N° d'ordre : 1348

UNIVERSITÉ PARIS-SUD 11
UFR SCIENTIFIQUE D'ORSAY

THÈSE

Présentée par

XAVIER URBAIN

Pour obtenir

L'HABILITATION À DIRIGER LES RECHERCHES
DE L'UNIVERSITÉ PARIS-SUD 11

Sujet :

PREUVE AUTOMATIQUE : TECHNIQUES, OUTILS ET CERTIFICATION

Soutenue le 29 novembre 2010 devant la commission d'examen composée de

M. NACHUM DERSHOWITZ	rapporteur
M. JÜRGEN GIESL	rapporteur
M. JEAN GOUBAULT-LARRECQ	rapporteur
M. CLAUDE MARCHÉ	
M. JEAN-FRANÇOIS MONIN	
Mme BRIGITTE ROZOY	présidente

Contents

I Introduction **5**

II Automated Proofs of Termination **11**
II.1 Termination of Rewriting, Overview 11
II.1.1 Preliminaries and Notations 11
II.1.2 Generalities 11
II.1.3 Problem Transformations, Criteria, Orderings 12
II.1.4 Dependency Pairs, Dependency Graphs 13
II.2 Orderings 19
II.2.1 Polynomial Interpretations 20
II.3 Incremental and Modular Criteria 25
II.4 Equational Theories, Strategies 31
II.4.1 AC Rewriting 31
II.4.2 Context-Sensitive Rewriting 37
II.4.3 Membership Equational Programs 39

III An Architecture for Certification of Automated Proofs **45**
III.1 Trustworthy Decision Procedures 45
III.2 Satellite-Based Architecture 46
III.2.1 The COQ Proof Assistant 46
III.2.2 COCCINELLE 48
III.2.3 C*i*ME 3 50
III.2.4 Trace Language 52
III.3 Proof Techniques, the Case of Graphs 53
III.3.1 Formalisation of Hierarchical Decomposition 54
III.4 Formal Proofs and Criteria 56

IV Conclusion and Perspectives **63**

Chapter I

Introduction

The omnipresence of software and computing devices, notably in often critical domains such as medical techniques, transportation, nuclear platforms, etc., brings to the fore the question of the level of confidence we put in the programs which surround us. Does this program always return a result? Does it compute what it is supposed to? Will this software behave as expected? Ensuring and moreover certifying behaviour or good properties of a program or a procedure, in particular with the additional guarantee of a mechanically checked proof, often amounts to tackling intermediate problems that are computationally or technically difficult to solve for a non-expert, and to which the developer should not dedicate too much of his proving effort. My research interests aim to contribute to the development of trustworthy software. They address automated proof, and in particular the means to bring as much automation as possible into the process of (mechanical) verification of program properties.

The general context of my contributions runs along a journey starting from the choice of a base formalism and a preferred property. It goes through the development of proof techniques for this property, that must be well-suited to automation, programming practices, and more into solving problems one meets in practice than pathological cases seldom encountered.

Its next step is to come closer to actual programming languages, principally by considering expressive features upon the base formalism, and then studying how our automated techniques extend, for instance, to symbols with associative and commutative behaviour, to evaluation strategies or to additional conditions, etc.

I strongly believe that, at each step along the way, results have to be implemented, as they were in the first place designed to be used in concrete situations. Software developments are thus a fundamental component of my work. As a consequence, a key point is the trust that one can place in automated provers, especially with reference to their interaction with proof assistants (within which proofs are usually developed interactively, step by step, and mechanically checked). This involves contributions on formal models, communication between tools, techniques for formal proof, etc. At this stage, one can start thinking about delegating proofs to automated provers in formal developments.

I shall hereafter describe this journey, with a summary of my contributions. I shall eventually mention additional inflowing experiences that offer interesting perspectives on different programming paradigms.

Formalism, Property

The course of my research starts in the context of first-order *rewriting systems*, when I joined the DÉMONS team at LRI, at that time headed by Jouannaud, and already very active on this topic.

First-order rewriting constitutes a readable and clear way of presenting computations and equational reasoning. For example, a function @ that computes the concatenation of two lists may be defined with the two rules $\mathit{nil} @ l \rightarrow l$ and $(e :: l_1) @ l_2 \rightarrow e :: (l_1 @ l_2)$. Basically, rewriting consists of directed equations possibly making equational reasoning 'one way': associativity of @ can for instance be expressed with the single rule $(l_1 @ l_2) @ l_3 \rightarrow l_1 @(l_2 @ l_3)$. Widely used for proof, specification, programming,

modelling languages (especially declarative style paradigms), etc., that is, almost everywhere equations may be involved, rewriting enjoys also the remarkable property of being easily implementable: one can find efficient rewriting engines like MAUDE developed at SRI [27], ÉLAN from LORIA [19], or C*i*ME from LRI [36].

In this context, I have focused since 1997 on the property of *termination*, that is the fundamental property of a program that eventually returns a result on *any* input. Termination is closely related to the existence of any calculus defined by a program: no termination means no result in finite time, at least not always. It notably plays a fundamental role at several levels in programs, proofs and definitions. For instance, termination of a relation $\rightarrow$, i.e. the *well-foundedness* of its inverse, is crucial for induction proofs with reference to $\leftarrow$. Some proof assistants require that the functions one defines are *total*, i.e. that their computations always terminate. It also often acts as a preliminary for proofs of other properties: for example the confluence of a relation defined by a finite rewriting system becomes decidable as soon as the relation is proven to be terminating. Showing termination is, in particular, compulsory when one wants to prove *total correctness* of programs.

The halting problem, well-known to be undecidable, boils down to termination of first-order rewriting, even for rewriting systems consisting of only one rule [43]. Thus, most efforts focus on defining techniques devoted to prove termination of as many programs as possible, but which cannot be complete. In the following chapters, I shall emphasise that proving termination consists in translating termination problems into other termination problems until one reaches an empty or trivial problem. One can then define a termination criterion as a translation that is (of course) sound, and, moreover, complete. Writing criteria as inference rules, a termination proof may be naturally represented as a proof tree. I shall give a brief overview of termination techniques in Section II.1, from the historical Manna-Ness criterion [120] to the breakthrough for automated proof introduced by the dependency pair approach [9] and its various refinements (marks, graphs, etc.). From our inference rules point of view, a termination proof (tree) can be seen as follows: termination of a system R will be equivalent, by application of rule DP (for dependency pairs), to the termination of its dependency chain relation $\twoheadrightarrow_{\mathsf{DP}(R),R}$. This new problem will in turn be equivalent to terminations of all relations $\twoheadrightarrow_{D_i,R}$ corresponding to, say, strongly connected component in a dependency graph (rule GRAPH), and so on, until all the obtained branches in this tree will be eventually closed, either by application of an axiom, or by reaching a termination problem for a relation that is empty.

Proof Techniques

A technique well-suited to automation obviously enjoys effective means to close those proof trees. The use of orderings is one of them. The Manna-Ness criterion requires a well-founded reduction ordering in which the system is embedded; dependency pair approaches, noting that non-termination implies infinite chains of recursive calls, may require weak orderings pairs to prove finiteness of the so-called dependency chains, if any. Section II.2.1 will describe orderings based on polynomial interpretations, and in particular contributions that Contejean, Marché, Tomás and I brought to their automation [38]: efficient techniques regarding suitability checking, as well as regarding discovery of suitable polynomials with full automation.

An important step is to define proof techniques that are well-suited to program verification. Programs are usually devised over libraries and more generally in an incremental and modular (hierarchical) fashion; a natural question is whether one can obtain termination techniques that match this good practice.

This concern was at the heart of my PhD [158], supervised by Marché, and I proposed some notions to represent systems, as well as some results on modularity and techniques to prove termination hierarchically [159, 160] that I shall present in Section II.3. In sharp contrast to previous attempts, the constraints I put on unions of systems are very weak, and a first goal is reached in the sense that hierarchical unions are allowed. Instead, I focus on a stronger notion of termination that behaves nicely when

systems contain non-deterministic projections: CE-termination, while being far less restrictive than the other compatible notion of simple termination.

The theoretical results obtained can be made effective with the use of orderings provided that they do not increase for projections, which is the case for most classical orderings and their usual combinations. The approach I propose, in particular, allows one to discard some irrelevant rules in the process of discovering a termination proof, hence simplifying the problem to solve considerably.

Expressive features

Aiming to verify properties of programs, it is worth noticing that programming languages enjoy expressive features such as equational theories, evaluation strategies, or membership conditions. Extensions of rewriting with some of these features are discussed in Section II.4.

Associative and commutative (AC) operations are often met in programming languages, addition of integers for instance is one of the most common. In the context of term rewriting, the usual way to handle AC-symbols is to work on AC-equivalence classes: simply adding $x + y \rightarrow y + x$ or working modulo C with an associativity rule would actually result in losing termination immediately. In fact, equational theories like AC make termination proofs considerably more difficult as, for example, AC-equivalent terms cannot be strictly ordered: one asks suitable orderings to be *compatible* with the theory. Compatibility conditions for polynomial interpretations may be easily checked [14], but defining an AC-compatible version of RPO-like orderings [47] is an arduous task, widely studied [12, 14, 44, 94, 95, 145], that leads to very restrictive conditions (with transformations systems) and/or intricate definitions of simplification orderings.

Dependency pairs had recently been introduced when Marché and I decided to explore the new possibilities offered by this approach, in particular for AC-rewriting on flat terms. In a first article in 1998 we defined an extension to the AC-case of dependency pairs that do not involve *marks* [122]. Some versions of the dependency pair approach may indeed use a set of symbols that is extended with marks, and they can get more powerful this way. However, marks do not mix easily with equational theories that can permute symbols. We proposed a definition for marked dependency pairs in 2002 in collaboration with Kusakari [104], which involves a system that, roughly speaking, 'unflattens' marked terms.

The hierarchy of modules, however, is unchanged by AC-operations, and studying the extension of my results on modularity to AC-rewriting gave Marché and me the opportunity to define in [123] a notion of *abstract* AC-dependency pairs that can be instantiated by the concrete (marked or unmarked) definitions existing at that time [70, 104, 106, 122].

Remarkably, the approach I use to establish the results of my thesis transfers smoothly to AC-rewriting; in addition to techniques for hierarchical termination of AC-rewriting, [123] proposes modularity results for CE-termination modulo AC.

Another common practice is to define an evaluation strategy. A *context-sensitive* (CS) strategy consists in forbidding any evaluation/reduction step for some arguments of functions. Such syntactic replacement restrictions are found in the design of several programming languages, such as MAUDE [27], CAFEOBJ [67], etc. CS-strategies are related to lazy evaluation, and can provide the ability to deal with infinite objects. A classical example is to forbid evaluation in the two branches of an *if*, to force evaluation of the condition, as usual. Termination of CS-rewriting has been widely studied; Alarcón, Gutiérrez, and Lucas proposed in 2006 a first definition of dependency pairs that is well-suited to context-sensitive systems [3, 4].

Section II.4.2 summarises my work with Gutiérrez and Lucas on termination techniques for CS-rewriting, based on recent modular approaches. We proposed in [82] a notion of *usable rules* [73, 83] for CS-rewriting. A straightforward (and cheaper) adaptation of usable rules for CS-dependency pairs is shown to be unsound; we nonetheless characterise a class of CS-systems for which this direct efficient version can be applied safely. These techniques are implemented in MU-TERM [2].

MAUDE programs enjoy even more features. From the OBJ family, MAUDE is based on membership equational logic (MEL). In short, specifications and programs in MAUDE are written using context-sensitive rules, with conditions, and involving membership constraints. I took the opportunity offered by a bilateral contract (2003–2005) between University of Illinois at Urbana-Champaign (UIUC) and CNRS to collaborate on termination proof techniques for membership equational logic programs. Note that in this context, standard termination notions are not satisfactory: a MAUDE interpretor may 'loop' on the condition of a rule that otherwise would terminate. For this reason, we consider the notion of operational termination [119]. The idea, described in Section II.4.3, is to use in back-end the power of available automated termination provers by translating any MEL program into a suitable input for such provers, which seldom support such a combination of expressive features. To this end, Durán, Lucas, Marché, Meseguer, and I defined two transformations, so as to remove successively membership constraints and conditions from the original program [54, 55]. The first transformation eliminates sorts and memberships, resulting in an unsorted context-sensitive and conditional rewrite theory. The second transformation, extending previous works by Ohlebusch [139], and Marchiori [126], eliminates conditions for context-sensitive systems. This last translation step is shown to be incomplete.

We prove that both transformations are sound, that is termination of the resulting rewriting system implies operational termination of the original program. These techniques are implemented in MTT, a termination tool for MAUDE programs, which uses external provers like C*i*ME, APROVE [71], MU-TERM, etc.

From theory to practice

Putting theory into practice, all the aforementioned techniques have been implemented in different tools (though I am one of the main programmers of C*i*ME, I did not contribute personally to the implementation of MTT or MU-TERM).

Polynomial ordering techniques, hierarchical termination methods and their extensions to AC-rewriting were in particular integrated into the termination engine of C*i*ME 2, of which I am a main author.

Designed for automation, our techniques proved to be very efficient. As a matter of fact, all termination proofs throughout this thesis have been obtained using our tools. The hierarchical approach allowed C*i*ME to be the first tool able to prove termination for systems consisting of hundreds of rules, with proof discovery times of a few seconds (systems coming from μ-CRL specifications of communicating processes). We obtained in particular, with full automation, the first known proof of termination for some AC-systems of the literature, the termination of which was conjectured. These achievements contributed to the growth of activity in this area, and motivated the creation of a competition for automated termination provers [124].

Mechanical Guarantees

All the tools mentioned above are large pieces of software, C*i*ME 2 gathers about 60 000 lines of Objective Caml code, and they are prone to bugs themselves. The recent history of the termination competition showed that incorrect 'proofs' could be returned. It was usually because of a small typing error in the code but still the question is raised: to which extend can we trust automated provers?

Being confident in an automated prover is one first concern. Using a delegated prover to add automation to a proof assistant, for example to get rid of technically difficult proof steps in a development, is another one. Skeptical proof assistants [151, 135] cannot indeed take for granted the answers of external procedures: they need a formal proof of the handled properties. Hence, in addition to developing powerful techniques to prove termination of more and more relations with full automation, there is an important need for formal (mechanical) certificates for these relations' termination proofs, in order to enable their definition and their use in skeptical assistants.

In 2005 the French National Agency for Scientific Research (ANR) launched a call for proposals and gave me the opportunity to coordinate a project aimed to add automation to proof assistants: A3PAT, of which I was principal investigator. The spectrum of the project[1] was broad enough to cover several aspects of this topic, not limited to termination; its funds allowed three post-doc students to be recruited, amongst whom two were under my supervision.

Chapter III is dedicated to the approach developed in this project [32], and some of its results. The A3PAT project is based on satellite tools: it aims to bridge the gap between proof assistants yielding formal guarantees of reliability, and highly automated tools one has to trust.

The general architecture of the approach is described in Section III.2. It involves an evolution of C*i*ME, that produces traces for the proofs it discovers (termination of course [32, 31] but also unification/disunification [33]). This new version features a trace compiler, targeted to the COQ proof assistant. It generates COQ scripts that refer to a formal library, COCCINELLE, developed in the project, and which contains definitions, structures and theorems about rewriting and termination. The COQ proof assistant may then check the proofs. Note that our approach mixes shallow and deep embeddings that is, respectively, direct use of the proof assistant's own structure of terms (thus using built-in proof mechanisms, induction, inversion, etc.), and complete encoding of term algebras within the proof assistant (thus applying our own efficient/generic techniques).

