success in that branch. This is the *stop* step.

These different steps are performed by rules that transform 3-tuples (T, A, C) where

- T is a set of terms of $\mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$, containing the current term whose innermost termination has to be proved. This is either a singleton or the empty set.
- A is a conjunction of abstraction constraints of the form $u \downarrow = X, u \in \mathcal{T}(\mathcal{F}, \mathcal{X}), X \in \mathcal{N}$. A ground substitution θ satisfies $u \downarrow = X$ if $X\theta = (u\theta) \downarrow$.
- C is a conjunction of ordering constraints. A ground substitution θ and an ordering $\succ$ on $\mathcal{T}(\mathcal{F})$ satisfy $t > t'$ if $t\theta \succ t'\theta$. Note that $t > t'$ is a kind of higher-order constraint here since the relation $>$ is unknown.

Applying the *stop* and the *abstraction* steps requires to prove some property. More precisely, given $t, u \in \mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$, we say that (A, C) implies $t > u$, denoted $(A, C) \twoheadrightarrow t > u$, if for any $\succ_1$ and any set of ground substitutions Θ satisfying (A,C), there exists $\succ_2$ such that $\succ_2$ and the same set of ground substitutions Θ satisfy $(A, C \wedge t > u)$. In general such a property is undecidable, but in practice, the proof developper can often find a proof by scrutining the constraints. The termination proof procedure is described by a set of rules, given in Table 1.

These rules must be applied with a specific strategy, implemented in ELAN as follows :

Table 1
Inference rules for t_{ref}-termination

Stop: $\dfrac{\{u\},\ A,\ C}{\emptyset,\ A,\ C \wedge t_{ref} > u}$

if $(A, C) \twoheadrightarrow t_{ref} > u$

Abstract: $\dfrac{\{f(u_1, ..., u_m)\},\ A,\ C}{\{f(U_1, ..., U_m)\},\ A \wedge u_{i_1} \downarrow = X_{i_1} \wedge \ldots \wedge u_{i_p} \downarrow = X_{i_p},\ C \wedge (t_{ref} > u_{i_1}, ..., u_{i_p})}$

where $f(u_1, ..., u_m)$ is abstracted by $f(U_1, ..., U_m)$ at the positions $i_1, ..., , i_p$
if $(A, C) \twoheadrightarrow t_{ref} > u_{i1}, ..., u_{ip}$

StopA: $\dfrac{\{f(u_1, ..., u_m)\},\ A,\ C}{\bot,\ A,\ C}$

if $(A, C) \twoheadrightarrow t_{ref} > u_1, ..., u_m$ cannot be proved

Narrow: $\dfrac{\{f(u_1, ..., u_m)\},\ A,\ C}{\{v\},\ \sigma A,\ C}$

if $f(u_1, ..., u_m) \leadsto_R^{\epsilon,\sigma} v$ and $(\sigma A,\ C)$ satisfiable

StopN: $\dfrac{\{u\},\ A,\ C}{\emptyset,\ A,\ C}$

if u is not narrowable or u is narrowable with σ and $(\sigma A,\ C)$ unsatisfiable

repeat*(first(**Abstract**,**StopA**) ;
first(dk(**Narrow**), **StopN**) ;
first one(dc(**test_narrow**), **StopN**) ;
first(**Stop**, id))

where **test_narrow** just checks whether narrowing on top is possible, but does not apply it.

There are three cases for the behavior of this process. The strategy, applied to the initial state $(\{t_{ref}\}, \top, \top)$, where $\top$ is the empty conjunction of constraints, may not terminate if there is an infinite number of applications of **Abstract** and **Narrow**. If all rules fail, and the process stops with a state of the form $(T \neq \emptyset, A, C)$, it is impossible to conclude anything. The good case is when the strategy terminates because the rules do not apply anymore and all states are of the form $(\emptyset, A, C)$. We write $SUCCESS(g, \succ)$ if application of the inference rules on $(\{g(x_1, ...x_m)\}, \top, \top)$, whose conditions are satisfied by $\succ$, gives a state of the form $(\emptyset, A, C)$ on any branch of the derivation tree.

Theorem 4.1 *[GKF00] Let R be a TRS on a set $\mathcal{F}$ of symbols. Let $\succ$ be an ordering on $\mathcal{T}(\mathcal{F})$, closed under context and containing the subterm ordering, such that for any defined symbol $g \in Def_R$, $SUCCESS(g, \succ)$. If the constants of $\mathcal{F}$ terminate innermost, then any term of $\mathcal{T}(\mathcal{F})$ terminates innermost.*

A refinement of this method consists in determining a priori a partial ordering

on symbols of the signature: f depends of g (noted $f >_d g$) if g occurs in the right-hand side of a rule defining f (i.e. f is the top symbol of the left-hand side). Then smaller symbols are handled first and partial conclusions may be used in checking conditions of **Abstract** or **Stop**.

The method has been implemented in ELAN. The program interacts with the user when it is not possible to automatically prove the conditions of **Abstract** or **Stop**. Let us illustrate on a small example the input given to ELAN, which is a specification file with the rewrite rules, whose termination has to be proved. Note that the rewrite system is not terminating but is innermost terminating.

```
specification example
Vars
x y
Ops
f:3 g:1
Rules
f(g(x),x,y) -> f(y,y,g(y))
g(g(x)) -> g(x)
nil
end of specification
```

The output is represented both as a derivation tree with the names of the applied rules, and a trace of the derivation.

```
f(g(x),x,y) -> f(y,y,g(y))
g(g(x)) -> g(x)

 --- Hanling symbol [g] ---
 * DERIVATION TREE *

 +- Abstract
    +- Narrow
       +- StopN


 * DERIVATION *

[Init]
t_ref =  g(x_1)
A = true
C = true

1 --> [Abstract]
t =  g(var(1))
A =  x_1|=var(1)
C =  g(x_1)>x_1

1.1 --> [Narrow]
t =  g(var(2))
A =  x_1|=g(var(2))
C =  g(x_1)>x_1

1.1.1 --> [StopN]
t =  vide
A =  x_1|=g(var(2))
C =  g(x_1)>x_1

 --- End of handling symbol [g] ---

 --- Handling symbol [f] ---
 * DERIVATION TREE *
 +- Abstract
    +- Narrow
       +- Abstract
```

```
          +- StopN

 * DERIVATION *

[Init]
t_ref =  f(x_1,x_2,x_3)
A = true
C = true

1 --> [Abstract]
t =  f(var(1),var(2),var(3))
A =  x_3|=var(3)  /\  x_2|=var(2)  /\  x_1|=var(1)
C =  f(x_1,x_2,x_3)>x_1,x_2,x_3

1.1 --> [Narrow]
t =  f(var(5),var(5),g(var(5)))
A =  x_3|=var(5)  /\  x_2|=var(4)  /\  x_1|=g(var(4))
C =  f(x_1,x_2,x_3)>x_1,x_2,x_3

1.1.1 --> [Abstract]
t =  f(var(5),var(5),var(6))
A =  g(var(5))|=var(6)  /\  x_3|=var(5)  /\  x_2|=var(4)  /\  x_1|=g(var(4))
C =  f(x_1,x_2,x_3)>x_1,x_2,x_3

1.1.1.1 --> [StopN]
t =  vide
A =  g(var(5))|=var(6)  /\  x_3|=var(5)  /\  x_2|=var(4)  /\  x_1|=g(var(4))
C =  f(x_1,x_2,x_3)>x_1,x_2,x_3
 --- End of handling symbol [f] ---
```