A challenging topic here is the definition of an adequate efficient trace language. It must be kept of low verbosity while being expressive enough for proofs of different properties, and if possible not related to a single prover, so as to allow certification of other provers' results. A brief description of our choices is in Section III.2.4.

Proving termination can amount to dealing with properties on data structures that may require heavy computations and particularly involved algorithmics. This is in particular the case for proofs on graphs. Efficient ways of proving relevant properties on these structures have to be devised so as to overcome the potential difficulties. To this end, we proposed in 2008 a new formalisation for graphs [40], that allows certification of (termination) proofs based on a graph analysis, and that can manage efficiently graphs containing thousands of arcs. This approach is discussed in Section III.3.

Finally, an interesting benefit of abstract formalisations is that keeping them as general as possible may lead to relaxing premises of theorems. An illustration is an extension of the *subterm criterion* [31] for termination of rewriting systems, which we proposed recently while giving its first full formalisation. This is the version of the criterion that is described Section II.1.4. Another example is a new notion of matrix interpretations [42] developed within the project.

Proof of Imperative Programs

A post-doc position at INRIA gave me the opportunity to experience a different point of view on program verification, in the context of a widespread mainstream imperative language. I shall sketch in Conclusion our experience with the KRAKATOA tool [121] and its initial development.

KRAKATOA is a tool that translates a Java/Javacard program (hence imperative) annotated in JML into a program with the same semantics but in the language of the WHY tool, a verification conditions (VC) generator. The VC produced, once proved, allow one to guaranty the correctness of the original program with reference to its specification (annotations). We use notably a memory model *à la* Burstall-Bornat. Amongst the first tools dedicated to formal verification of programs written in widely used languages, KRAKATOA has been successfully applied to the study of Javacard applets provided by industrial partners (Gemalto, etc.). The original approach served as a basis for other verification tools, and was adapted to the analysis of programs written in C (Caduceus).

[1] `http://a3pat.ensiie.fr`

This dissertation

This dissertation describes and illustrates my contributions on automated and mechanical proof. Some technical details and proofs of theorems are omitted. Most of the presented definitions and theorems appeared with their proofs in the relevant articles, to which I shall refer, as well as in the course material for the lecture on *Automated Reasoning* I have been giving at MPRI[2].

[2] `http://mpri.master.univ-paris7.fr/`

Chapter II

Automated Proofs of Termination

II.1 Termination of Rewriting, Overview

II.1.1 Preliminaries and Notations

I suppose that the reader is familiar with basic notions of rewriting and refer to surveys [52, 11, 139] for technical details. I shall however give some notations.

A *signature* $\mathcal{F}$ is a set of *symbols* (with arities). Let X be a set of *variables*; $T(\mathcal{F}, X)$ denotes the set of finite *terms* on $\mathcal{F}$ and X. The root position in a term is denoted by Λ, the symbol at root position in a term t is denoted by $\Lambda(t)$, $t|_p$ denotes the subterm of t at position p. A *context* $C[]_p$ is a term with a special constant 'hole' at position p, $C[s]_p$ denotes the replacement of the hole at p with a term s.

A *substitution* is a mapping σ from variables to terms. We use postfix notation for substitution applications. A binary relation $\mathscr{R}$ is said to be *stable* if $s\sigma \mathscr{R} t\sigma$ for every substitution σ whenever $s \mathscr{R} t$; it is said to be *monotonic* (denoted by $\mathsf{M}(\mathscr{R})$) if $C[s] \mathscr{R} C[t]$ for every context C whenever $s \mathscr{R} t$. The *accessibility* predicate over a relation and a term is inductively defined as follows: an element t is said to be *accessible* for $\mathscr{R}$ if all its predecessors (that is all s such that $s \mathscr{R} t$) are accessible. The relation $\mathscr{R}$ is said to be *well-founded* (denoted by $\mathsf{WF}(\mathscr{R})$) if all elements in its support are accessible, that is if there is no infinite sequence $x_1 \mathscr{R}^{-1} x_2 \mathscr{R}^{-1} \cdots$

A *term rewriting system* (TRS for short) over a signature $\mathcal{F}$ and a set of variables X is a set R of *rewrite rules* written $l \to r$. A term rewriting system R defines a *rewrite relation* $\to_R$ the following way: $s \to_R t$ (which constitutes a *rewriting step*) 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. We say then that *s reduces to t at position p with $l \to r$* or, if the rule of R is not relevant, *with* R, respectively denoted by $s \xrightarrow[l\to r]{p} t$ and $s \xrightarrow[R]{p} t$. That is: the rewriting relation is the monotonic and stable closure of R.

Following Dershowitz, I shall say that a term is *mortal* (for a given relation) if it cannot be source of infinite sequences of rewriting steps, otherwise it is said to be *immortal*. A rewriting relation $\to$ defined by a system R is said to be *terminating* (or that it *terminates*) if all terms are mortal, (written $\mathsf{SN}(\to)$)[1], that is when $\mathsf{WF}(\leftarrow)$. I shall override this terminology for R, that is denote by $\mathsf{SN}(R)$ and state that R terminates, when $\mathsf{SN}(\to_R)$.

The notations $\mathsf{SN}(\to)$ and $\mathsf{WF}(\leftarrow)$ are both kept to emphasise that inductive *and* computational aspects of a relation are addressed here, in particular regarding their use in a proof assistant (Sec. III.2.1).

II.1.2 Generalities

Termination of rewriting is an undecidable property as it is closely related to the Turing machine halting problem. This is shown firstly by Huet & Lankford in 1978 [90]; they give a simple translation from configurations of a Turing machine into *words*, and they model the behaviour of the machine by an unbounded number of rewrite rules. In 1987, this number is kept fixed (and small) by Dershowitz [48].

[1] SN stands for Strongly Normalising.

Finally, Dauchet proposes in 1989 a correspondence between the behaviour of a Turing machine and a term rewriting system, left linear, non-overlapping, and consisting of only *one* rule [43].

The problem being undecidable even for systems with only one rule, the techniques developed and implemented to show termination are sound, of course, but they will always be incomplete. The user is eventually given a termination argument, a non-termination argument, or a "do not know" statement.

Intuitively, from the automation point of view, a proof of termination is thus always difficult to discover. It might be easy to see that $f(f(x)) \rightarrow f(x)$ terminates; it requires a little more work to show termination of $f(a,b,x) \rightarrow f(x,x,x)$. In practice, it is not uncommon to face systems consisting of hundreds (if not thousands [56]) of rules, possibly involving equational theories. Moreover, size is not all that matters: termination of a small and simple system like $\{a(a(x)) \rightarrow b(c(x)),\ b(b(x)) \rightarrow a(c(x)),\ c(c(x)) \rightarrow a(b(x))\}$ has been shown only recently [86, 87] with original ordering techniques. Finally, termination of the Syracuse sequence[2], also known as Collatz's conjecture, is still a challenging... conjecture.

II.1.3 Problem Transformations, Criteria, Orderings

A termination proof generally consists in transforming termination problems into (sets of) problems that are equivalent with reference to termination but easier to solve, eventually reaching empty problems that close the proof. Sound and complete transformations are called *criteria*. Being transformations of problems, it has been noticed that criteria can be expressed in a uniform setting, as we noted in [32], and as Thiemann *et al.* independently pictured as *processors* [153, 73]. I shall follow the idea we introduced [32, 40], and model a termination proof by an *inference tree* where inference rules are criteria possibly guarded by conditions (such as the existence of an adequate well-founded ordering, inclusion properties over relations, etc.). The general form of a rule is the following:

$$\text{RULE NAME}(\text{PARAM})\ \frac{p_1 \ldots p_n}{p}\ \text{CONDITIONS}$$

where p, $p_1 \ldots p_n$ are properties on relations, and PARAM is an (optional) parameter of the rule.

A First Criterion

The fundamental operation is actually to mimic steps in the initial relation by steps in another relation that is known to be terminating.

Obviously, if the inverse rewriting relation is included into a well-founded ordering, the relation terminates. However, checking this embedding directly may require to test an infinite number of pairs, and is not convenient at all in practice and for automation. One may think of an easier test by checking that the rules of the system only (and not the whole relation) is embedded into the inverse of a well-founded ordering that is stable by the very same operations extending the system to the relation: monotonicity and instantiation.

The historical Manna and Ness criterion [120] consists in discovering a well-founded, monotonic and stable ordering, for which *each rule* of the system decreases strictly, thus embedding the relation in the well-founded ordering and ensuring its termination. In terms of inference rules, the criterion is:

$$\mathsf{MN}(>)\ \frac{}{\mathsf{SN}(\rightarrow_S)}\quad \mathsf{WF}(<) \wedge \mathsf{M}(<) \wedge (S \subseteq >)$$

Intuitively, the ordering strictly controls every single step of the relation[3].

[2] A Syracuse sequence for an integer $u_n > 1$ is defined as $u_{n+1} = \frac{u_n}{2}$ if u_n is even, and $u_{n+1} = 3u_n + 1$ if u_n is odd. It is conjectured that sequences are finite for all initial u_0.

[3] The orderings that are suitable to prove termination of a system give in particular information on maximal derivation length of the relation, see Section II.2.1.

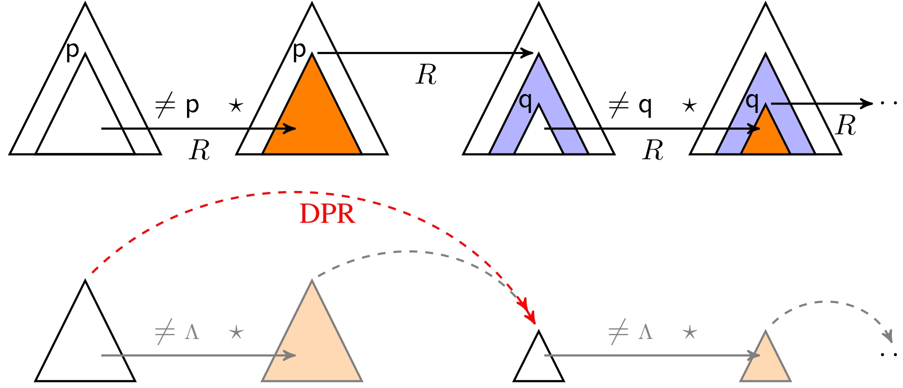


Figure II.1. Chain of recursive calls (at positions p then q) and corresponding dependency chain DPR.

II.1.4 Dependency Pairs, Dependency Graphs

The aforementioned method is in fact too restrictive in the sense that it requires monotonicity over suitable orderings. A breakthrough occurs in 1997 when Arts & Giesl introduce a powerful technique, based on a fine analysis of immortal terms: the *dependency pairs* (DP) approach [8, 9]. In contrast to criterion MN, the dependency pair approach focuses on the possible *inner recursive calls* of rules (the so-called dependency pairs). This leads in particular to a weakening of the constraints on suitable orderings.

Dependency Pairs

Let us partition the signature into two subsets: the set of *defined* symbols, occurring at Λ in left-hand sides of rules, which intuitively represent functions, and the set of *constructors*, which intuitively represent data. A recursive call involves a function, that is a defined symbol.

Definition 1 (Dependency pairs [9]) *The set of* unmarked dependency pairs *of a rewriting system R, denoted by $\mathsf{DP}(R)$, is defined as* $\mathsf{DP}(R) = \{\langle u, v\rangle \mid u \to t \in R$ *and* $t|_p = v$ *with* $\Lambda(v)$ *defined*$\}$.

Distinguishing root symbols of dependency pairs (by means of marks, or 'tuple-symbols') emphasises termination-relevant steps and enhances this technique significantly. Let us denote by $\widehat{\mathcal{F}}$ a copy of $\mathcal{F}$ where every symbol is now labelled with a mark. A *marked* dependency pair is simply[4] a pair $\langle u, v\rangle$ of terms in $T(\mathcal{F} \cup \widehat{\mathcal{F}}, X)$ defined as in Definition 1 where symbols at Λ in both members have been replaced with their marked copy. It will be clear from the context if $\mathsf{DP}(R)$ is the set of marked or unmarked pairs, as considering marks is not a dramatic change in the proof of relevant theorems, with the noticeable exception of rewriting modulo equational theories (see Section II.4.1).

Given a system R and a set D of pairs, we note $s \twoheadrightarrow_{D,R} t$ if and only if $s \xrightarrow[R]{\neq\Lambda\,\star} u\sigma \xrightarrow[\langle u,v\rangle\in D]{\Lambda} v\sigma \equiv t$. A *dependency chain* (of D over R) is a sequence in $\twoheadrightarrow_{D,R}$. It is said to be *minimal* if all proper subterms of every terms in this sequence are mortal. The link between recursive calls and dependency chains is illustrated on Figure II.1.

For both marked and unmarked case, the dependency pair criterion [9] can be rephrased as follows:

Theorem 1 (Arts & Giesl [9]) *For a rewriting system R,* $\mathsf{SN}(\twoheadrightarrow_{\mathsf{DP}(R),R})$ *is equivalent to* $\mathsf{SN}(\to_R)$.

[4]It is simple in the case of a free algebra, but we will see that it gets much more intricate when equational theories are involved.

We obtain the rule DP $$\text{DP}\ \frac{\mathsf{SN}(\twoheadrightarrow_{DP(S),S})}{\mathsf{SN}(\rightarrow_S)}$$

It is left to prove that dependency chains are terminating. The absence of infinite (minimal) dependency chains may be proven directly with the use of relevant orderings, see Theorem 5. Amongst the gains: the control over rules is not necessarily strict, the control over DP steps has to be strict but is not required to be monotonic.

Example 1 *Consider this system R from [8], modelling division for Peano integers:*

$$\begin{array}{lcllcl} x-0 & \rightarrow & x & s(x)-s(y) & \rightarrow & x-y \\ 0 \div s(y) & \rightarrow & 0 & s(x)\div s(y) & \rightarrow & s((x-y)\div s(y)) \end{array}$$

The set of defined symbols contains two elements: $\{-,\div\}$, the set $\mathsf{DP}(R)$ has three dependency pairs:

$$(1)\ \langle s(x)\widehat{-}s(y), x\widehat{-}y\rangle \qquad (2)\ \langle s(x)\widehat{\div}s(y), (x-y)\widehat{\div}s(y)\rangle \qquad (3)\ \langle s(x)\widehat{\div}s(y), x\widehat{-}y\rangle$$

To prove termination of R, it is sufficient to prove that $\twoheadrightarrow_{\mathsf{DP}(R),R}$ terminates.

Remark 1 *It is sufficient to prove that only minimal chains terminate. Dershowitz remarked [51] that some pairs were irrelevant in that case: in particular, pairs $\langle \widehat{l}, \widehat{r}\rangle$ coming from a rule $l \rightarrow C[r]$ where r is a proper subterm of l. For example, the pair $\langle \widehat{f}(f(x)), \widehat{f}(x)\rangle$ from a rule $f(f(x)) \rightarrow f(g(f(x)))$ can be discarded. Dershowitz's improvement is now part of the classical definition of dependency pairs, see for instance [83].*

Very well-suited for automation, this approach has been made even more powerful by use of multiple refinements: for example it can benefit greatly from the analysis of a *dependency graph*.

Dependency Graphs

We restrict now to systems that consist of a finite number of rules. As noticed in [9], not all DP can follow another in a dependency chain: one may consider the graph of possible sequences of DP. This graph is finite as long as we restrict to *finite systems*, and each dependency chain corresponds to a *path* in this graph.

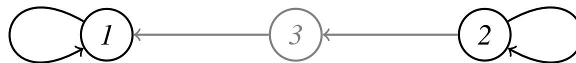
Definition 2 (Dependency graph [9]) *Let R be a rewriting system. The* dependency graph of a set of dependency pairs D over a system R, *denoted by $\mathcal{G}(D,R)$, is defined as (D,A) where $\langle s,t\rangle \mapsto \langle s',t'\rangle \in A$ if and only if there exists a substitution σ such that $t\sigma \xrightarrow[R]{\neq\Lambda\ \star} s'\sigma$.*

Therefore, if there is no infinite path *corresponding to a dependency chain* in the graph, then there is no infinite dependency chain. For finite systems, infinite dependency chains can only correspond to cycles of the finite graph. The fundamental theorem of graph refinement is as follows:

Theorem 2 (Giesl *et al.* [69]) *A system R terminates if and only if for each circuit C in the dependency graph $\mathcal{G}(\mathsf{DP}(R),R)$, the corresponding relation $\twoheadrightarrow_{C,R}$ is terminating*

Note that it is not sufficient to consider *elementary* circuits only, see [10] for a counter-example.

Example 2 *From the dependency pairs of Example 1 one can build the following dependency graph:*



We may note that the dependency pair (3) is not on any circuit of the dependency graph. Pairs (1) and (2) belong to one circuit each. By Theorem 2, termination of R is equivalent to termination of both relations $\twoheadrightarrow_{(1),R}$ and $\twoheadrightarrow_{(2),R}$. Pair (3) does not have to be considered.

The relevant criterion is the following GRAPH rule:

$$\textsc{Graph}\ \frac{\mathsf{SN}(\twoheadrightarrow_{D_1,S}) \quad \dots \quad \mathsf{SN}(\twoheadrightarrow_{D_n,S})}{\mathsf{SN}(\twoheadrightarrow_{D,S})}\ \mathsf{Decomp}(D, \{D_1, \dots, D_n\})$$

It states that one can consider any decomposition $\{D_1, \dots, D_n\} \subseteq \mathscr{P}(D)$ of the set of pairs D that is compatible with the topology of strongly connected components, and then be left with termination problems for the corresponding $\twoheadrightarrow_{D_i,S}$, each of which being potentially simpler than the original one. Amongst the several notions of a compatible decomposition, the one that is most used is the decomposition into SCC. Sets D_i without any circuit lead to trivially terminating $\twoheadrightarrow_{D_i,R}$ and, thus, can be omitted.

This theorem is not applied *as is* in the practice of discovering termination proofs. A decomposition into strongly connected components (SCC) (or any other compatible decomposition) allows to search for some *uniform* arguments for some classes of circuits (see Hirokawa and Middeldorp [83]), thus preventing any exponential blow-up. This is in particular how we will certify graph criteria (see [40], and Section III.3). The efficient technique of [83] is to prove at once that all circuits in an SCC *involving a selected pair* correspond to terminating rewritings, to recompute a compatible decomposition on what remains when the selected pair is removed, and to prove recursively that relevant rewritings terminate (that is to apply Criterion GRAPH on the remaining set of pairs). This can be represented by a rule that has to be instantiated by relevant pairs-pruning arguments :

$$\textsc{RmVertex}(arg)\ \frac{\mathsf{SN}(\twoheadrightarrow_{D\setminus\langle u,v\rangle,S})}{\mathsf{SN}(\twoheadrightarrow_{D,S})}\ \text{guards for } arg \text{ (removing } \langle u, v\rangle)$$

A method for proving termination is thus to apply recursively GRAPH in conjunction with RMVERTEX. Of course the argument of rule RMVERTEX may be sufficient to remove several pairs at once, and I shall denote it by RMVERTEX*.

Remark 2 *It is worth noticing that the analysis of graph allows the use of different techniques and orderings for each subset of dependency pairs [69].*

The dependency graph $\mathcal{G}(D, R)$ is not computable in general since knowing if a σ such that $s\sigma \to t\sigma$ exists is not decidable. As a consequence, one uses an *approximated* graph, that is a graph $(D, A' \supseteq A)$ that contains it.

Arts & Giesl propose in 1997 a simple and quite efficient technique based on *connectability* to build an approximated graph [9]. The technique and the resulting graph are usually referred to as EDG. The principle is the following: reductions between DP steps, that is rewriting steps that (potentially) occur in proper subterms, must leave the constructor cap untouched. One may however consider that subterms with a defined symbol at root can be rewritten into anything (a good deal of the approximation is here).

Definition 3 (EDG [9]) *Let R be a rewriting system over $T(\mathcal{F}, X)$. For all $t \in T(\mathcal{F}, X)$,*

- $\textsc{cap}(t)$ *is the term t where all proper subterms with a defined symbol at Λ are replaced with fresh variables,*
- $\textsc{ren}(t)$ *is the term t where each occurrence of each variable is replaced with a fresh one*[5].

[5] A term may indeed be rewritten into two different reducts. Thus, several instances of a term must have their reducts abstracted with different variables.

Term t is said to be connectable *to term t' if* $\text{REN}(\text{CAP}(t))$ *and t' unify.*

Arts & Giesl prove that connectability defines a sound approximation of the dependency graph:

Lemma 3 *Let R be a rewriting system, if there is a σ such that $t\sigma \xrightarrow[R]{\neq\Lambda\ *} t'\sigma$, then t is connectable to t'.*

This approximation is quite powerful in practice, and preserves good properties such as CE termination (see Section II.3).

There are many other approximations, of various complexity, and not always comparable. For example, simply taking into account the fact that proper subterms do not rewrite to anything but to instances of the right-hand sides of rules, one obtains EDG* approximation [130]. Some other techniques make use of tree automata [129], etc.

Termination of $\twoheadrightarrow_{D,R}$: Subterm

As usual, termination of dependency chains can be proven directly by embedding the original relation into another one that is proven to be terminating. This can be done by construction, for instance with the *subterm criterion* [83] for which we proposed in 2010 an extension [31], as well as giving its first certification[6] (see Chapter III), as part of a work with Contejean, Courtieu, Forest, Paskevich and Pons from the A3PAT team.

The classical idea behind the subterm criterion is that some dependency chains cannot be minimal. Similarly to Hirokawa and Middeldorp [83], we define a projection over terms, based on a mapping $\mathfrak{p}$ from function symbols to natural numbers. The projection is also denoted by $\mathfrak{p}$. In contrast to [83], we do not require that $\mathfrak{p}(f)$ should be less than the arity of f. Our projection is defined as follows:

$$\mathfrak{p}(x) = x$$
$$\mathfrak{p}(f(t_1,\ldots,t_n)) = \begin{cases} t_{\mathfrak{p}(f)} & \text{if } 1 \leq \mathfrak{p}(f) \leq n \\ f(t_1,\ldots,t_n) & \text{otherwise} \end{cases}$$

Theorem 4 (Contejean *et al.* [31]) *Let R be a TRS, and $D = D' \cup \langle u_0, v_0\rangle$ be a subset of $\mathsf{DP}(R)$. If there exists a projection based on $\mathfrak{p}$ such that*

- $\mathfrak{p}(v_0)$ *is a proper subterm of v_0,*
- $\mathfrak{p}(u_0)(\rightarrow_R \cup \rhd)^+\mathfrak{p}(v_0)$,
- *for every pair $\langle u, v\rangle$ of D', $\mathfrak{p}(u)(\rightarrow_R \cup \rhd)^\star\mathfrak{p}(v)$,*

then, $\twoheadrightarrow_{D,R}$ minimal wrt R is terminating if $\twoheadrightarrow_{D',R}$ minimal wrt R is terminating.

One obtains then the following instance:

$$\text{RMVERTEX}(\mathfrak{p})\ \frac{\mathsf{SN}(\twoheadrightarrow_{D\setminus\langle u_0,v_0\rangle,S})}{\mathsf{SN}(\twoheadrightarrow_{D,S})}\ \begin{array}{l}\mathfrak{p}(v_0) \lhd v_0 \\ \mathfrak{p}(u_0)(\rightarrow_R \cup \rhd)^+\mathfrak{p}(v_0) \\ \mathfrak{p}(u)(\rightarrow_R \cup \rhd)^\star\mathfrak{p}(v),\ \forall\langle u,v\rangle \in D\end{array}$$

Our statement is more general than the one given in [83], since we consider a less restrictive projection and the relation $\rightarrow_R \cup \rhd$ instead of $\rhd$, but our proof follows the same line. We remark that if $s \twoheadrightarrow_{\langle u,v\rangle,R} t$ for a pair $\langle u, v\rangle$ of D' which is strongly connected to $\langle u_0, v_0\rangle$, then $\mathfrak{p}(s)(\rightarrow_R \cup \rhd)^\star\mathfrak{p}(t)$, and that if $s \twoheadrightarrow_{\langle u_0,v_0\rangle,R} t$, then $\mathfrak{p}(s)(\rightarrow_R \cup \rhd)^+\mathfrak{p}(t)$. Thus we mimic the original sequence of steps by another sequence in a terminating relation.

[6] The formalisation we proposed is for the COQ proof assistant. Another formalisation by the IsaFoR team was obtained independently for ISABELLE/HOL, and for the plain historical criterion only.

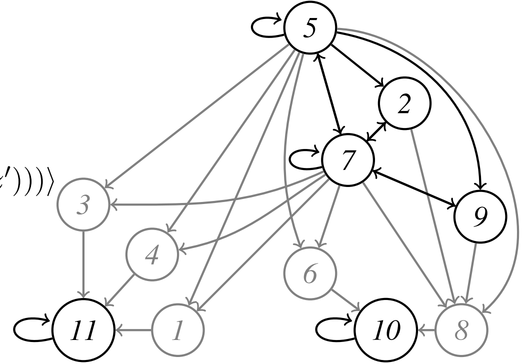
Quite simple to automate and computationally frugal, the historical subterm criterion of [83] is already impressive by its efficiency in practice. Regarding our generalised subterm criterion, it may apply on systems where some redundant rules are added in order to speed up computations. For instance, it proves useful on the following system where the original subterm criterion fails.

Example 3 *This system is a slight modification of the one in [83] that computes the list of all integers between two given bounds, in increasing order. The last rules does not change the underlying equational theory of the rest of the TRS; they are, in fact, expressible as a composition of two other rules. However, such 'derived' rules naturally appear in the practice of program optimisation, like deforestation.*

$$
\begin{array}{lcl}
x+0 & \rightarrow & x \\
x+s(y) & \rightarrow & s(x+y) \\
incrlist(nil) & \rightarrow & nil \\
incrlist(x::l) & \rightarrow & s(x)::(incrlist(l)) \\
btw(0,0) & \rightarrow & 0::nil \\
btw(0,s(y)) & \rightarrow & 0::(btw(s(0),s(y))) \\
btw(s(x),0) & \rightarrow & nil \\
btw(s(x),s(y)) & \rightarrow & incrlist(btw(x,y)) \\
btw(x+s(z),y+s(z')) & \rightarrow & incrlist(btw((x+z),(y+z'))) \\
btw(0,x+s(y)) & \rightarrow & 0::btw(s(0),s(x+y))
\end{array}
$$

This system admits 11 dependency pairs, building the following estimated graph:

$1:\ \langle btw(0,x+s(y)),x+y\rangle$
$2:\ \langle btw(0,x+s(y))),btw(s(0),s(x+y))\rangle$
$3:\ \langle btw(x+s(z),y+s(z')),y+z'\rangle$
$4:\ \langle btw(x+s(z),y+s(z')),x+z\rangle$
$5:\ \langle btw(x+s(z),y+s(z')),btw((x+z),(y+z'))\rangle$
$6:\ \langle btw(x+s(z),y+s(z')),incrlist(btw((x+z),(y+z')))\rangle$
$7:\ \langle btw(s(x),s(y)),btw(x,y)\rangle$
$8:\ \langle btw(s(x),s(y)),incrlist(btw(x,y))\rangle$
$9:\ \langle btw(0,s(y)),btw(s(0),s(y))\rangle$
$10:\ \langle incrlist(x::l),incrlist(l)\rangle$
$11:\ \langle x+s(y),x+y\rangle$



The pair 5: $\langle btw(x+s(z),y+s(z')),btw((x+z),(y+z'))\rangle$ *can be discarded if one uses a projection of* btw *on its second argument, since one can rewrite* $y+s(z')$ *into* $s(y+z')$ *which admits* $y+z'$ *as a proper subterm.*

The pair 2: $\langle btw(0,x+s(y)),btw(s(0),s(x+y))\rangle$ *can be discarded with the same projection since one has the reduction* $x+s(y)\underset{R}{\rightarrow}s(x+y)=s(x+y)$ *(note that there is no use of* $\rhd$ *to conclude on this last pruning).*

Applications of Theorem 4 alone are actually sufficient for a termination proof of this example.

Termination of $\twoheadrightarrow_{D,R}$: Ordering Pairs

Termination can also be proven with the help of *ordering pairs*. As relation $\twoheadrightarrow_{D,R}$ does not preserve context, strict monotonicity is not required for an ordering that contains the (inverse) relation. It is sufficient to control that (non monotonic) DP steps decrease and that (monotonic) rewriting steps do not increase, as illustrated on Figure II.2. Ordering pairs are defined to this end.

A very general definition of ordering pairs may be found in [105]. Due to our definition of $\twoheadrightarrow_{D,R}$, we use a slightly restricted notion.

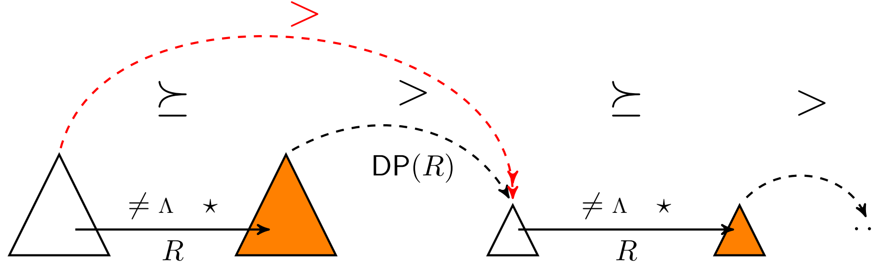


Figure II.2. Control of the dependency chain using an ordering pair $(\preceq, <)$.

Definition 4 (Ordering pair) *An* ordering pair *is a pair* $(\preceq, <)$ *of relations over* $T(\mathcal{F}, X)$*, stable*[7]*, and such that:*

- $\preceq$ *is reflexive,* $<$ *is irreflexive, and*
- $< \cdot \preceq \subseteq <$.

An ordering pair $(\preceq, <)$ *is* well-founded *if* $<$ *is well-founded; overriding the notation this is denoted by* $\mathsf{WF}(\preceq, <)$*. It is said to be* weakly monotonic *if* $\preceq$ *is monotonic, denoted by* $\mathsf{WM}(\preceq, <)$*, and* strictly monotonic *if* $<$ *is monotonic.*

Rephrasing [9], and coming back to the definition of $\twoheadrightarrow_{D,R}$, an effective corollary of Theorem 1 consists in discovering a well-founded weakly monotonic ordering pair $\preceq, <$ such that rewriting steps decrease for $\succeq$ and dependency pairs steps decrease for $>$ to prove that $\twoheadrightarrow_{\mathsf{DP}(R),R}$ terminates, by inclusion in $>$.