In this work, ELAN has been used to prototype the inference rules for the innermost termination proof method, to run examples and to experiment with different strategies. Thanks to the library of functions on terms already available in the system, the development of such rules has been done easily. This is also an example where user interaction is really important and can be implemented easily with the input-outpout functionalities of ELAN.

5 Computation or approximation of the set of normal forms

The previous section studies normalisation of ground terms under a specific strategy (the innermost one), but another kind of restriction is also interesting to consider: actually proving termination or weak normalisation on $\mathcal{T}(\mathcal{F},\mathcal{X})$ or on $\mathcal{T}(\mathcal{F})$ is not always needed. In practice, a TRS is often designed to rewrite terms from a subset $E \subseteq \mathcal{T}(\mathcal{F})$. Moreover, some TRSs are weakly normalising on a strict subset E of $\mathcal{T}(\mathcal{F})$, but not on $\mathcal{T}(\mathcal{F})$. For such cases, automaton-based techniques are quite convenient.

Several authors have proposed to use tree automata techniques for proving various properties on TRSs. For a given TRS R and a set of terms E, these proofs are based on the computation of the set of R-descendants of E and of the set of R-normal forms of E. Unfortunately, except on very few and specific cases, these computations do not terminate and are not regular sets. The idea proposed in [Gen98] is to use an approximation automaton that recognises a superset of the set of R-descendants of E and an automaton that recognises the set of R-normal forms of terms. Thanks to this method, it is shown in [Gen98] how to prove sufficient completeness of a

program on a set of possible initial inputs, or how to achieve reachability tests on a program.

Before giving a more precise description of how to build approximations of R-descendants and R-normal forms of E, some definitions and notations are needed.

5.1 Automata, Regular Tree Languages

Comprehensive surveys can be found in [GS84,CDG+97] for tree automata and tree language theory, and in [GT95] for connections between regular tree languages and rewrite systems.

Let $\mathcal{Q}$ be a finite set of symbols, with arity 0, called *states*. $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ is called the set of *configurations*. A *transition* is a rewrite rule $c \to q$, where $c \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ and $q \in \mathcal{Q}$. A *normalised transition* is a transition $c \to q$ where $c = f(q_1, \ldots, q_n)$, $f \in \mathcal{F}$, $ar(f) = n$, and $q_1, \ldots, q_n \in \mathcal{Q}$. For every transition, there exists an equivalent set of normalised transitions. Normalisation consists in decomposing a transition $s \to q$, into a set $Norm(s \to q)$ of flat transitions $f(u_1, \ldots, u_n) \to q'$ where $u_1, \ldots, u_n$, and q' are states, by abstracting subterms $s' \notin \mathcal{Q}$ of s by states of $\mathcal{Q}$.

A bottom-up finite tree automaton (tree automaton for short) is a quadruple $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, where $\mathcal{Q}_f \subseteq \mathcal{Q}$ is the set of final states, and Δ is a set of normalised transitions. The rewriting relation induced by Δ is denoted by $\to_\Delta$.

The tree language recognised by A is $\mathcal{L}(A) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists q \in \mathcal{Q}_f \ s.t. \ t \to_\Delta^* q\}$. For a given $q \in \mathcal{Q}$, the tree language recognised by A and q is $\mathcal{L}(A, q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \to_\Delta^* q\}$. A tree language (or a set of terms) E is *regular* if there exists a bottom-up tree automaton A such that $\mathcal{L}(A) = E$. The class of regular tree language is closed under boolean operations $\cup, \cap, \setminus$, and inclusion is decidable.

A $\mathcal{Q}$-substitution is a substitution σ such that $\forall x \in \mathcal{D}om(\sigma)$, $x\sigma \in \mathcal{Q}$. Let $\Sigma(\mathcal{Q}, \mathcal{X})$ be the set of $\mathcal{Q}$-substitutions.

5.2 Approximating $R^*(E)$ and $\mathcal{R}^!(E)$

The set of all ground terms in R-normal form $IRR(R)$ is a regular tree language if R is left-linear [GB85], and a procedure for building a regular tree grammar (resp. a tree automaton) producing (resp. recognising) $IRR(R)$ can be found in [CR87]. However, the set of R-descendants of a set of ground terms E, denoted by $R^*(E)$, is not necessarily a regular tree language, even if E is. The language $R^*(E)$ is regular if E is regular and if R is either a ground TRS [DT90], a right-linear and monadic TRS [Sal88], a linear and semi-monadic TRS [CDGV91] or an "inversely-growing" TRS [Jac96] [1]. On the other hand, for a given regular language E, $R^*(E)$ is not necessarily regular, even if R is a confluent and terminating linear TRS [GT95]. If R is not "inversely-growing", then $R^*(E)$ is not necessarily regular [Jac96].

In order to study the set of ground R-normal forms of E, denoted by $R^!(E)$ and

[1] "Inversely-growing" means that every right-hand side is either a variable, or a term $f(t_1, \ldots, t_n)$ where $f \in \mathcal{F}$, $ar(f) = n$, and $\forall i = 1, \ldots, n$, t_i is a variable, a ground term, or a term whose variables do not occur in the left-hand side.

defined as $R^*(E) \cap IRR(R)$, for a reasonably expressive class of rewrite programs, the idea is to build an approximation of $R^*(E)$, i.e. a regular superset of $R^*(E)$ for left-linear TRSs and regular sets E. Then, since regular langages are closed by intersection, the intersection between the regular superset of $R^*(E)$ and $IRR(R)$ gives a regular superset of $R^!(E)$.

Let A be the tree automaton that recognizes the set of terms E. For building the approximation automaton $\mathcal{T}_{\mathcal{R}}\uparrow(A)$ that recognizes a regular superset of $R^!(E)$, the algorithm proposed in [Gen98] starts from the tree automaton A and incrementally adds to Δ new transitions by computing critical peaks between rules of R and rules of Δ. This amounts to compute substitutions σ such that $r\sigma \ {}_{\mathcal{R}}\!\leftarrow l\sigma \rightarrow^*_\Delta q$. If $r\sigma \not\rightarrow^*_\Delta q$, then the transition $r\sigma \rightarrow q$ is added to Δ and normalised if needed. The choice of new states used to normalise $r\sigma \rightarrow q$ is guided by an approximation function γ that, given the rule $l \rightarrow r$, the state q and the substitution σ, computes a sequence of new states, one for each subterm under the root of r. The next two definitions are borrowed from [Gen98].

Definition 5.1 *(Approximation function) Let $\mathcal{Q}$ be a set of states, $\mathcal{Q}_{new}$ be a set of new states such that $\mathcal{Q} \cap \mathcal{Q}_{new} = \emptyset$, and $\mathcal{Q}^*_{new}$ the set of sequences $q_1 \cdots q_k$ of states in $\mathcal{Q}_{new}$. An approximation function is a mapping $\gamma : R \times (\mathcal{Q} \cup \mathcal{Q}_{new}) \times \Sigma(\mathcal{Q} \cup \mathcal{Q}_{new}, \mathcal{X}) \mapsto \mathcal{Q}^*_{new}$, such that $\gamma(l \rightarrow r, q, \sigma) = q_1 \cdots q_k$, where $k = Card(\mathcal{P}os_{\mathcal{F}}(r))$.*

In the following, for any sequence $S = q_1 \cdots q_k \in \mathcal{Q}^*_{new}$, and for all i such that $1 \leq i \leq k$, $\pi_i(S)$ denotes the i-th element of the sequence S, i.e. q_i.

Definition 5.2 *(Approximation Automaton) Let $A = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton, R a left-linear TRS, $\mathcal{Q}_{new}$ a set of new states such that $\mathcal{Q} \cap \mathcal{Q}_{new} = \emptyset$, and γ an approximation function. An approximation automaton $\mathcal{T}_{\mathcal{R}}\uparrow(A)$ is a tree automaton $\langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle$ such that*

(1) $\mathcal{Q}' = \mathcal{Q} \cup \mathcal{Q}_{new}$, and

(2) $\Delta \subseteq \Delta'$, and

(3)$\forall l \rightarrow r \in R$, $\forall q \in \mathcal{Q}'$, $\forall \sigma \in \Sigma(\mathcal{Q}', \mathcal{X})$, $l\sigma \rightarrow^*_{\Delta'} q$ implies

$$Norm(r\sigma \rightarrow q) \subseteq \Delta'.$$

The normalisation introduces new states according to the approximation function γ. The precise definition can be found in [Gen98].

By choosing specific approximation functions γ, we obtain different approximations. However, every approximation used with the previous construction provides a regular superset of $R^*(\mathcal{L}(A))$:

Theorem 5.3 *[Gen98] Given a tree automaton A and a left-linear TRS R, every approximation automaton satisfies: for any approximation function γ,*

$$\mathcal{L}(\mathcal{T}_{\mathcal{R}}\uparrow(A)) \supseteq R^*(\mathcal{L}(A))$$

In order to have a finite automaton approximating the set $R^*(\mathcal{L}(A))$, the intuition is to fold recursive calls into a unique new state. This is one of the basic idea of the ancestor approximation: informally, every state $q \in \mathcal{Q}' = \mathcal{Q} \cup \mathcal{Q}_{new}$ has a unique

ancestor $q_a \in \mathcal{Q}$. The ancestor of any state $q \in \mathcal{Q}$ is q itself, and the ancestor of every new state $q' \in \mathcal{Q}_{new}$ occuring in the sequence $\gamma(l \to r, q, \sigma)$ (used to normalise a new transition $r\sigma \to q$), is the ancestor of q. It is easy to see that in the ancestor approximation, the γ function does not depend on the σ parameter and moreover for any new state $q' \in \mathcal{Q}_{new}$, $\gamma(l \to r, q', \sigma) = \gamma(l \to r, q, \sigma)$, where $q \in \mathcal{Q}$ is the ancestor of q'. This property is crucial to make the construction of $\mathcal{T}_{\mathcal{R}\uparrow}(A)$ terminating.

Theorem 5.4 *[Gen98] Approximation automata built using ancestor approximation are finite automata.*

A library of algorithms on tree automata is available in ELAN. Usual algorithms on tree automata are implemented: union, intersection, cleaning, inclusion test, as well as algorithms for building the tree automata $\mathcal{T}_{\mathcal{R}\uparrow}(A)$, and $A_{IRR(R)}$, which is the automaton recognising the set $IRR(R)$ for a given automaton A and a given left-linear TRS R. Let us illustrate on a small example the functionalities available in this library.

A specification file contains the description of the signature (variables and operators) the set of rules defining the program, and an automaton describing the regular set of input terms for the program.

```
specification ex_nf_automaton
Vars x y z

Ops
a:0 b:0 cons:2 append:2 null:0
R1
append(null, x) -> x
append(cons(x, y), z) -> cons(x, append(y, z))
nil

Automata

// Terms of the form append(l1, l2) where l1 et l2 are
// any flat lists of a and b.

Description of A(0)
states q|0.q|1.q|2nil
final states q|0.nil
transitions append(q|1, q|1) -> q|0.
cons(q|2, q|1) -> q|1.
null -> q|1.
a -> q|2.
b -> q|2.
nil
End of Description
nil
end of specification
```

To compute the approximation automaton of `R1` on `A(0)`, i.e. a superset of the `R1`-descendants of `L(A(0))`, the user gives the query `T_up(R1)on(!A(0))`.

```
enter query term finished by the key word 'end':
T_up(R1) on(!A(0)) end

[] start with term :
    T_up(R1)on(!A(0))

[] result term:
    Description of A(0)
        states q|4.q|0.q|1.q|2.nil
```

```
        final states q|0.nil
        transitions cons(q|1,q|4)->q|4.append(q|1,q|1)->q|4.cons(q|1,q|4)->q|0.
                    append(q|1,q|1)->q|0.cons(q|2,q|1)->q|1.cons(q|2,q|1)->q|4.
                    cons(q|2,q|1)->q|0.null->q|1.null->q|4.null->q|0.a->q|2.b->q|2.nil
        End of Description
```