Corollary 5 *If there is a well-founded weakly monotonic ordering pair* $(\preceq, <)$ *for which:* $l \succeq r$ *for all* $l \to r \in R$*, and* $u > v$ *for all* $\langle u, v \rangle \in D$*, then* $\twoheadrightarrow_{\mathsf{DP}(R),R}$ *is terminating.*

We may define the relevant inference rule:

$$\mathsf{DP}_{\subseteq}(\preceq, <)\ \frac{}{\mathsf{SN}(\twoheadrightarrow_{D,R})}\ \begin{array}{l}\mathsf{WF}(\preceq, <) \wedge \mathsf{WM}(\preceq, <)\\ \wedge (R \subseteq \succeq) \wedge (D \subseteq >)\end{array}$$

Example 4 *A suitable ordering pair* $(\preceq_1, <_1)$ *must fulfil the following constraints in order to prove termination of* $\twoheadrightarrow_{(1),R}$ *in Example 2:*

$$s(x)\hat{-}s(y) >_1 x\hat{-}y \qquad \begin{array}{rcl} x - 0 & \succeq_1 & x \\ s(x) - s(y) & \succeq_1 & x - y \end{array} \qquad \begin{array}{rcl} 0 \div s(y) & \succeq_1 & 0 \\ s(x) \div s(y) & \succeq_1 & s((x - y) \div s(y)) \end{array}$$

Similarly, an ordering pair $(\preceq_2, <_2)$ *to prove termination of* $\twoheadrightarrow_{(2),R}$ *must fulfil:*

$$s(x)\hat{\div}s(y) >_2 (x - y)\hat{\div}s(y) \qquad \begin{array}{rcl} x - 0 & \succeq_2 & x \\ s(x) - s(y) & \succeq_2 & x - y \end{array} \qquad \begin{array}{rcl} 0 \div s(y) & \succeq_2 & 0 \\ s(x) \div s(y) & \succeq_2 & s((x - y) \div s(y)) \end{array}$$

Discovering adequate pairs $(\preceq_1, <_1)$ *and* $(\preceq_2, <_2)$ *suffices to prove termination of R. Note the use of several orderings as mentioned in Remark 2.*

It suffices to find an adequate ordering pair:

[7] The original definitions require those relations to be transitive, but it has been noticed that this is actually unnecessary.

Corollary 6 *If there is a well-founded weakly monotonic ordering pair $(\preceq, <)$ for which: $l \succeq r$ for all $l \to r \in R$ and $u \succeq v$ for all $\langle u, v\rangle \in D$, and $u_0 > v_0$ for a $\langle u_0, v_0\rangle$, then termination of $\twoheadrightarrow_{D\cup\{\langle u_0,v_0\rangle\},R}$ is equivalent to termination of $\twoheadrightarrow_{D,R}$.*

This effective corollary can be summarised in the following instance:

$$\textsc{RmVertex}(\preceq,<)\ \frac{\mathsf{SN}(\twoheadrightarrow_{D\backslash\langle u,v\rangle,S})}{\mathsf{SN}(\twoheadrightarrow_{D,S})}\quad \begin{array}{l}\mathsf{WF}(\preceq,<)\wedge\mathsf{WM}(\preceq,<)\wedge u>v\\ \wedge(S\subseteq\succeq)\wedge(D\backslash\langle u,v\rangle\subseteq\succeq)\end{array}$$

Remark 3 *Note that premises of Corollary 6 can actually be weakened by requiring an additional reflexive relation $\leq$ giving a* relation triple $(\preceq,\leq,<)$ *with $<\cdot\leq\,\subseteq\,<$ and $<\cdot\preceq\,\subseteq\,<$. Corollary 6 could then be rephrased as follows: if $l \succeq r$ for all $l \to r \in R$ and $u \geq v$ for all $\langle u, v\rangle \in D$, and $u_0 > v_0$ for a $\langle u_0, v_0\rangle$, then termination of $\twoheadrightarrow_{D\cup\{\langle u_0,v_0\rangle\},R}$ is equivalent to termination of $\twoheadrightarrow_{D,R}$. However, for the sake of readability, and to stay compatible with previous works, we will stick to orderings pairs throughout this thesis.*

An important point is that the constraints to be fulfilled by the embedding relations are much weaker than in the case of the Manna & Ness criterion, especially regarding monotonicity. We will see an interesting illustration with Example 6.

II.2 Orderings

Orderings to prove termination are classically divided in two classes: *syntactical* orderings and *semantical* orderings.

Syntactical orderings are built from a precedence on symbols which is extended to terms. Introduced independently by Manna, Dershowitz [53, 46] and Plaisted [142], syntactical path orderings rely on the idea that a term s is smaller than t if it is built from subterms smaller than those of t for the ordering, into a context made from symbols smaller for the precedence than those of t. One may compare the *multisets* of subterms, as in [53, 46, 142] thus defining Multiset Path Orderings (MPO). One may also compare them lexicographically as in Kamin & Levy [93] (LPO). Dershowitz proposes in 1982 a more general notion of path ordering, the Recursive Path Ordering [47] (RPO), which combines those two ways of comparing subterms, by adding to functions symbols a status that determines how the comparison of subterms may be performed (multiset, lexico left-right, lexico right-left, etc.). RPO is stable, monotonic, and it is well-founded whenever the precedence is well-founded.

Remark 4 *Historically, syntactical orderings are on purpose simplification orderings by construction. It is actually easier to prove that they are well-founded on finite signatures in that case, using Kruskal's Theorem. However this restriction to finite signatures is not compulsory: as stated above, RPO is well-founded provided that the precedence is well-founded, that is even when the signature is infinite*[8].

Transformation techniques like Recursive Program Schemes (RPS) and in particular Argument Filtering Systems (AFS) [9] allow one to lose monotonicity when applied prior to use an RPO (in particular by erasing arguments).

Regarding automation, syntactical orderings are easy to implement, and are decidable on finite signatures. In addition to the naive enumerative approach, several techniques have been devised to search efficiently for these orderings with full automation, from constraint resolution as in the pioneer tool REVE [109, 65], to the recent SAT-solver based methods [6]. For many years they have been the most widely used orderings, in particular for termination purposes.

[8] In particular, the COQ formalisation of RPO in COCCINELLE does not use any finiteness assumption on the signature, see Section III.

Remark 5 *An intermediate ordering is the Knuth-Bendix Ordering (KBO) [96], which uses a weight function (an extension to terms of a mapping from symbols to nonnegative integers) in conjunction with a compatible precedence. KBO is successfully used for (termination and) completion purposes.*

For more detailed surveys on various path orderings, see [147, 149].

Semantical orderings make use of an interpretation of terms in an ordered algebra. Many kinds of interpretation were studied amongst which natural numbers, ordinals, elementary interpretations [110], and *polynomial* interpretations [120, 107]. See [49] for examples. Recently, new kinds of interpretation were brought to the fore, and proved to be very successful in practice: matrix-based interpretations [57, 42], over arctic domains [97], rational or real [46, 117] numbers[9], etc.

I shall focus in this section on polynomial interpretations over nonnegative integers and their implementation, in particular on the contribution of a joint work, published in 2005, by Contejean, Marché, Tomás and myself [38].

II.2.1 Polynomial Interpretations

Several reasons have made, in the past, polynomial interpretations less popular than syntactical methods like RPO.

A theoretical limitation comes from the standard Manna-Ness criterion that dominated amongst termination techniques before 1997 and the dependency pair approach. As each single rewriting step decreases for the ordering, the complexity measure of the computed function and maximal length of derivation are captured by the ordering. In particular, the combination of polynomials with the Manna-Ness criterion puts strong restrictions on the class of relations the obtained ordering can contain: the length of derivations has a double exponential bound [85], and the computed function necessarily belongs to a restricted complexity class [16]. For instance, the termination of the famous Ackermann-Péter function can easily be proven using Manna-Ness and RPO, while it is impossible to obtain a polynomial interpretation suitable to that criterion[10].

From a practical point of view, precedence-based orderings are much easier to implement, and decidable, whereas the search for suitable polynomials is necessarily incomplete.

The dependency pair approach and its effective criteria changed the context. These new techniques demand much weaker properties on the underlying orderings used in termination proofs. In particular, monotonicity of the strict part of the ordering is not required. As a consequence, some orderings previously seen as less powerful than others (for termination proof using the Manna-Ness criterion) underwent a regain of interest. This is the case for polynomial orderings.

This new opportunity for polynomial interpretations-based orderings led us to design a new implementation of them inside the C*i*ME rewrite tool [36], and Contejean, Marché, Tomás and I proposed [38] efficient techniques:

- to check that a polynomial interpretation induces an ordering that proves termination of a given system, and
- to discover well-suited polynomial interpretations with full automation.

I shall briefly describe these techniques hereafter, full details can be found in the journal article in which they were published [38].

The techniques presented here brought to the fore how powerful polynomial-based orderings could be in practice. As a matter of fact, they are still widely used today. C*i*ME 2 has been a very efficient

[9]Note that polynomial interpretations over integers are subsumed neither by interpretation over rational numbers [118] nor by interpretations over real numbers [133].

[10]The situation is completely different with dependency pairs, as illustrated in Example 6.

solver for finding polynomial interpretations for years, and was also used as a back-end for other automated provers like MU-TERM [2, 116], or Cariboo [63]. However, the constraints solving techniques presented there have been recently replaced with SAT-solver targeted translations [66].

Domains and interpretations

Term orderings induced by polynomial interpretations in their present form have been defined in 1979 in a pioneer paper of Lankford [107].

Let D be an arbitrary non-empty domain equipped with some ordering $\geq_D$, and let $>_D$ be $\geq_D - \leq_D$.

Definition 5 *Let ϕ be a function that maps each ground term $t \in T(\mathcal{F})$ to an element of D. The relations $\succeq_\phi$ and $>_\phi$ generated by ϕ are defined by*

$$\begin{array}{lll} t_1 \succeq_\phi t_2 & \textit{if and only if} & \phi(t_1) \geq_D \phi(t_2) \\ t_1 >_\phi t_2 & \textit{if and only if} & \phi(t_1) >_D \phi(t_2) \end{array}$$

Lemma 7 *$(\preceq_\phi, <_\phi)$ is an ordering pair on ground terms. It is well-founded if $<_D$ is well-founded.*

To generalise this construction to non-ground terms, a natural way would be to define $t_1 \succeq_\phi t_2$ when $t_1\sigma \succeq_\phi t_2\sigma$ for any ground substitution σ. However, such a definition is not well suited for automation, and we proceed in a different way which leads to an almost equivalent but slightly restricted definition.

The idea is the following: we should not interpret a non-ground term into an element of D, but actually into an abstraction mapping any interpretation (or *valuation*) of its variables in D into an element in D. In other words, interpretation $\phi(t)$ of a non-ground term t is a function from $X \to D$ to D. This set $(X \to D) \to D$ of functions is naturally equipped with the ordering defined by

$$\begin{array}{lll} f \succeq_{D,X} g & \text{if and only if for all } \rho \in X \to D, & f(\rho) \geq_D g(\rho) \\ f >_{D,X} g & \text{if and only if for all } \rho \in X \to D, & f(\rho) >_D g(\rho) \end{array}$$

We should point out that $>_{D,X}$ is *not* $\succeq_{D,X} - \preceq_{D,X}$. For instance, if one maps a constant a to the minimal element of D then, for any variable x, $x \succeq_{D,X} a$ but $x \not\preceq_{D,X} a$. Hence $(x, a) \in \succeq_{D,X} - \preceq_{D,X}$ while $(x, a) \notin >_{D,X}$.

Automation of such an ordering only relies on the automation of this ordering on functions. We will see how to automate that ordering in the special case of D being a set of integers.

Definition 6 *Let ϕ be a function which maps each term $t \in T(\mathcal{F}, X)$ to a function from $X \to D$ to D. The relations $\succeq_\phi$ and $>_\phi$ generated by ϕ are defined by*

$$\begin{array}{lll} t_1 \succeq_\phi t_2 & \textit{if and only if} & \phi(t_1) \succeq_{D,X} \phi(t_2) \\ t_1 >_\phi t_2 & \textit{if and only if} & \phi(t_1) >_{D,X} \phi(t_2) \end{array}$$

Lemma 8 *$<_\phi \cdot \preceq_\phi \subseteq <_\phi$ on non-ground terms. $<_\phi$ is well-founded if $<_D$ is well-founded.*

Definition 7 *We define an* homomorphic interpretation *ϕ by giving, for each f of arity n, a function $[\![f]\!]_\phi$ from D^n to D, and then by induction on terms: for any $\rho \in X \to D$,*

$$\begin{array}{rcl} \phi(f(t_1, \ldots, t_n))(\rho) & = & [\![f]\!]_\phi(\phi(t_1)(\rho), \ldots, \phi(t_n)(\rho)) \\ \phi(x)(\rho) & = & \rho(x) \end{array}$$

For the sake of readability we write $[\![f]\!]$ if the relevant interpretation is clear from the context.

Lemma 9 *If ϕ is an homomorphic interpretation, then $(\preceq_\phi, <_\phi)$ is stable, hence an ordering pair.*

Lemma 10 *For any symbol f of arity n, and $1 \leq i \leq n$, if for all $d_1, \dots, d_{i-1}, d_{i+1}, \dots, d_n$ in D, $[\![f]\!](d_1, \dots, d_{i-1}, x, d_{i+1}, \dots, d_n)$ is monotonic (resp. strictly) nondecreasing in x then $\succeq_\phi$ (resp. $>_\phi$) is monotonic with reference to the i-th argument of f.*

In order to automate the search for interpretations, it is necessary to focus on a particular interpretation domain. The most convenient one is the set of integers. Since it is not well-ordered, we have in fact to consider a set of integers greater than or equal to a given minimum value μ.

Definition 8 *For a given $\mu \in \mathbb{Z}$, let $D_\mu = \{x \in \mathbb{Z} \mid x \geq \mu\}$. It is clear that the usual ordering $<$ is well-founded over each D_μ. Interpretations into D_μ are called* arithmetic*, they may be called μ-interpretations in order to specify the value of μ.*

An arithmetic homomorphic interpretation ϕ defined by functions $[\![f]\!]_\phi$, $f \in \mathcal{F}$, is called a polynomial *interpretation if for every f, $[\![f]\!]_\phi$ is a polynomial function.*

To check whether a given polynomial interpretation is suitable for proving termination of a given system, one must be able to check that:

1. each polynomial effectively maps D_μ^n into D_μ;

2. any of those polynomials is weakly and/or strictly increasing in some/all of its arguments.

Moreover, so as to perform comparisons we must be able to check that:

3. for any two terms t_1 and t_2, $t_1 \succeq_\phi t_2$ and/or $t_1 >_\phi t_2$.

Item (1) can be dealt with as follows: given a polynomial P with n variables, P effectively maps D_μ^n into D_μ if and only if polynomial $P - \mu$ is nonnegative on D_μ^n.

Item (2) can be dealt with as follows: given a polynomial P with n variables, P is nondecreasing in its i-th argument if and only if polynomial

$$\begin{aligned} Q(X_1, \dots, X_n) &= P(X_1, \dots, X_{i-1}, X_i + 1, X_{i+1}, \dots, X_n) \\ &\quad - P(X_1, \dots, X_{i-1}, X_i, X_{i+1}, \dots, X_n) \end{aligned}$$

is nonnegative on D_μ^n. Similarly, P is strictly increasing in its i-th argument if and only if polynomial

$$\begin{aligned} Q(X_1, \dots, X_n) &= P(X_1, \dots, X_{i-1}, X_i + 1, X_{i+1}, \dots, X_n) \\ &\quad - P(X_1, \dots, X_{i-1}, X_i, X_{i+1}, \dots, X_n) - 1 \end{aligned}$$

is nonnegative on D_μ^n.

Item (3) can be taken care of as follows: given t_1 and t_2, $\phi(t_1)$ and $\phi(t_2)$ can be computed as polynomials P_1 and P_2 over their variables, and then $t_1 \succeq_\phi t_2$ if and only if polynomial $P_1 - P_2$ is nonnegative on D_μ^n, and $t_1 >_\phi t_2$ if and only if polynomial $P_1 - P_2 - 1$ is nonnegative on D_μ^n.

We see that proving termination of TRS using a given polynomial μ-interpretation can be done automatically, as soon as one can check whether a given polynomial with n variables is nonnegative over D_μ^n.

Checking Positiveness

Given a polynomial $P \in \mathbb{Z}[X_1, \dots, X_n]$, how can one prove that $P(x_1, \dots, x_n) \geq 0$ for any value $x_i \geq \mu$? Checking whether a polynomial is nonnegative or not is far from being a simple task. This problem is undecidable in general since Hilbert's Tenth Problem can be reduced to it [127, 128].

Testing positiveness, in the context of automated termination proofs, has been studied by several authors [14, 68, 110, 143, 144, 148]. All their methods propose to approximate the problem by checking

positiveness for any *real* values greater than μ of their arguments, a problem that becomes decidable (Tarski 1930) but still algorithmically very complex. These authors proposed partial methods (i.e. correct and terminating but incomplete) supposed to be sufficient for an application to termination of TRS.

In 1998, Hong & Jakuš [88] make a comparison between some of these methods while proposing a new one: the *absolute positiveness* method. They prove it to be strictly more powerful than methods of Ben Cherifa & Lescanne [14] and Steinbach [148], and to be equivalent to Giesl's method [68] computing successive derivatives.

The methods of Rouyer [143, 144] and Lescanne [110] are not comparable with the absolute positiveness method. On a few examples, they succeed to prove positiveness of non-absolutely positive polynomials, such as $x^2 + y^2 - 2xy$. However, they do not propose any way to automate the search for polynomial interpretations. Since they do not offer a clear increment of power in practice, our method of choice will be the absolute-positiveness method.

Definition 9 *A polynomial P is said to be μ-*absolutely positive *if and only if polynomial $Q(X_1, \ldots, X_n) = P(X_1 + \mu, \ldots, X_n + \mu)$ has nonnegative coefficients only.*

This is not exactly the definition by Hong & Jakuš: they consider *strict* positiveness, which can be obtained by taking $Q - 1$ instead of Q in our definition. A straightforward sufficient condition for positiveness (adapted from Hong & Jakuš [88]) is the following:

Lemma 11 *If P is μ-absolutely positive, then it is nonnegative for all values in D_μ of its variables.*

This check seems to be nice and easy to perform. However, computing the 'translated' polynomial Q above can be costly in general. More precisely, Hong & Jakuš show that the algorithmic complexity of computing translations is the same as Giesl's method [68]. Nevertheless, this method is at least as powerful as the former ones while being quite simple.

We improve it a bit in our context by computing μ-translation in advance. If $\mu = 0$, testing absolute positiveness becomes completely trivial. Hence taking $\mu = 0$ as often as possible seems to be a good choice. We show in [38] that is always possible.

Lemma 12 *If $(\preceq_\phi, \prec_\phi)$ is an ordering pair defined by polynomial interpretations with a given μ, then this ordering pair can also be defined by some polynomial interpretations with $\mu = 0$.*

The gain is twofold. Regarding efficiency, μ-translation can now be performed once and for all, and not at each positiveness check. Regarding automated search for polynomials, setting $\mu = 0$ is enough.

Finally we give a simplified criterion for weak or strict monotonicity of the generated ordering when $\mu = 0$.

Lemma 13 *A polynomial 0-interpretation $P(x_1, \ldots, x_n)$ with nonnegative coefficients is always non-decreasing in each of its arguments. It is strictly increasing in its i-th argument if and only if there is a monomial ax_i^k with $a > 0$ and $k > 0$.*

In particular, there is no need of AFS to obtain a non-strictly monotonic ordering[11].

Example 5 *Orderings pairs $(\preceq_1, <_1)$ and $(\preceq_2, <_2)$ in Example 4 may be instantiated respectively with the following polynomial-based orderings:*

$$\begin{array}{rclrclrcl} [\![\widehat{-}]\!]_1(x,y) & = & x & [\![\div]\!]_1(x,y) & = & x & [\![s]\!]_1(x) & = & x+1 \\ & & & [\![-]\!]_1(x,y) & = & x & [\![0]\!]_1 & = & 0 \end{array}$$

[11] Which does not mean that AFS are useless *in a whole proof branch* closed by a polynomial ordering, as they can interact with usable rules (Section II.3) for example.

$$
\begin{array}{lllllllll}
[\![\hat{\div}]\!]_2(x,y) & = & x & [\![\div]\!]_2(x,y) & = & x & [\![s]\!]_2(x) & = & x+1 \\
 & & & [\![-]\!]_2(x,y) & = & x & [\![0]\!]_2 & = & 0
\end{array}
$$

The termination proof tree can now be closed (guards are omitted, as well as pairs outside any SCC):

$$
\textsc{Graph}\ \dfrac{\mathrm{DP}_{\subseteq}(\preceq_1,<_1)\ \dfrac{}{\mathsf{SN}(\twoheadrightarrow_{\langle s(x)\hat{-}s(y),x\hat{-}y\rangle,R})}\ \cdots\ \mathrm{DP}_{\subseteq}(\preceq_2,<_2)\ \dfrac{}{\mathsf{SN}(\twoheadrightarrow_{\langle s(x)\hat{\div}s(y),(x-y)\hat{\div}s(y)\rangle,R})}\ \cdots}{\mathrm{DP}\ \dfrac{\mathsf{SN}(\twoheadrightarrow_{\mathsf{DP}(R),R})}{\mathsf{SN}(\rightarrow_R)}}\ \mathsf{Decomp}(SCC\ldots)
$$

Note that in this case using rule RMVERTEX *is unnecessary.*

Discovering Suitable Interpretations with Full Automation

Firstly, to find a suitable interpretation automatically, we choose for each symbol of the signature a *parametric* polynomial, that is a polynomial where coefficients are variables the values of which have to be found. In order to have a finite number of such variables, we need to fix a bound on the degree of the polynomials we search for. In this respect, we follow Steinbach's classification [148], looking for linear, simple, simple-mixed polynomials[12].

Secondly, we have to translate each of the conditions that ensure suitability of the defined ordering (termination constraints) into constraints on these variables. We end up with non-linear Diophantine constraints on the coefficients introduced in the parametric interpretations.

Finally, we have to solve these constraints, a problem which is undecidable [127, 128].

The choice we made at that time, and which we implemented in C*i*ME 2, is to turn this problem into a decidable one by putting an arbitrary bound on the solutions we look for: we restrain the search for values of variables satisfying the constraints to a given interval $[0, B]$ where B is some nonnegative integer bound. The problem becomes then an instance of the so-called *finite domain constraint satisfaction* problems, which have been extensively studied in the literature, especially in the context of constraint logic programming [28, 91].

Translating into finite domain constraints is quite a well-known approach for *linear* Diophantine constraints [28] which is a major case in applications of constraint logic programming. However, the case of *non-linear* Diophantine constraints not being widely studied, we designed a specialised variant of finite domain constraints to fit our needs, and we developed original techniques for variable abstraction so as to keep constraints as small as possible, thus adding efficiency to the propagation procedure.

As already mentioned at the beginning of this section, provers nowadays use SAT-solving techniques to discover polynomial interpretations [66].

Example 6 *The following system R is a translation in first-order rewriting of a Prolog program computing the Ackermann-Péter function. The translation technique is the one used in the tool TALP [140] dedicated to termination of logic programs.*

$$
R = \left\{ \begin{array}{llllll}
\mathit{u11}(\mathit{Aout}(v_0)) & \rightarrow & \mathit{Aout}(v_0) & \mathit{u22}(\mathit{Aout}(v_0)) & \rightarrow & \mathit{Aout}(v_0) \\
\mathit{u21}(\mathit{Aout}(v_0), v_1) & \rightarrow & \mathit{u22}(\mathit{Ain}(v_1, v_0)) & \mathit{Ain}(O, v_0) & \rightarrow & \mathit{Aout}(S(v_0)) \\
\mathit{Ain}(S(v_0), O) & \rightarrow & \mathit{u11}(\mathit{Ain}(v_0, S(O))) & \mathit{Ain}(S(v_0), S(v_1)) & \rightarrow & \mathit{u21}(\mathit{Ain}(S(v_0), v_1), v_0)
\end{array} \right.
$$

The use of dependency pairs and dependency graphs, in conjunction with the following two polynomial

[12] A quadratic class was added as an extension of simple-mixed polynomials.

interpretations, allows us to prove termination of this system.

$$
\begin{array}{lll}
[\![\mathit{Aout}]\!]_1(X_0) = 0 & [\![\mathit{u11}]\!]_1(X_0) = 0 & [\![\mathit{u21}]\!]_1(X_0, X_1) = 0 \\
[\![\mathit{Ain}]\!]_1(X_0, X_1) = 0 & [\![\mathit{u22}]\!]_1(X_0) = 0 & [\![O]\!]_1 = 0 \\
[\![S]\!]_1(X_0) = X_0 + 1 & [\![\widehat{\mathit{u21}}]\!]_1(X_0, X_1) = X_1 + 1 & [\![\widehat{\mathit{Ain}}]\!]_1(X_0, X_1) = X_0
\end{array}
$$

$$
\begin{array}{lll}
[\![\mathit{Aout}]\!]_2(X_0) = 0 & [\![\mathit{u11}]\!]_2(X_0) = 0 & [\![\mathit{u21}]\!]_2(X_0, X_1) = 0 \\
[\![\mathit{Ain}]\!]_2(X_0, X_1) = 0 & [\![\mathit{u22}]\!]_2(X_0) = 0 & [\![O]\!]_2 = 0 \\
[\![S]\!]_2(X_0) = X_0 + 1 & [\![\widehat{\mathit{Ain}}]\!]_2(X_0, X_1) = X_1 &
\end{array}
$$

This termination proof is discovered with full automation by CiME.

Note that the complexity of Ackermann-Péter function makes impossible any termination proof with polynomial orderings using the Manna-Ness criterion.

II.3 Incremental and Modular Criteria

The size of systems that one meets in practice (several hundreds or thousands of rules) suggests that a Divide and Conquer approach would be of great help to prove termination. One can see the dependency graph analysis as a first step forward to this end, however it is worth noticing that the criteria briefly described in Section II.1.4, as of 2001, amounts to proving termination of $\twoheadrightarrow_{D,\mathbf{R}}$ for a system $\mathbf{R}$, that is *with the same set of rules*[13]. We are now interested in boiling down termination of R to termination of some $\twoheadrightarrow_{D_i,R_i}$ with $D_i \subseteq \mathsf{DP}(R)$ and some relevant $R_i \subseteq R$ only. I tackled this problem during my PhD, supervised by Marché.

What one would like is to prove termination of the whole system by using the knowledge that (some of) its subparts are terminating. Unfortunately termination is not a *modular* property, even when subparts share no symbol, as shown in a famous example by Toyama in 1987 [157].

Example 7 *Let us consider the following rewriting systems, not sharing any symbols:*

$$R_1 = \{f(a, b, x) \to f(x, x, x)\}$$

$$\Pi = \{\pi(x, y) \to x, \pi(x, y) \to y\}$$

System Π has no dependency pair, and R_1 terminates as its dependency graph contains no strongly connected part. However, the union $R_1 \cup \Pi$ yields an infinite rewriting sequence:

$$\cdots \to f(a, b, \pi(a, b)) \underset{R_1}{\longrightarrow} f(\pi(a, b), \pi(a, b), \pi(a, b)) \underset{\Pi}{\to} f(a, \pi(a, b), \pi(a, b)) \underset{\Pi}{\to} f(a, b, \pi(a, b)) \to \cdots$$

One way to avoid this problem is to enforce strong constraints on relations and unions (linearity properties, local confluence, non-projective rules, constructor-based extension, strategies, restricted proper extensions etc. [50, 101, 131, 58]). An important drawback is that it prevents from tackling most of the systems one meets in practice, and it excludes most of the unions that occur naturally in programming.

Instead of restraining the scope of our study by putting strong constraints upon unions, we choose to allow weakly constrained unions (so-called *hierarchical*) and consider a (slightly) restricted notion of termination that is preserved under non-deterministic collapse: CE-*termination*.

Remark 6 *The notion of* simple *termination, equivalent to termination preserved when one adds all projections for every symbol in the signature, could be seen as an interesting candidate, as it is modular for unions of composable finitely branching systems [79]. However, it is not modular for hierarchical unions (which in particular motivated the definition of proper extensions [101]), and above that, it is*

[13] Some very interesting results can however be obtained if one considers innermost rewriting [9].

far too restrictive in practice. Many systems are not simply terminating, and that would forbid most of the benefits one can expect from a dependency pair approach. In particular, CE*-termination strictly contains simple termination.*

From a historical point of view, Rusinowitch [146] remarks that the infinite reduction of Example 7 is due to the projective behaviour of system Π. In fact, similar counter-examples may arise when one deals with systems with a non-deterministic projective behaviour, more precisely when a term t may be reduced to two distinct variables. Gramlich [77] defines then the notion of *termination preserved under non-deterministic collapse* for which he obtains results. This is the property of terminating TRS that remain terminating when one adds rules $\{C[x, y] \to x; C[x, y] \to y\}$ for a context C (which express exactly that some term t containing variables x and y can be reduced to one of those). This is equivalent to adding system Π where π is fresh. That property is later renamed *Collapse Extended termination* (CE-termination, for short) and studied by Ohlebusch [136]: a system R is said to be CE-terminating if $R \cup \Pi$ (where π is a fresh symbol) is terminating.

CE-termination enjoys a good behaviour with reference to unions of systems. It is a modular property for unions of systems that do not share symbols [76, 136]. It is also modular for constructor-sharing [76] or *composable* unions [103] of systems that are *finitely branching* (that is where any term has only a finite number of reducts in one step). As it is not a severe restriction in practice (finite systems are indeed finitely branching), we consider hereafter finitely branching systems only.

Remark 7 *Further note that the basic estimation of dependency graphs EDG is coarse enough to allow the proof of* CE*-termination, as arcs potentially added by* Π *are not pruned.*

The common presentation of TRS is based on the union of sets of rules (over some signatures) which are TRS by themselves. This is not well suited to the expression of their inner hierarchical structure for at least the following two reasons: 1) It brings redundancy at the common subsystem definition level when symbols or rules are shared, and 2) It does not keep track of the actual sequential construction and thus does not enhance any incremental process (say for instance a termination proof). Both contribute to separating rewriting from actual programming.