To build the automaton recognising the set of `R1`-irreducible terms, the user gives the query `build_nf(R1)`.

```
enter query term finished by the key word 'end':
build_nf(R1) end

[] start with term :
    build_nf(R1)

[] result term:
    Description of A(1)
        states q|0.q|1.nil
        final states q|1.nil
        transitions a->q|1.b->q|1.cons(q|1,q|1)->q|1.append(q|0,q|1)->q|1.
                    null->q|1.a->q|0.b->q|0.append(q|0,q|1)->q|0.nil
    End of Description
```

To compute the approximation of `R1`-normal forms of `L(A(0))`, i.e. a regular superset of `R1!(L(A(0))`, the user gives the query

```
simplify(T_up(R1) on(!A(0)) inter build_nf(R1)).
```

```
enter query term finished by the key word 'end':
simplify(T_up(R1) on(!A(0)) inter build_nf(R1)) end

[] start with term :
    simplify((T_up(R1)on(!A(0))inter build_nf(R1)))

[] result term:
    Description of A(1)
        states q|0.q|1.q|2.q|3.nil
        final states q|3.nil
        transitions b->q|0.a->q|0.null->q|3.null->q|2.null->q|1.cons(q|0,q|1)->q|3.
                    cons(q|0,q|1)->q|2.cons(q|0,q|1)->q|1.cons(q|1,q|2)->q|3.
                    cons(q|1,q|2)->q|2.nil
    End of Description
```

This construction of approximation automata and the ELAN library of algorithms on tree automata have been used for verification of properties on telecommunication protocols [GK00].

6 Modular termination of union of rewrite systems

For proving termination of a program with a large number of rewrite rules, it is obviously important to partition it into smaller systems whose termination is easier to prove. Unfortunately there is no hope to apply this divide and conquer approach in general for the termination property. Consider for instance $R_1 = \{a \to b\}$ and $R_2 = \{b \to a\}$ where a, b are constants. Each of these systems is terminating, but $R_1 \cup R_2$ is not. In this example however, function symbols are shared and a natural idea is to eliminate this case. The research on modularity for disjoint rewrite systems was initiated by [Toy87,Rus87]. The disjointness assumption was relaxed in the case of constructor systems that are allowed to share constructors,

while preserving the termination property [MT91,Gra94]. A survey of properties of rewrite systems preserved under (disjoint) unions can be found in [Mid90] and for CTRS in [Gra96]. In [Gra97], some syntactic criteria on the rewrite systems are studied in connection with modular aspects of rule based programs.

Following [KK90], we concentrate here on hierarchical TRSs defined as follows. Let R_1 and R_2 be TRSs with respective sets of symbols $\mathcal{F}_1$ and $\mathcal{F}_2$, respective sets of defined symbols $\mathcal{D}_1$ and $\mathcal{D}_2$, and respective sets of constructors $\mathcal{C}_1$ and $\mathcal{C}_2$. TRSs R_1 and R_2 are *hierarchical* if $\mathcal{F}_2 \cap \mathcal{D}_1 = \emptyset$ and $R_1 \subset \mathcal{T}(\mathcal{F}_1 \setminus \mathcal{D}_2, \mathcal{X}) \times \mathcal{T}(\mathcal{F}_1, \mathcal{X})$. Termination of hierarchical TRSs has been addressed in [Der94,KR95].

The strategy studied here is called the *Sequential Reduction Strategy* (SRS for short) and consists in separating a TRS R into several TRSs $R_1, \ldots, R_n$ such that $R = R_1 \cup \ldots \cup R_n$ and in normalising terms successively w.r.t. $R_1, \ldots, R_n$. This rewriting relation under SRS, denoted by $\rightarrow_{\mathcal{R}_1;\ldots;\mathcal{R}_n}$, and called *modular reduction relation* in [KK90], is defined as follows:
Let $R = R_1 \cup \ldots \cup R_n$ be TRSs. For $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $s \rightarrow_{\mathcal{R}_1;\ldots;\mathcal{R}_n} t$ if s is reducible by R and $\exists s_1, \ldots, s_{n-1} \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ such that $s \rightarrow^!_{\mathcal{R}_1} s_1$ and $s_1 \rightarrow^!_{\mathcal{R}_2} s_2$ and $\ldots$ and $s_{n-1} \rightarrow^!_{\mathcal{R}_n} t$.

This kind of strategy is of great interest when normalising terms w.r.t. a TRS splitted into several hierarchical TRSs $R_1, \ldots, R_n$. In this situation, interleaving rewriting steps from $R_1, \ldots, R_n$ is often not needed. If rewrite systems $R_1, \ldots, R_n$ are WN and share only constructors, then $\rightarrow_{\mathcal{R}_1;\ldots;\mathcal{R}_n}$ is terminating [Gen97], as a corollary of results of [KO91,Ohl94].

Now let us give an intuition on how to prove termination of $\rightarrow_{\mathcal{R}_1;\ldots;\mathcal{R}_n}$ for WN TRSs $R_1, \ldots, R_n$ sharing function symbols. Let $R = R_1 \cup R_2$, for example. For proving termination of $\rightarrow_{\mathcal{R}_1;\mathcal{R}_2}$ on a set of initial terms $E \subseteq \mathcal{T}(\mathcal{F})$, we need to prove that for any term $s \in E$, there is no possible infinite derivation $s \rightarrow^!_{\mathcal{R}_1} s'_1 \rightarrow^!_{\mathcal{R}_2} s_1 \rightarrow^!_{\mathcal{R}_1} s'_2 \rightarrow^!_{\mathcal{R}_2} s_2 \rightarrow^!_{\mathcal{R}_1} \ldots$ In that case, a criterion for proving termination of $\rightarrow_{\mathcal{R}_1;\mathcal{R}_2}$ on E is the following: construct the sets $G_1 = R^!_2(R^!_1(E))$, $G_2 = R^!_2(R^!_1(G_1))$, $\ldots$ until getting a fixpoint G_m such that $G_m = R^!_2(R^!_1(G_m))$. Then if R_1 is WN on $E, G_1, G_2, \ldots$, and R_2 is WN on $R^!_1(E), R^!_1(G_1), \ldots$ and if $G_m \subseteq IRR(R_1 \cup R_2)$, then R is WN on E and $\mathcal{R}$ is terminating on E under the Sequential Reduction Strategy.