In order to give a general framework bringing the hierarchical structure of TRS to the fore, and so as to provide automated methods to prove termination incrementally, I introduced in 2001 the notion of *rewriting modules*, and relevant termination techniques [160, 159, 158].

From an operational point of view, a module consists of a set of "new" symbols together with the rules that define them.

Definition 10 (Modules [160]) *Let R_1 be a term rewriting system over a signature $\mathcal{F}_1$. A* module *extending $R_1(\mathcal{F}_1)$ is a pair $(\mathcal{F}_2, R_2)$ such that:*

1. *$\mathcal{F}_1 \cap \mathcal{F}_2 = \emptyset$ (signatures are disjoint);*
2. *R_2 is a term rewriting system over $\mathcal{F}_1 \cup \mathcal{F}_2$;*
3. *for every $l \to r \in R_2$, $\Lambda(l) \in \mathcal{F}_2$.*

One can easily see that $R_1 \cup R_2$ is a TRS over $\mathcal{F}_1 \cup \mathcal{F}_2$. We say then that system $R_1 \cup R_2$ over $\mathcal{F}_1 \cup \mathcal{F}_2$ is a hierarchical extension of $R_1(\mathcal{F}_1)$ with module $(\mathcal{F}_2, R_2)$. *We write such an extension: $R_1(\mathcal{F}_1) \leftarrow (\mathcal{F}_2, R_2)$.*

Two disjoint modules may extend the same base hierarchy.

Definition 11 (Independent extension [160]) *We say that a module $(\mathcal{F}_2, R_2)$ extends a hierarchy headed by $(\mathcal{F}_0, R_0)$* independently of *a module $(\mathcal{F}_1, R_1)$ if: $\mathcal{F}_1 \cap \mathcal{F}_2 = \emptyset$, $(\mathcal{F}_0, R_0) \leftarrow (\mathcal{F}_1, R_1)$, and $(\mathcal{F}_0, R_0) \leftarrow (\mathcal{F}_2, R_2)$.*

Such an extension may be seen as a *union of composable systems* [131, 103, 137].

It is actually possible to express different kinds of classical extensions by means of extensions of modules. For instance, we obtain a *disjoint union* (union of systems sharing no symbol) $R_1(\mathcal{F}_1) \cup R_2(\mathcal{F}_2)$ whenever $(\mathcal{F}_1, R_1)$ and $(\mathcal{F}_2, R_2)$ extend $(\emptyset, \emptyset)$ independently of each other.

We may also describe a *constructor sharing union* with modules. A union of two TRS $R_1(\mathcal{F}_1)$ and $R_2(\mathcal{F}_2)$ is said to be a constructor sharing union if all symbols in $\mathcal{F}_1 \cap \mathcal{F}_2$ are constructors for both R_1 and R_2. Let us consider two systems $R_1(\mathcal{F}_1)$ and $R_2(\mathcal{F}_2)$ which only share a set of common constructors $C_0 = \mathcal{F}_1 \cap \mathcal{F}_2$. The constructor shared union $R_1 \cup R_2$ is easily denoted with two modules $(\mathcal{F}_1, R_1)$ and $(\mathcal{F}_2, R_2)$ independently extending the module of constructors $(C_0, \emptyset)$.

The notion of *hierarchical extensions with common subsystem* [136] is captured by left-associativity of module extension.

The only condition on extensions is that new rules must have a new symbol at root position of their left-hand side (see Definition 10). Thus, modules extensions subsume notions of heavily constrained hierarchical extensions such as *constructor-based extensions* [50] (systems in which no left-hand side has a symbol below the top that appears at the top of any left-hand side) and *proper extensions* [101, 102] (involving constraints on right-hand sides' subterms with reference to a dependency relation on symbols).

The intrinsic hierarchical structure of TRS naturally emerges from their formalisation in terms of modules. In particular, this approach allows one to build TRS the same way programs are built: modules after modules. It is also possible to split up a whole TRS in several modules.

Example 8 *The following system $R = R_0 \cup R_+ \cup R_*$ describes addition and multiplication of natural numbers in functional binary notation: $\#$ denotes $0_{\mathbb{N}}$, $(x)0$ denotes the value of x multiplied by two and $(x)1$ denotes the value of x multiplied by $2_{\mathbb{N}}$ plus $1_{\mathbb{N}}$. In this formalism, $6_{\mathbb{N}}$ is written $\#110$, that is the usual binary notation with a $\#$ in front.*

$$R_0 \quad = \quad \{\#0 \to \#$$

$$R_+ \quad = \quad \left\{ \begin{array}{lllllll} \# + x & \to & x & & x0 + y1 & \to & (x+y)1 \\ x + \# & \to & x & & x1 + y0 & \to & (x+y)1 \\ x0 + y0 & \to & (x+y)0 & & x1 + y1 & \to & ((x+y) + \#1)0 \end{array} \right.$$

$$R_\times \quad = \quad \left\{ \begin{array}{lllllll} \# \times x & \to & \# & & x0 \times y & \to & (x \times y)0 \\ x \times \# & \to & \# & & x1 \times y & \to & (x \times y)0 + y \end{array} \right.$$

The first (simplification) rule ensures that $2_{\mathbb{N}} \times 0_{\mathbb{N}}$ is $0_{\mathbb{N}}$, i.e. that naughts in front of numbers in binary notation can be erased.

One may partition the original signature into three disjoint sets that reflect respectively binary numbers, addition, and multiplication:

$$\mathcal{F}_0 = \{\#, 0, 1\} \;\; \mathcal{F}_+ = \{+\} \;\; \mathcal{F}_\times = \{\times\}.$$

System R may now be seen as an extension involving three modules:

$$(\mathcal{F}_0, R_0) \leftarrow (\mathcal{F}_+, R_+) \leftarrow (\mathcal{F}_\times, R_\times).$$

Now adding rules for subtraction only consists in extending $(\mathcal{F}_0, R_0)$ by the relevant module $(\mathcal{F}_-, R_-)$ independently of any other module.

$$\mathcal{F}_- \qquad \{-\}$$

$$R_- = \left\{ \begin{array}{lllllll} x - \# & \to & x & & \# - x & \to & \# \\ (x)0 - (y)0 & \to & (x-y)0 & & (x)0 - (y)1 & \to & ((x-y) - (\#)1)1 \\ (x)1 - (y)0 & \to & (x-y)1 & & (x)1 - (y)1 & \to & (x-y)0 \end{array} \right.$$

The module framework provides a dependency pair approach applicable to an incremental treatment of the termination proof.

Definition 12 (Dependency pairs of module [160]) *Let $M = (\mathcal{F}, R)$ be a module. A* dependency pair of module *M is a pair of terms $\langle \widehat{l}, \widehat{r'} \rangle$ such that there is a rule $l \to r \in R$ for which term r' is a subterm of r with $\Lambda(r') \in \mathcal{F}$ defined in R.*

We override $\mathsf{DP}(_)$ *and denote by* $\mathsf{DP}(M)$ *the set of all dependency pairs of a module M.*

Dependency pairs of modules obviously belong to the set of dependency pairs in the sense of Arts & Giesl. Moreover, for a system R over $T(\mathcal{F}, X)$, and splitting the original signature $\mathcal{F}$ into $\mathcal{F}_D \cup \mathcal{F}_C$ (respectively defined symbols and constructors), one obtains exactly the complete set of dependency pairs $\mathsf{DP}(R)$ simply by considering the extension $(\mathcal{F}_C, \emptyset) \leftarrow (\mathcal{F}_D, R)$ and $\mathsf{DP}((\mathcal{F}_D, R))$.

Chains of dependency pairs of modules rely on which rules *occurring in the hierarchy* may be applied between $\mathsf{DP}(M)$-steps, and not only on what is provided by the module M. In other words, dependency chains become relative to some relevant set of rules. For a module $M = (\mathcal{F}, R)$ and an arbitrary term rewriting system S, a *dependency chain of M over S* is a sequence of terms t_j such that

$$t_i \xrightarrow[S]{\neq\Lambda}{}^{\star} u\sigma \xrightarrow[\langle u,v\rangle \in \mathsf{DP}(M)]{\Lambda} v\sigma \equiv t_{i+1}.$$

That is exactly $\twoheadrightarrow_{\mathsf{DP}(M),S}$. It is said to be *minimal* if all proper subterms of every term in this sequence are mortal for S.

Note that since S is an arbitrary TRS, it may be completely different from the original system R. In particular, we may have $S \supset R$. As shown in Theorem 15, this allows to disregard rules locally irrelevant to termination.

Dependency pairs of modules and relative dependency chains allow us to define some purely syntactical tests so as to prove termination in an incremental fashion. The two main criteria obtained are the following:

Theorem 14 (Urbain [160]) *Let $(\mathcal{F}_2, R_2)$ be extending R_1 (over $\mathcal{F}_1$); if R_1 is* CE*-terminating, and $\twoheadrightarrow_{\mathsf{DP}((\mathcal{F}_2,R_2)),R_1 \cup R_2 \cup \Pi}$ is terminating, then $R_1 \cup R_2$ is* CE*-terminating.*

Theorem 15 (Urbain [160]) *Let $(\mathcal{F}_2, R_2)$ and $(\mathcal{F}_3, R_3)$ be extending R_1 (over $\mathcal{F}_1$)* independently*; if $R_1 \cup R_2$ is* CE*-terminating, and $\twoheadrightarrow_{\mathsf{DP}((\mathcal{F}_3,R_3)),R_1 \cup R_3 \cup \Pi}$ is terminating, then $R_1 \cup R_2 \cup R_3$ is* CE*-terminating.*

Note that R_2 is not involved in $\twoheadrightarrow_{\mathsf{DP}((\mathcal{F}_3,R_3)),R_1 \cup R_3 \cup \Pi}$.

Module-based termination criteria take indeed the full benefit of a really modular and incremental approach together with the advantages of a dependency pair approach: proving termination with modules amounts to solving fewer and simpler termination constraints than using a global approach.

Remark 8 *Further note that some dependency pairs of modules may be pruned using graphs criteria, provided that* CE *termination is preserved. As noticed in Remark 7, this is in particular always the case with the simple approximation EDG.*

Following the incremental development of a program, the proof search can be performed along the hierarchy provided by the programmer, as in Example 8: step by step, module after module. An alternative is to partition a whole system in modules that are minimal, in the sense that all functions they contain are mutually recursive. Such a minimal split is easily computed and proves useful on systems provided as a single set of numerous rules.

Effective Corollaries

Finding a relevant ordering pair $(\preceq, <)$ to complete the proof of termination requires that this ordering behaves well with reference to non-deterministic collapse. In fact it is sufficient to ask that $\pi(x,y) \succeq x$ and $\pi(x,y) \succeq y$ hold for a weakly monotonic extension of $(\preceq, <)$ to $T(\mathcal{F} \cup \{\pi\}, X)$. Those well-suited orderings are said to be π-expandable. This is not a very restrictive property: all simplification orderings are π-expandable, as well as their combination with AFS, or lexicographical composition [160].

Corollary 16 *Let $(\mathcal{F}_2, R_2)$ be extending R_1 (over $\mathcal{F}_1$); if*

1. *R_1 is* CE*-terminating, and*
2. *there is a weakly monotonic π-expandable ordering $(\preceq, \prec)$, such that: $R_1 \cup R_2 \subseteq \succeq$, and* $\mathsf{DP}((\mathcal{F}_2, R_2)) \subseteq \succ$,

then $R_1 \cup R_2$ is CE*-terminating.*

Corollary 17 *Let $(\mathcal{F}_2, R_2)$ and $(\mathcal{F}_3, R_3)$ be extending R_1 over $\mathcal{F}_1$* independently*; if*

1. *$R_1 \cup R_2$ is* CE*-terminating, and*
2. *there is a weakly monotonic π-expandable ordering $(\preceq, \prec)$, such that: $R_1 \cup R_3 \subseteq \succeq$, and* $\mathsf{DP}((\mathcal{F}_3, R_3)) \subseteq \succ$,

then $R_1 \cup R_2 \cup R_3$ is CE*-terminating.*

Note that for Corollary 17, as for Theorem 15, the ordering fulfils no constraint with reference to R_2.

Example 9 *So let us consider again the system $R_0 = \{\#0 \to \#\}$ describing natural numbers and which clearly* CE*-terminates. We already defined some arithmetic on these integers, in particular addition with $(\mathcal{F}_+, R_+)$ which we modify slightly by adding an associativity rule:*

$$R_+ = \left\{ \begin{array}{llllll} x + \# & \to & x & \# + x & \to & x \\ x0 + y0 & \to & (x+y)0 & x0 + y1 & \to & (x+y)1 \\ x1 + y0 & \to & (x+y)1 & x1 + y1 & \to & ((x+y) + \#1)0 \\ x + (y+z) & \to & (x+y)+z & & & \end{array} \right.$$

Termination of $R_0 \leftarrow (\mathcal{F}_+, R_+)$ is proven using dependency pairs and a polynomial interpretation.

$$\mathsf{DP}((\mathcal{F}_+, R_+)) = \left\{ \begin{array}{ll} \langle x0 \widehat{+} y0, x \widehat{+} y\rangle & \langle x1 \widehat{+} y1, (x+y) \widehat{+} \#1)\rangle \\ \langle x0 \widehat{+} y1, x \widehat{+} y\rangle & \langle x \widehat{+} (y+z), x \widehat{+} y\rangle \\ \langle x1 \widehat{+} y0, x \widehat{+} y\rangle & \langle x \widehat{+} (y+z), (x+y) \widehat{+} z\rangle \\ \langle x1 \widehat{+} y1, x \widehat{+} y\rangle & \end{array} \right\} \quad \begin{array}{l} [\![\#]\!] = 0 \\ [\![0]\!](x) = x+1 \\ [\![1]\!](x) = x+2 \\ [\![+]\!](x,y) = x+y+1 \\ [\![\widehat{+}]\!](x,y) = x+2y \end{array}$$

With reference to the ordering defined using the π-expandable interpretation above, pairs of $\mathsf{DP}((\mathcal{F}_+, R_+))$ *strictly decrease while rules of $R_0 \cup R_+ \cup\ \pi$ weakly decrease. Corollary 16 allows to conclude on* CE*-termination.*

When one adds subtraction, dependency pairs of modules together with polynomial-based orderings are sufficient again to show that $R_0 \cup R_-$ CE*-terminates. Indeed, for*

$$[\![\widehat{-}]\!](x,y) = x \quad [\![-]\!](x,y) = x \quad [\![\#]\!] = 0 \quad [\![0]\!](x) = x+1 \quad [\![1]\!](x) = x+1$$

pairs of $(\mathcal{F}_-, R_-)$ strictly decrease while rules in $R_0 \cup R_-$ weakly decrease. Corollary 17 yields that $R_0 \cup R_- \cup R_+$ CE*-terminates.*

In order to compare integers, one needs boolean operators. Let us add a new module, namely $(\mathcal{F}_{Bool}, R_{Bool})$.

$$\begin{array}{ll} \mathcal{F}_{Bool} & \{\mathit{true}; \mathit{false}; \neg; \wedge; \mathit{if}\} \\ R_{Bool} = & \left\{ \begin{array}{lllllllll} \neg(\mathit{true}) & \to & \mathit{false} & x \wedge \mathit{true} & \to & x & \mathit{if}(\mathit{true}, x, y) & \to & x \\ \neg(\mathit{false}) & \to & \mathit{true} & x \wedge \mathit{false} & \to & \mathit{false} & \mathit{if}(\mathit{false}, x, y) & \to & y \end{array} \right. \end{array}$$

This system is free of dependency pairs, hence it trivially CE*-terminates.*

Comparisons can be defined in a module $(\mathcal{F}_{\mathrm{ge}}, R_{\mathrm{ge}})$ *extending both* R_0 *and* R_{Bool}.

$$\begin{array}{ll} \mathcal{F}_{\mathrm{ge}} & \{\mathit{ge}\} \\ R_{\mathrm{ge}} = & \left\{ \begin{array}{llllll} \mathit{ge}(x0, y0) & \to & \mathit{ge}(x, y) & \mathit{ge}(x0, y1) & \to & \neg \mathit{ge}(y, x) \\ \mathit{ge}(x1, y0) & \to & \mathit{ge}(x, y) & \mathit{ge}(x1, y1) & \to & \mathit{ge}(x, y) \\ \mathit{ge}(x, \#) & \to & \mathit{true} & \mathit{ge}(\#, x0) & \to & \mathit{ge}(\#, x) \\ \mathit{ge}(\#, x1) & \to & \mathit{false} & & & \end{array} \right. \end{array}$$

CE*-termination of* $R_0 \cup R_{Bool} \cup R_{\mathrm{ge}}$ *is shown by RPO with* $\{\mathit{ge} > \neg > (\mathit{true}, \mathit{false})\}$ *directly. Remember that, being a simplification ordering, RPO is* π*-expandable. One may then apply Theorem 15 and obtain* CE*-termination of* $R_0 \cup R_{Bool} \cup R_{\mathrm{ge}} \cup R_+ \cup R_-$.