Proposition 6.1 *If $R_1, \ldots, R_n$ are WN resp. on subsets $E_1, \ldots, E_n$ of $\mathcal{T}(\mathcal{F})$, the rewriting relation under the Sequential Reduction Strategy $\rightarrow_{\mathcal{R}_1;\ldots;\mathcal{R}_n}$ is terminating on E if the iterated sequence of sets $G_{k+1} = R^!_n(\ldots R^!_1(G_k)\ldots)$, starting from $G_0 = E$, has a fixpoint which is a subset of $IRR(R_1 \cup \ldots \cup R_n)$, and for all $k \geq 0$, $G_k \subseteq E_1, R^!_1(G_k) \subseteq E_2, \ldots,$ and $R^!_{n-1}(\ldots R^!_1(G_k)\ldots) \subseteq E_n$.*

In order to illustrate a termination proof in ELAN using this proof technique, let us consider two simple rewrite systems `R1` and `R2` defined in the following specification file.

```
specification ex_termination

Vars none
```

```
Ops
f:1 g:1 a:0 b:0
R1
f(a) -> b
nil

R2
a -> b
g(b) -> g(f(a))
nil

Automata
Description of A(0)
states q|0.q|1.q|2 nil
final states q|0.nil
transitions f(q|0) -> q|0.
a -> q|0.
g(q|1) -> q|0.
g(q|1) -> q|1.
a -> q|1.
nil
End of Description
nil
end of specification
```

To prove the termination of the sequential reduction relation on $\mathcal{T}(\mathcal{F})$, the user gives the query `start`.

```
enter query term finished by the key word 'end':
start end

[] start with term :
    start

[] result term:
   [true,
    Description of A(1)
        states q|0.q|1.q|2.q|3.q|4.q|5.q|6.nil
        final states q|6.nil
        transitions f(q|3)->q|4.f(q|3)->q|6.g(q|4)->q|4.g(q|4)->q|6.b->q|6.f(q|3)->q|3.
                    g(q|4)->q|3.b->q|3.g(q|2)->q|3.f(q|1)->q|2.b->q|1.g(q|5)->q|4.
                    g(q|5)->q|6.f(q|0)->q|5.b->q|0.nil
    End of Description]
```

To prove the termination of sequential reduction relation on `L(A(0))`, the user gives the query `start(!A(0))`.

```
enter query term finished by the key word 'end':
start(!A(0)) end

[] start with term :
    start(!A(0))

[] result term:
   [true,
    Description of A(2)
        states q|0.q|1.q|2.q|3.q|4.q|5.q|6.q|7.q|8.nil
        final states q|8.nil
        transitions b->q|5.b->q|8.f(q|5)->q|5.f(q|5)->q|8.g(q|6)->q|5.g(q|6)->q|8.
                    g(q|6)->q|6.g(q|4)->q|6.f(q|3)->q|4.b->q|3.g(q|7)->q|8.f(q|2)->q|7.
                    b->q|2.g(q|1)->q|5.f(q|0)->q|1.b->q|0.nil
    End of Description]
```

The sequential reduction strategy is interesting for proving termination of programs combining different methods of termination proof. The following specification

defines a function $fact\_list(i, j)$, that constructs a list of naturals $(i!, (i+1)!, \ldots, (j-1)!, j!)$. The rewrite program R_1 constructs the list and the rewrite program R_2 achieves the computation of the factorial function.

```
specification fact_list
Vars    x y z

Ops
  o:0 p:1 s:1 fact:1 plus:2 mult:2 cons:2 int:2
  intlist:1 null:0 fact_list:2 appfact:1

R1
  fact_list(x, y) -> appfact(int(x, y))
  appfact(null) -> null
  appfact(cons(x,y)) -> cons(fact(x),appfact(y))
  intlist(null) -> null
  intlist(cons(x, y)) -> cons(s(x), intlist(y))
  int(o,o) -> cons(o,null)
  int(o,s(y)) -> cons(o, int(s(o), s(y)))
  int(s(x),o) -> null
  int(s(x), s(y)) -> intlist(int(x, y))
  nil
```

```
R2
  p(s(x)) -> x
  mult(o, x) -> x
  mult(s(x), y) -> plus(mult(x, y), y)
  plus(x, o) -> x
  plus(x, s(y)) -> s(plus(x, y))
  fact(s(x)) -> mult(s(x), fact(p(s(x))))
  fact(o) -> s(o)
  nil

Automata
  Description of Aut(0)
  states q|0.q|1.nil
  final states q|0.nil
  transitions
    fact_list(q|1,q|1) -> q|0
    o -> q|1 s(q|1) -> q|1
    nil
  End of Description
  nil
```

Note that neither termination of R_1 nor termination of R_2 can be proved by a simplification ordering. However, termination of R_1 can be proved by the dependency pair method [AG97b], and on the other hand, termination of R_2 can be proved by GPO [DH95]. Instead of reconsidering the termination of the whole TRS $R_1 \cup R_2$, we can automatically verify that the (hierarchical) combination of those two systems is terminating under the sequential reduction strategy, for every initial term from the regular set $\mathcal{L}(Aut(0)) = \mathcal{L}(Aut(0), q_0) = \{fact\_list(n, p) \mid n, p \in \mathcal{L}(Aut(0), q_1)\}$ where $\mathcal{L}(Aut(0), q_1) = \{0, s(0), \ldots\} = Nat$. The query `start(Aut(0))` iterates the process described above and implemented with the `T_up` and `build_nf` operations, until getting a fixpoint. The result of the proof, performed in ELAN, is the following:

```
[] result term:
   [true,Description of nil states q|0.q|1.q|2.q|3.q|4.nil final
   states q|4.nil transitions cons(q|2,q|3)->q|3.null->q|4.null->q|3.cons(q|2,q|3)->q|4.
   s(q|0)->q|1.o->q|0.s(q|1)->q|1.s(q|1)->q|2.s(q|0)->q|2.nil End of Description]
```

where the first field is `true` — the combination is terminating under the sequential reduction strategy — and the second field contains the automaton recognising the superset of the normal forms: lists (possibly empty) of strictly positive natural numbers, which is what was expected by definition of function $fact\_list$, and which also proves sufficient completeness of $R_1 \cup R_2$ under sequential reduction strategy on $\mathcal{L}(Aut(0))$.