One may now define base 2 logarithm, rounded down. For technical reasons, it is easier to define firstly a Log' *such that* $\mathrm{Log}'(x) = \mathrm{Log}(x) + 1$ *with convention* $\mathrm{Log}'(0) = 0$.

$$\begin{array}{ll} \mathcal{F}_{\mathrm{Log}'} & \{\mathrm{Log}'\} \\ R_{\mathrm{Log}'} = & \left\{ \begin{array}{llllll} \mathrm{Log}'(\#) & \to & \# & \mathrm{Log}'(x1) & \to & \mathrm{Log}'(x) + \#1 \\ \mathrm{Log}'(x0) & \to & \mathit{if}(\mathit{ge}(x, \#1), \mathrm{Log}'(x) + \#1, \#) & & & \end{array} \right. \end{array}$$

Dependency pairs of modules and the following polynomial interpretations suffice to prove CE*-termination.*

$$\mathsf{DP}((\mathcal{F}_{\mathrm{Log}'}, R_{\mathrm{Log}'})) = \{\langle \widehat{\mathrm{Log}'}(x1), \widehat{\mathrm{Log}'}(x) \rangle \quad \langle \widehat{\mathrm{Log}'}(x0), \widehat{\mathrm{Log}'}(x) \rangle\}$$

$$\begin{array}{llll} [\![\#]\!] = 0 & [\![\mathit{false}]\!] = 0 & [\![+]\!](x, y) = x + y & [\![\mathit{if}]\!](x, y, z) = y + z \\ [\![0]\!](x) = x + 1 & [\![\mathit{true}]\!] = 0 & [\![\mathit{ge}]\!](x) = 0 & [\![\mathrm{Log}']\!](x) = x \\ [\![1]\!](x) = x + 1 & [\![\neg]\!](x) = 0 & [\![\wedge]\!](x, y) = x & [\![\widehat{\mathrm{Log}'}]\!](x) = x \end{array}$$

Dependency pairs strictly decrease while rules in $R_0 \cup R_+ \cup R_{Bool} \cup R_{\mathrm{ge}} \cup R_{\mathrm{Log}'}$ *weakly decrease.* CE*-termination of* $R_0 \cup R_+ \cup R_{Bool} \cup R_{\mathrm{ge}} \cup R_{\mathrm{Log}'} \cup R_-$ *follows from Corollary 17.*

The 'correct' logarithm is computed using module $(\mathcal{F}_{\mathrm{Log}}, R_{\mathrm{Log}})$ *defined as* $\mathcal{F}_{\mathrm{Log}} = \{\mathrm{Log}\}$ *and* $R_{\mathrm{Log}} = \{\mathrm{Log}(x) \to \mathrm{Log}'(x) - \#1\}$*. Since* $(\mathcal{F}_{\mathrm{Log}}, R_{\mathrm{Log}})$ *has no dependency pair, Theorem 15 gives* CE*-termination of* $R_0 \cup R_+ \cup R_- \cup R_{Bool} \cup R_{\mathrm{ge}} \cup R_{\mathrm{Log}'} \cup R_{\mathrm{Log}}$.

A minimal split of the whole system (31 rules) lead to a hierarchy of 13 modules, of which 4 contain no rule.

Proof Technique, Evolution towards Usable Rules

Suppose R_2 is an irrelevant set of rules according to the previous theorems. The proof of Theorems 14 and 15 rely again on a translation of relations. They are based on an interpretation I, inspired from another interpretation by Gramlich [76, 77], that translates steps in minimal chains $\twoheadrightarrow_{D,R_1 \cup R_2}$ over terms in $T(\mathcal{F}, X)$ into steps in chains $\twoheadrightarrow_{D,R_1 \cup \Pi}$ over terms in $T(\mathcal{F} \cup \{\pi : 2, \bot : 0\}, X)$ which are finite by assumption.

This approach has been fruitful. The proof technique is in turn adapted independently by Thiemann *et al.* [154] and Hirokawa & Middeldorp [83] to define a notion of *usable rules* in the context of rewriting without a given strategy[14].

[14] A notion of usable rules is given by Arts & Giesl for innermost rewriting in [9].

Usable rules yield less constraints than corollaries 16 and 17, and do not require CE-termination (but still use π-expandable orderings). They are stricly more powerful to prove termination of a given system, and subsume our approach. In particular they allow arbitrary graph approximations, notably those subtle enough to establish that there is no circuit in the graph for $f(0, 1, x) \to f(x, x, x)$.

However, one might note that they require to compute the whole dependency graph (and not only a graph per module), and thus they are less local and not incremental.

Finally note that theorems of this section and usable rules are completely formalised by Contejean in the COCCINELLE library (Chapter III).

From Theory to Practice

All those results, as well as some of the next section, were developed with a constant thought for automation. Thanks to their generality as well as the purely syntactical tests provided, they are part of the C*i*ME 2 rewriting tool [35], which I have already mentioned a few times, and which I shall briefly describe here.

Undertaken by Contejean, Marché, Monate and myself, the development of C*i*ME 2 began in 1997 as a complete reorganisation of the historical C*i*ME tool [34] with, in addition to completion techniques, a priority set on termination and *automation of termination proof discovery*. It has been replaced with C*i*ME 3 in 2009, even though not all techniques have migrated yet.

C*i*ME 2 provides a strongly typed programming language targeted to the study of term rewriting systems (computation, completion, termination, etc.). It features a Diophantine constraint solver (finite domains). The termination engine may discover suitable polynomial orderings (linear, simple, simple-mixed or quadratic) using the techniques of Section II.2.1, as well as path orderings, for standard rewriting and C or AC-rewriting (see Section II.4.1). Term rewriting systems can be entered as a single set, or as a hierarchy of modules. The user may then ask for a termination proof, possibly with a partition in minimal modules. The available criteria include Manna-Ness, dependency pairs: marked or unmarked, with or without dependency graphs (EDG).

While one may see REVE [109, 65] as the first tool offering the opportunity to search for termination proof with full automation, C*i*ME 2 may be considered as a pioneer of fully automated termination proving for rewriting systems. It was notably the first tool to look with full automation for orderings suitable for termination of systems with AC-symbols.

C*i*ME 2 is used mainly in academia (see for example [7, 100]) for proofs of termination, possibly modulo equational theories like associativity and commutativity (AC), proofs of confluence, completion (Knuth-Bendix), including completion modulo equational theories. It is also used in other provers like TALP [140] (U. Bielefeld) dedicated to termination proofs of well-moded logic programs, and, as we shall see, in MU-TERM (U. Valencia) for termination of context-sensitive rewriting, MTT (U. Málaga) for termination of MAUDE programs, etc.

II.4 Equational Theories, Strategies

II.4.1 AC Rewriting

In the practice of programming, it is common to use operations with an associative and commutative (AC for short) behaviour. This is for example typically the case for some classical operations in arithmetics (addition, multiplication, etc.), or logics (boolean disjunction, conjunction)...

However, a direct expression of AC by means of rules (for instance, $x + y \to y + x$ when $+$ is AC) is impracticable since these rules do not terminate. Moreover, using rewriting rules for associativity, $f(f(x, y), z) \to f(x, f(y, z))$ for instance, would also lead to non-termination when working modulo commutativity (C) as the following rewriting sequence does not terminate: $f(f(x, y), z) \to$

$f(x, f(y, z)) =_{\mathrm{C}} f(f(y, z), x) \to \cdots$ In order to preserve the termination property, rewriting over AC-equivalence classes, namely rewriting *modulo AC*, is used.

There are several slightly different variants of AC-rewriting, see [122] for a detailed comparison.

Class rewriting is defined by $s \xrightarrow[l \to r/\mathrm{AC}]{} t$ if there are s' and t' such that $s =_{\mathrm{AC}} s'$, $t =_{\mathrm{AC}} t'$ and $s' \xrightarrow[l \to r]{} t'$.

This definition is algorithmically complex because deciding whether a term s rewrites to another modulo AC amounts to searching for an instance of a rule in any term AC-equivalent to s.

AC-extended rewriting, proposed by Peterson and Stickel [141], is a restricted definition where AC-equivalence is allowed only at or below the position p of a term where one wants to match a rule: $s \xrightarrow[\mathrm{AC}\backslash l \to r]{p} t$ if there is a substitution σ such that $s|_p =_{\mathrm{AC}} l\sigma$ and $t = s[r\sigma]_p$.

There is a drawback in this restriction: rewriting is not necessarily *coherent* with AC in the sense defined by Jouannaud and Kirchner [92] where $s =_{\mathrm{AC}} t$ and $t \xrightarrow[\mathrm{AC}\backslash R]{} u$ should imply that there exist v and w such that $s \xrightarrow[\mathrm{AC}\backslash R]{}^{\star} v =_{\mathrm{AC}} w \xleftarrow[\mathrm{AC}\backslash R]{}^{\star} u$. Indeed, there are rewrite systems R for which $\to_{\mathrm{AC}\backslash R}$ terminates whereas $\to_{R/\mathrm{AC}}$ does not [122] (for example $a + a \to (a + b) + a$).

However, one may make any rewrite system R AC-coherent by adding *extended rules* to R. That is, for each rule $f(s, t) \to r$ where f is AC, adding the rule $f(f(s, t), x) \to f(r, x)$ where x is a fresh variable.

Varyadic AC-rewriting is a variant of AC-extended rewriting that, in some sense, builds the extended rules in the rewriting relation, by considering varyadic terms. It is defined by $s \xrightarrow[l \to r/\overline{\mathrm{AC}}]{p} t$ if there is a substitution σ such that either

- $s|_p =_{\mathrm{AC}} l\sigma$ and $t = s[r\sigma]_p$ or
- $l = f(l_1, \ldots, l_n)$ where f is AC and for a new variable x, $s|_p =_{\mathrm{AC}} f(l_1, \ldots, l_n, x)\sigma$ and $t = s[f(r, x)\sigma]_p/\overline{\mathrm{AC}}$.

Actually, when it comes to termination, the notion of rewriting used does not matter [70, 122]: for any TRS R, the following statements are equivalent:

1. class rewriting with R is terminating;

2. varyadic AC-rewriting with R is terminating;

3. AC-extended rewriting with R augmented with its extended rules is terminating.

We consider hereafter varyadic AC-rewriting, and we denote rewriting steps by $s \xrightarrow[l \to r/\mathrm{AC}]{p} t$.

Rewriting modulo AC complicates termination proofs significantly. In particular, the orders involved in proofs have to be *AC-compatible*, that is: AC-equivalent terms must be equivalent with reference to the ordering. For years, the problem of finding suitable orderings has been widely studied [12, 14, 44, 94, 95, 145] but all propositions lead to simplification orderings, too restrictive in practice.

There has been several approaches to extend the dependency pair approach to the AC-case [104, 106, 122], which enhances feasibility of automating termination proofs of AC-rewriting. Giesl and Kapur [70], propose in particular an extension of the dependency pair technique to rewriting modulo an arbitrary theory (nevertheless satisfying some properties), and the specialisation of their work to AC provides again another variant of the AC-dependency pair notion.

In 2005, Marché and I propose [123] a combination of AC-dependency pair techniques, CE-termination, and hierarchical combinations so as to obtain, firstly, on a theoretical plan, new modularity results on CE-termination modulo AC, and secondly, on a practical plan, powerful and incremental methods for proving AC-termination of TRS of large size, suitable for automation.

The first step to achieve this is to consider dependency pair techniques for AC-rewriting.

AC-Dependency Pairs

There has been several extensions of the dependency pair criteria to AC-rewriting, with similar but different definitions of AC-dependency pairs: Marché and I in 1998 [122], Kusakari & Toyama [106] and Giesl & Kapur [70] in 2001, Kusakari, Marché and I again [104] in 2002. For some of those generalisations, two variants of dependency pairs have been given: one for which root symbols are distinguished (like in the standard case), and one for which root symbols stay in the original signature.

Variants without marks have been considered for two reasons. Firstly, dependency pair criteria without marks are simpler, because marks require to introduce additional rules to handle some constraints existing between f and $\widehat{f}$ when f is AC; these rules are called here *marks-handling rules*. Secondly, the constraints given by those marks-handling rules usually impose that a symbol f and its marked copy $\widehat{f}$ are considered equivalent in the term ordering eventually found. Indeed, in practice, the set of constraints to solve for proving termination of a given TRS may be significantly larger when one uses marks than when one does not, with a larger search space (because the signature size is doubled), even without AC-symbols. Hence the time and space used to solve them can be much larger. As a matter of fact, it appears that, in practice, it is often sufficient to put marks only on non-AC symbols, see Remark 9.

The unmarked dependency pair definition Marché and I proposed in 1998 for varyadic AC-rewriting is the following.

Definition 13 (Unmarked AC dependency pairs [122]) *Let R be a rewriting system, let R_{ext} be the set of extended rules of R in the sense of Peterson & Stickel, then the set of AC unmarked dependency pairs of R (denoted by $\mathsf{DP}_{AC}(R)$) is the set of classical dependency pairs of $R \cup R_{ext}$.*

The definition of $\twoheadrightarrow_{D,S}$ is adapted to the AC-case as follows: $s \twoheadrightarrow_{D,S} t$ if there is a substitution σ such that $s \xrightarrow[S/\mathrm{AC}]{\neq\Lambda\ \star} s' \equiv u\sigma \xrightarrow[\langle u,v\rangle\in D]{\Lambda} t$. If all proper subterms of s and t are mortal, this sequence is said to be minimal. This definition leads to the following criterion for AC-rewriting.

Theorem 18 (Marché & Urbain [122]) *R is terminating for AC-rewriting if and only if $\twoheadrightarrow_{\mathsf{DP}_{AC}(R),R}$ is terminating.*

The extension to the dependency graph criterion is straightforward. However, computing EDG now requires AC-unification.

In order to design criteria for incremental proofs of termination that work with any of those AC-dependency pair notions, Marché and I defined an abstract notion of AC-dependency pairs, with some required properties which are satisfied by all of the former AC-dependency pair notions, and which I shall briefly describe.