7 Normal form analysis of programs with strategies

Let us now consider programs in ELAN, given by a set of labelled rules whose application is defined thanks to strategies, expressed with the rule labels and with strategy constructors available in the language. In this section, we show how it is possible to perform a static analysis of the program to detect whether it has no

result, just one or many, when it terminates. This is done in particular by the ELAN compiler, whose performance has been improved by this analysis [KM98,KM00]. This technique is also useful to detect some cases of non-termination.

In order to facilitate the determinism analysis, we introduce five primitive operators that allow us to classify the cases according to two different levels of control.
For controlling the number of results: given a rewrite rule or a strategy,
- the one operator builds a strategy that returns at most one result;
- the all operator builds a strategy that returns all possible results of the strategy or the rule.

For controlling the choice mechanism: given a list of strategies (possibly reduced to a singleton),
- the select_one operator chooses and returns a non-failing strategy among the list of strategies;
- the select_first operator chooses and returns the first (from left to right) non-failing strategy among the list of strategies;
- the select_all operator returns all unfailing strategies.

In the current version of ELAN, these five primitives are hidden to the user but are internally used to perform the determinism analysis. However, all strategy constructors dk, dc, first, dc_one and first_one can be expressed using these primitives, using the following axioms, where S_i stands for a rule or a strategy:

$$
\begin{array}{lcl}
\mathsf{dk}(S_1,\ldots,S_n) & = & \mathsf{select\_all}(\mathsf{all}(S_1),\ldots,\mathsf{all}(S_n)) \\
\mathsf{dc}(S_1,\ldots,S_n) & = & \mathsf{select\_one}(\mathsf{all}(S_1),\ldots,\mathsf{all}(S_n)) \\
\mathsf{first}(S_1,\ldots,S_n) & = & \mathsf{select\_first}(\mathsf{all}(S_1),\ldots,\mathsf{all}(S_n)) \\
\mathsf{dc\_one}(S_1,\ldots,S_n) & = & \mathsf{select\_one}(\mathsf{one}(S_1),\ldots,\mathsf{one}(S_n)) \\
\mathsf{first\_one}(S_1,\ldots,S_n) & = & \mathsf{select\_first}(\mathsf{one}(S_1),\ldots,\mathsf{one}(S_n))
\end{array}
$$

For each strategy, a determinism information is inferred according to the maximum number of results it can produce (one or more than one) and whether or not it can fail. We adopt the same terminology for determinism as in Mercury [HCS96,HSC96]:

- if the strategy has exactly one result, its determinism is det(*deterministic*).
- if the strategy can fail and has at most one result, its determinism is semi(*semi-deterministic*).
- if the strategy cannot fail and has more than one result, its determinism is multi(*multi-result*).
- if the strategy can fail and may have more than one result, its determinism is nondet(*non-deterministic*).
- if the strategy always fail, i.e. has no result, its determinism is fail(*failure*).

The algorithm for inferring the determinism of strategies uses two operators *And* and *Or* that intuitively correspond to the composition and the union of two strategies (the union of two strategies is defined by the union of their results). For instance, a conjunction of two strategies is semi-deterministic if any one can fail and none of them can return more than one result ($And(\mathsf{det},\mathsf{semi}) = And(\mathsf{semi},\mathsf{det}) = And(\mathsf{semi},\mathsf{semi}) = \mathsf{semi}$).

The algorithm for inferring the determinism is presented here in three steps: for a

strategy, it uses the decomposed form of the strategy into the primitives introduced above. For a rule, it analyses the determinism of the matching conditions. Finally it deals with the recursion problem due to the fact that strategies are built from rules and that rules call strategies in their matching conditions.

Strategy detism inference. The detism of a strategy is inferred from its expression using one, all, select_one and select_all.

- detism(one(S)) = semi if S is a rewrite rule, since application of a rewrite rule may fail; otherwise,
- detism(one(S)) = $\begin{cases} \text{det} & \text{if detism(S) is det or multi} \\ \text{semi} & \text{if detism(S) is semi or nondet} \end{cases}$
- detism(all(S)) = And(semi, detism(S)) if S is a rewrite rule, since application of a rewrite rule may fail; otherwise, detism(all(S)) = detism(S).
- detism(repeat(S)) = $\begin{cases} \text{det} & \text{if detism(S) is det or semi} \\ \text{multi} & \text{if detism(S) is multi or nondet} \end{cases}$

  The repeat operator cannot fail because zero application of the strategy is allowed. Note that if S cannot fail, the repeat construction cannot terminate.
- detism(iterate(S)) = multi. The iterate operator cannot fail either. In general, it returns more than one result because all intermediate steps are considered as results. If S cannot fail, the iterate construction cannot terminate, but this is quite useful to represent infinite data structures, like infinite lists.
- detism($S_1; S_2$) = And(detism(S_1), detism(S_2)).
- detism(select_one($S_1, \ldots, S_n$)) = And(detism(S_1), $\ldots$, detism(S_n))
- detism(select_all($S_1, \ldots, S_n$)) = Or(detism(S_1), $\ldots$, detism(S_n))

Rule detism inference. Inferring the determinism of a rewrite rule R consists of analysing the determinism of its matching conditions.

- Let us first consider a matching condition **where** $p := c$ where c does not involve any strategy. The normalisation of c (with unlabelled rules) cannot fail. If p does not match the normalised term, the current rule cannot be applied, but this does not modify the detism of the rule. Such a condition is usually said to be deterministic (det is a neutral element for the And operator).

  The only different situation is when a variable of c occurs in the left-hand side of the rule or in a pattern of a previous matching condition with AC function symbols: if this variable is involved in an AC matching problem, it may have several possible instances, thus, an application of the rule may return more than one result. The matching condition is said to be multi.
- Let us now consider a matching condition **where** $p := (S)t$ involving a strategy call. Then the matching condition has in general the determinism of the strategy S, except as before when a variable of t occurs in the left-hand side of the rule or in a pattern of a previous matching condition with AC function symbols: the detism of the matching condition is multi or nondet, and is computed as And(multi, detism(S)).

The determinism of the rewrite rule R is the conjunction (And operation) of the inferred determinisms of all its matching conditions.

Recursion problem. In general, strategy definitions may be (mutually) recursive. So the detism of a strategy may depend on itself. A similar problem arises in logic programming for finding the determinism of a predicate [ST85]. To avoid non-termination of the determinism analysis algorithm, when the detism of a strategy depends on itself, a default determinism is given. On the strategy constructors, this default is semi for **one**, nondet for **all**, multi for **repeat** and **iterate**, nondet for `;`. In order to refine this brute force approximation, fixpoint analysis should be explored.

Finally the determinism analysis is also useful to detect some non-terminating strategies, such as a strategy repeat(S), where S never fails. Detecting this non-termination problem at compile time allows the system to give a warning to the programmer and can help him to improve his strategy design.

It is relatively easy to write a non-terminating strategy in ELAN. For example the strategy `repeat(first(S,id))` where `S` is any strategy will never terminate since `first(S,id)` never fails: if `S` fails, `id` applies! The determinism analysis presented above is able to detect this non-termination case.

In order to improve this capability, it should be possible to consider other pathological cases. For instance, when the program contains a rule like `[r] x => x+x` whose left-hand side matches any expression of type integer, `repeat(r)` does not terminate.

Termination can be recognized too in some cases : assume that the program contains a definition of a function `f` by case analysis (an ML-like program) and that `f` is completely defined with respect to the constructors with rules labelled `r1` to `rm`. After a finite number of reductions by `(r1,...,rm)`, the reduced term does not contain `f` anymore. Then the strategy `first(r1,...,rm)` fails and `repeat(first(r1,...,rm))` terminates.

8 Conclusion

To conclude, let us mention some open problems that need further research. First modular termination criteria are yet very limited if the programs are not restricted to left-linear rules.

Termination under strategies, even with leftmost innermost strategies, has been very little studied. Lazy, outermost and needed strategies are also worth exploring, especially when we are interested in the size of rewriting derivations. In an on-going project where we use ELAN to compute rewrite proofs for the proof assistant Coq and where ELAN returns not only the normal form but also the proof term (i.e. the trace) of the derivation, it is crucial to generate shorter derivations [AN00].

Using strategy constructors, like in ELAN or Stratego [VB98], leads to consider other techniques for proving properties of strategies, possibly using temporal logic.

A last difficulty not yet mentioned is termination modulo an equational theory (like associative-commutative theories) or modulo a set of unlabelled rules.

References

[AG97a] T. Arts and J. Giesl. Automatically proving termination where simplification orderings fail. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development, Proceedings of 7th International Joint Conference CAAP/FASE*, volume 1214 of *Lecture Notes in Computer Science*, pages 261–272. Springer-Verlag, 1997.

[AG97b] T. Arts and J. Giesl. Proving innermost termination automatically. In *Proceedings 8th Conference on Rewriting Techniques and Applications, Sitges (Spain)*, volume 1232 of *Lecture Notes in Computer Science*, pages 157–171. Springer-Verlag, 1997.

[AG00] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

[AN00] C. Alvarado and Quang-Huy Nguyen. Elan for equational reasoning in Coq. In J. Despeyroux, editor, *Proceedings of the Workshop on Logical Frameworks and Meta-languages*, Santa Barbara (California), June 2000.

[Art97] T. Arts. *Automatically proving termination and innermost normalisation of term rewriting systems.* PhD thesis, Utrecht University, The Netherlands, 1997.

[BCL87] A. Ben Cherifa and P. Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9(2):137–160, October 1987.

[BKK+96] P. Borovanský, C. Kirchner, H. Kirchner, P-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in TCS*, Asilomar (California), September 1996.

[BKK+98] P. Borovanský, C. Kirchner, H. Kirchner, P-E. Moreau, and C. Ringeissen. An overview of ELAN. In C. and H. Kirchner, editors, *Proceedings of the second International Workshop on Rewriting Logic and Applications*, Pont-à-Mousson (France), September 1998. Electronic Notes in Theoretical Computer Science, volume 15.
`http://www.elsevier.nl/locate/entcs/volume15.html`.

[BKKR01] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 2001.

[BL90] F. Bellegarde and P. Lescanne. Termination by completion. *Applicable Algebra in Engineering, Communication and Computation*, 1(2):79–96, 1990.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and all That.* Cambridge University Press, 1998.

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and Tommasi. Tree automata techniques and applications, 1997.
`http://l3ux02.univ-lille3.fr/tata/`.

[CDGV91] J.L. Coquidé, M. Dauchet, R. Gilleron, and S. Vágvölgyi. Bottom-up tree pushdown automata and rewrite systems. In R. V. Book, editor, *Proceedings 4th Conference on Rewriting Techniques and Applications, Como (Italy)*, volume 488 of *Lecture Notes in Computer Science*, pages 287–298. Springer-Verlag, 1991.

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the 1st International Workshop on Rewriting Logic and its Applications, (Asilomar, Pacific Grove, CA, USA)*, volume 5 of *Electronic Notes in Theoretical Computer Science*. North Holland, September 1996.

[CK99] H. Cirstea and C. Kirchner. Combining higher-order and first-order computation using ρ-calculus: Towards a semantics of ELAN. In Dov Gabbay and Maarten de Rijke, editors, *Frontiers of Combining Systems 2*, Research Studies, ISBN 0863802524, pages 95–120. Wiley, 1999.

[CM96] M. Clavel and J. Meseguer. Reflection and Strategies in Rewriting Logic. In J. Meseguer, editor, *Proceedings of the 1st International Workshop on Rewriting Logic and its Applications, (Asilomar, Pacific Grove, CA, USA)*, volume 5 of *Electronic Notes in Theoretical Computer Science*. North Holland, September 1996.

[CR87] H. Comon and J-L. Rémy. How to characterize the language of ground normal forms. Technical Report 676, INRIA-Lorraine, 1987.

[Der82] N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[Der94] N. Dershowitz. Hierarchical termination. In *Proceedings 4th International Workshop on Conditional Term Rewriting Systems, Jerusalem (Israel)*, volume 968 of *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag, 1994.

[DH95] N. Dershowitz and C. Hoot. Natural termination. *Theoretical Computer Science*, 142(2):179–207, May 1995.

[DJ90] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990. Also as: Research report 478, LRI.

[DT90] M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *Proceedings 5th IEEE Symposium on Logic in Computer Science, Philadelphia (Pa., USA)*, pages 242–248, June 1990.

[FN97] K. Futatsugi and A. Nakagawa. An overview of CAFE specification environment – an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Proceedings of the 1st IEEE Int. Conference on Formal Engineering Methods*, 1997.

[GB85] J. H. Gallier and R. V. Book. Reductions in tree replacement systems. *Theoretical Computer Science*, 37:123–150, 1985.

[Gen97] T. Genet. Proving termination of sequential reduction relation using tree automata. Technical Report 97-R-091, Centre de Recherche en Informatique de Nancy, 1997.

[Gen98] T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In *Proceedings 9th Conference on Rewriting Techniques and Applications, Tsukuba (Japan)*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165. Springer-Verlag, 1998.

[GG97] T. Genet and I. Gnaedig. Termination proofs using gpo ordering constraints. In M. Dauchet, editor, *Proceedings 22nd International Colloquium on Trees in Algebra and Programming, Lille (France)*, volume 1214 of *Lecture Notes in Computer Science*, pages 249–260. Springer-Verlag, 1997.

[GK00] T. Genet and F. Klay. Rewriting for cryptographic protocol verification. In *Proceedings International Conference on Automated Deduction, CADE-17.* Springer-Verlag, LNAI, 2000.

[GKF00] I. Gnaedig, H. Kirchner, and O. Fissore. Induction for termination, 2000. Research report.

[GMW79] M. Gordon, A. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, New York (NY, USA), 1979.

[Gra94] B. Gramlich. Generalized sufficient conditions for modular termination of rewriting. *Applicable Algebra in Engineering, Communication and Computation*, 5:131–158, 1994.

[Gra96] B. Gramlich. On termination and confluence properties of disjoint and constructor-sharing conditional rewrite systems. *Theoretical Computer Science*, 165(1):97–131, September 1996.

[Gra97] B. Gramlich. Modular aspects of rewrite-based specifications. In *Proc. 12th Int. Workshop on Algebraic Development Techniques, WADT'97 (Abstracts)*, Tarquinia, Italy, June 1997.

[GS84] F. Gécseg and M. Steinby. *Tree automata.* Akadémiai Kiadó, Budapest, Hungary, 1984.

[GT95] R. Gilleron and S. Tison. Regular tree languages and rewrite systems. *Fundamenta Informaticae*, 24:157–175, 1995.

[HCS96] F. Henderson, T. Conway, and Z. Somogyi. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29:17–54, October-December 1996.

[HSC96] F. Henderson, Z. Somogyi, and T. Conway. Determinism analysis in the Mercury compiler. In *Proceedings of the Nineteenth Australian Computer Science Conference*, pages 337–346, Melbourne, Australia, January 1996.

[Jac96] F. Jacquemard. Decidable approximations of term rewriting systems. In H. Ganzinger, editor, *Proceedings 7th Conference on Rewriting Techniques and Applications, New Brunswick (New Jersey, USA)*, pages 362–376. Springer-Verlag, 1996.

[Kir99] H. Kirchner. *Term Rewriting*, chapter 9, pages 273–320. IFIP State-of-the-Art Reports. Springer, 1999. Report LORIA 99-R-098.

[KK90] M. Kurihara and I. Kaji. Modular term rewriting systems and the termination. *Information Processing Letters*, 34:1–4, February 1990.

[KK99] C. Kirchner and H. Kirchner. Rewriting, solving, proving. A preliminary version of a book available at `www.loria.fr/~ckirchne/rsp.ps.gz`, 1999.

[KL80] S. Kamin and J.-J. Lévy. Attempts for generalizing the recursive path ordering. Unpublished manuscript, 1980.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[KM98] H. Kirchner and P-E Moreau. Non-deterministic computations in ELAN. In J.L. Fiadeiro, editor, *Recent Developements in Algebraic Specification Techniques, Proc. 13th WADT'98, Selected Papers*, number 1548 in Lecture Notes in Computer Science, pages 168–182. Springer-Verlag, 1998. Report LORIA 98-R-278.

[KM00] H. Kirchner and P-E Moreau. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 2000. To appear.

[KO91] M. Kurihara and A. Ohuchi. Modular term rewriting systems with shared constructors. *Journal of Information Processing of Japan*, 14(3):357–358, 1991.

[KR95] M. Krishna Rao. Modular proofs for completeness of herarchical term rewriting systems. *Theoretical Computer Science*, 151(2):487–512, 1995.

[Lan79] D. S. Lankford. On proving term rewriting systems are noetherian. Technical report, Louisiana Tech. University, Mathematics Dept., Ruston LA, 1979.

[Mid90] A. Middeldorp. *Modular Properties of Term Rewriting Systems.* PhD thesis, Vrije Universiteit, Amsterdam, 1990.

[MT91] A. Middeldorp and Y. Toyama. Completeness of combinations of constructor systems. In *Proceedings 4th Conference on Rewriting Techniques and Applications, Como (Italy)*, 1991. also Report CS-R9058, CWI, 1990.

[OCM00] E. Ohlebusch, C. Claves, and C. Marché. TALP: a tool for the termination analysis of logic programs. In *Proceedings 11th Conference on Rewriting Techniques and Applications, Norwich (UK)*, volume 1833 of *Lecture Notes in Computer Science*, pages 270–273. Springer-Verlag, 2000.

[Ohl94] E. Ohlebusch. *Modular Properties of Composable Term Rewriting Systems.* PhD thesis, Universität Bielefeld, Bielefeld, 1994.

[Pau94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[Pla78] D. Plaisted. A recursively defined ordering for proving termination of term rewriting systems. Technical Report R-78-943, U. of Illinois, Dept of Computer Science, 1978.

[Plo77] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Rus87] M. Rusinowitch. On termination of the direct sum of term rewriting systems. *Information Processing Letters*, 26(2):65–70, 1987.

[Sal88] K. Salomaa. Deterministic Tree Pushdown Automata and Monadic Tree Rewriting Systems. *Journal of Computer and System Sciences*, 37:367–394, 1988.

[ST85] H. Sawamura and T. Takeshima. Recursive unsolvability of determinacy, solvable cases of determinacy and their applications to Prolog optimization. In *Proceedings of the Second International Logic Programming Conference*, pages 200–207, Boston, Massachusetts, 1985.

[Toy87] Y. Toyama. On the church-rosser property for the direct sum of term rewritig systens. *Journal of the ACM*, 34(1):128–143, January 1987.

[VB98] E. Visser and Z. Benaissa. A core language for rewriting. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the second International Workshop on Rewriting Logic and Applications*, volume 15, Pont-à-Mousson (France), September 1998. Electronic Notes in Theoretical Computer Science. `http://www.elsevier.nl/locate/entcs/volume15.html`

[Zan95] H. Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995.