Definition 14 (Abstract AC dependency pair [123]) *An* abstract AC dependency pair definition *is a function which, given a TRS R over a signature $\mathcal{F} = D \cup C$ where D are defined symbols of R and C are constructors, gives a set of pairs of terms denoted by $\langle u, v\rangle$ on signature $D \cup \widehat{D} \cup C$, and an additional rewrite relation $\rightarrow_{mh}$ called* marks-handling*, satisfying:*

i. For any dependency pair $\langle u, v\rangle$, $\Lambda(u)$ and $\Lambda(v)$ are in $\widehat{D}$. Moreover, there is a rewrite rule $l \rightarrow r$ in R such that $\Lambda(u) = \widehat{\Lambda(l)}$ (resp. $\Lambda(u) = \Lambda(l)$ for non-marked case) and $\Lambda(v) = \widehat{f}$ (resp. f) for some f occurring in r.

ii. $\to_{mh}$ *is generated by a finite set of rewrite rules* R_{mh} *such that for each* $l \to r$ *in* R_{mh}*, there exists one AC-symbol* f *such that* $\Lambda(l) = \Lambda(r) = \widehat{f}$ *and strict subterms of* l *and* r *may contain* f*,* $\widehat{f}$ *and variables only. Note that this system is not required to terminate, and it is necessarily empty if we do not use marks.*

iii. R *is terminating iff there is no infinite sequence of pairs* $\langle u_1, v_1\rangle, \langle u_2, v_2\rangle, \ldots$ *together with a substitution* σ *such that for all* i*:*

$$v_i\sigma \left(\xrightarrow[R/AC]{\neq\Lambda} \cup \to_{mh} \right)^{\star} u_{i+1}\sigma$$

Moreover, that sequence may be chosen minimal*, that is for which all proper subterms are mortal. This corresponds to the notion of dependency chain.*

It can be easily checked that properties (i) and (ii) are satisfied by existing concrete AC dependency pair techniques [70, 104, 106, 122] (property (iii) is the main result of each corresponding paper):

- Kusakari & Toyama [106] consider fixed-arity terms. With their variant without marks, $\to_{mh}$ is empty; and with marks, $\to_{mh}$ is generated by $\widehat{f}(\widehat{f}(x, y), z) \leftrightarrow \widehat{f}(f(x, y), z)$ and $\widehat{f}(\widehat{f}(x, y), z) \to \widehat{f}(x, y)$ for each AC-symbol f.
- Marché & Urbain [122] consider flat terms. In the concrete variant without marks given Definition 13 and with Theorem 18, $\to_{mh}$ is empty.
- Kusakari, Marché & Urbain [104] consider flat terms and marks, $\to_{mh}$ is the reduction relation generated by $\widehat{f}(x, y, z) \to \widehat{f}(f(x, y), z)$ for each AC-symbol f.
- Giesl & Kapur [70] (when specialised to the AC-case) consider fixed-arity terms and marks, $\to_{mh}$ is generated by $\widehat{f}(f(x, y), z) \leftrightarrow \widehat{f}(x, f(y, z))$ and $\widehat{f}(x, y) \leftrightarrow \widehat{f}(y, x)$ for each AC-symbol f.
- Marché & Urbain [36] implement a variant with flat terms, that uses marks for non-AC symbols but no marks for AC-symbols, in other words: $\widehat{D} = \{\widehat{f} \mid f \in D - \mathcal{F}_{AC}\} \cup \{f \mid f \in D \cap \mathcal{F}_{AC}\}$, with $\to_{mh}$ empty.

Remark 9 *As soon as it is chosen not to use marks for AC-symbols, all the variants above become equivalent. The last choice, marking non-AC symbols only, seems to be a good compromise, keeping the power of having marked symbols for non-AC symbols and avoiding the complexity induced by marked AC-symbols.*

Modularity Results for AC-Rewriting

While the proof of modularity of CE-termination for composable systems in [103] is quite involved, and extending it to rewriting modulo AC is probably a very hard task, the simple proof I gave in [160] is much easier to extend.

We keep the same definition of modules as in the non-AC case. A key point here is that AC behaves nicely with reference to hierarchies of modules: AC-steps introduce no additional dependencies between modules.

AC dependency pairs of modules are defined similarly to those of the non-AC case, that is: for a module $(\mathcal{F}, R)$, the *set of AC dependency pairs of module* $(\mathcal{F}, R)$ is the set of AC dependency pairs of R where all symbols not belonging to $\mathcal{F}$ are considered as constructors (in addition to the ones in $\mathcal{F}$).

AC dependency chains are defined accordingly: an *AC dependency chain of a module* $(\mathcal{F}, R)$ *over a TRS* S (with $R \subseteq S$) is a sequence of terms t_j such that

$$t_i \left(\xrightarrow[S/\mathrm{AC}]{\neq\Lambda} \cup \to_{mh} \right)^{\star} u\sigma \xrightarrow[\langle u,v\rangle \in \mathsf{DP}_{\mathrm{AC}}((\mathcal{F},R))]{\Lambda} v\sigma \equiv t_{i+1}$$

denoted by $t_i \twoheadrightarrow^{mh}_{\mathsf{DP}_{\mathrm{AC}}((\mathcal{F},R)),S} t_{i+1}$. This chain is said to be *minimal* if all proper subterms of every term in the chain are mortal for S/AC.

Quite remarkably, a slight modification of the interpretation involved in the proof technique of [160] allows to extend our modularity results to AC-rewriting easily. The main theorems and their effective corollaries become:

Theorem 19 (Marché & Urbain [123]) *Let $(\mathcal{F}_2, R_2)$ be extending R_1 over $\mathcal{F}_1$ where $\mathcal{F}_1 \cup \mathcal{F}_2$ contains AC-symbols; if R_1 is* CE*-terminating modulo AC, and $\twoheadrightarrow^{mh}_{\mathsf{DP}_{AC}((\mathcal{F}_2,R_2)),R_1\cup R_2\cup\Pi}$ is terminating, then $R_1 \cup R_2$ is* CE*-terminating modulo AC.*

Theorem 20 (Marché & Urbain [123]) *Let $(\mathcal{F}_2, R_2)$ and $(\mathcal{F}_3, R_3)$ be extending R_1 over $\mathcal{F}_1$* independently*; if $R_1 \cup R_2$ is* CE*-terminating modulo AC, and $\twoheadrightarrow^{mh}_{\mathsf{DP}_{AC}((\mathcal{F}_3,R_3)),R_1\cup R_3\cup\Pi}$ is terminating, then $R_1 \cup R_2 \cup R_3$ is* CE*-terminating modulo AC.*

The effective corollaries now require orderings that are also compatible with AC (and non-increasing on mark-handling steps), in particular Corollary 17 is extended to AC-rewriting as follows:

Corollary 21 *Let $(\mathcal{F}_2, R_2)$ and $(\mathcal{F}_3, R_3)$ be extending R_1 over $\mathcal{F}_1$* independently*; if*

1. *$R_1 \cup R_2$ is* CE*-terminating modulo AC, and*
2. *there is a weakly monotonic π-expandable ordering $(\preceq, \prec)$, AC (and marks) compatible, such that:*
   $R_1 \cup R_3 \subseteq\succeq$, and $\mathsf{DP}_{AC}((\mathcal{F}_3, R_3)) \subseteq\succ$,

then $R_1 \cup R_2 \cup R_3$ is CE*-terminating modulo AC.*

Finally, the extension to rewriting modulo AC of a previous result by Kurihara & Ohuchi [103] is obtained directly:

Theorem 22 (Marché & Urbain [123]) CE*-AC termination is a modular property for unions of composable finitely branching TRS.*

Example 10 *Instead of binary arithmetic, we may choose another efficient representation, proposed by Contejean, Marché and Rabehasaina [37]: ternary notation. This is implemented as a module $(\mathcal{F}_{int}, R_{int})$ where signature $\mathcal{F}_{int}$ contains the constant $\#$, and unary symbols 1, 0 and j which will be used as postfix operators. Intuitively, $\#$ represents 0, $(x)0$ represents $3x$, $(x)1$ represents $3x+1$ and $(x)j$ represents $3x-1$. As for Example 9 the rule part contains only one rule $R_{int} = \{\#0 \to \#\}$ (which clearly* CE*-terminates) to encode the relation between those constructors.*

We define addition with extending module $(\mathcal{F}_+, R_+)$ with $\mathcal{F}_+ = \{+\}$ where $+$ is associative and commutative,

$$R_+ = \begin{cases} x + \# \;\to\; x \\ x0 + y0 \;\to\; (x+y)0 & x0 + y1 \;\to\; (x+y)1 & x1 + y1 \;\to\; (x+y+\#1)j \\ x0 + yj \;\to\; (x+y)j & xj + yj \;\to\; (x+y+\#j)1 & x1 + yj \;\to\; (x+y)0 \end{cases}$$

Termination of $R_{int} \cup R_+$ is proven using our results, with dependency graphs. The compatible interpretation:

$$[\![\#]\!] = 0 \qquad [\![0]\!](x) = [\![1]\!](x) = [\![j]\!](x) = x+1 \qquad [\![+]\!](x,y) = x+y$$

removes a first pair from the whole graph, and the interpretation

$$[\![\#]\!] = [\![0]\!](x) = 0 \qquad [\![1]\!](x) = [\![j]\!](x) = 1 \qquad [\![+]\!](x,y) = x+y$$

finishes the proof for the remaining component, formed by pairs $\langle xj + yj + z, (x + y + \#j)1 + z\rangle$ *and* $\langle x1 + y1 + z, (x + y + \#1)j + z\rangle$.

Now we prove termination of the extension with the module below, defining multiplication, as well as opposite and subtraction as auxiliary operations:

$$R_\times = \begin{cases} x \times \# \rightarrow \# & x0 \times y \rightarrow (x \times y)0 & opp((x)0) \rightarrow (opp(x))0 \\ opp(\#) \rightarrow \# & x1 \times y \rightarrow (x \times y)0 + y & opp((x)1) \rightarrow (opp(x))j \\ x - y \rightarrow x + opp(y) & xj \times y \rightarrow (x \times y)0 - y & opp((x)j) \rightarrow (opp(x))1 \end{cases}$$

Knowing that $R_{int} \cup R_+$ CE*-terminates, the following polynomial interpretation is sufficient to fulfil the remaining constraints required for termination of* $R_{int} \cup R_+ \cup R_\times$*:*

$$\begin{array}{lll} [\![\#]\!] = 0 & [\![0]\!](x) = [\![1]\!](x) = [\![j]\!](x) = x + 1 & \\ [\![opp]\!](x) = [\![\widehat{opp}]\!] = x & [\![+]\!](x,y) = [\![-]\!](x,y) = x + y & [\![\times]\!](x,y) = xy + x + y \end{array}$$

Note that the termination proof given in [37] was a hand-made proof, involving a composition of complicated polynomial interpretations as well as ad-hoc *reasoning. The first automatic termination proof of* $R_{int} \cup R_+ \cup R_\times$ *was given by Borralleras & Rubio [21], and the above is another quite simple and automated proof.*

Now, we may define bags, then sum and product of elements in a bag of integers. Our fourth module is independent of the others, and defines bags, using signature $\mathcal{F}_{Bag}$ *which contains the constant* $\emptyset$ *to denote the empty bag, the unary symbol* $\{_\}$ *to build singletons, and the union* $\cup$ which is AC. *Its rule part consists of* $R_{Bag} = \{\emptyset \cup x \rightarrow x\}$*, which clearly* CE*-terminates.*

Our fifth module defines the sum of elements of a bag of integers:

$$R_\Sigma = \{\ \Sigma(\emptyset) \rightarrow \# \qquad \Sigma(\{x\}) \rightarrow x \qquad \Sigma(x \cup y) \rightarrow \Sigma(x) + \Sigma(y)$$

Applying Corollary 21, the polynomial interpretation

$$\begin{array}{ll} [\![\#]\!] = [\![\emptyset]\!] = [\![0]\!](x) = [\![1]\!](x) = [\![j]\!](x) = 0 & [\![+]\!](x,y) = x + y \\ [\![\{_\}]\!](x) = [\![\Sigma]\!](x) = [\![\widehat{\Sigma}]\!](x) = x & [\![\cup]\!](x,y) = x + y + 1 \end{array}$$

suffices to prove the termination of the set of all rules introduced so far. In particular, no additional constraint on multiplication is required, since $(\mathcal{F}_\Sigma, R_\Sigma)$ *extends* $(\mathcal{F}_+, R_+)$ independently *of* $(\mathcal{F}_\times, R_\times)$.

We may now introduce a last module defining product of elements of a bag of integers:

$$R_\bigotimes = \{\ \bigotimes(\emptyset) \rightarrow \#1 \qquad \bigotimes(\{x\}) \rightarrow x \qquad \bigotimes(x \cup y) \rightarrow \bigotimes(x) \times \bigotimes(y)$$

Applying again Corollary 21, the polynomial interpretation:

$$\begin{array}{ll} [\![\#]\!] = [\![\emptyset]\!] = 0 & [\![0]\!](x) = [\![1]\!](x) = [\![j]\!](x) = [\![opp]\!](x) = 0 \\ [\![\cup]\!](x,y) = x + y + 1 & [\![+]\!](x,y) = [\![\times]\!](x,y) = x + y \\ [\![-]\!](x,y) = x & [\![\{_\}]\!](x) = [\![\bigotimes]\!](x) = [\![\widehat{\bigotimes}]\!](x) = x \end{array}$$

is enough to prove the termination of the set of all rules introduced, again no constraint on Σ *is required.*

Finally, we establish the termination of the whole system (containing 23 rules, 3 AC-symbols and 6 modules) by using polynomial interpretations only. As far as we know, any former method would need a more complicated ordering: other approaches may need for instance lexicographical composition which is difficult to search for automatically, and because of rule $x1 + y1 \rightarrow (x + y + \#1)j$ *one cannot use ACRPO [145]. All those interpretations were found in a few seconds by CiME in a fully automated way.*

Another interesting example is the following system by Deplagne, for which the techniques presented above and their implementation in C*i*ME allowed to obtain the first known proof of termination. It comes from his study of sequent calculus modulo [45] to which we refer for further details.

The considered TRS (53 rules and 2 AC-symbols) is in two parts, given in Figure II.3. The first one, R_1, provides explicit substitutions for quantifiers, while the second one, R_2, describes a congruence over a sequent calculus. Operators $\{_{\mathtt{s}}_\}$ and $\{_{\mathtt{f}}_\}$ denote respectively singletons of sequents and singletons of formulas. The union operators of these sets are AC and are denoted respectively by '•' and by ','.

The termination proof for TRS $R_1 \cup R_2$ was discovered by C*i*ME in a few seconds, with full automation. The system was in fact decomposed by C*i*ME into a hierarchy of 19 modules, 9 of which yielding no constraint. Note that the proof search strategy used no mark on AC-symbols and estimated dependency graphs in conjunction with polynomial interpretations.

These examples emphasise how hierarchical proof techniques and their extensions to the AC-case, open the way to termination proofs of systems that come from real applications and practical examples. Let us go further on this way.

II.4.2 Context-Sensitive Rewriting

Another paradigm well-suited to model programming languages is *context-sensitive* (CS) rewriting where the use of a *local* strategy blocks or enables rewriting steps at certain positions (see [114, 115] for a detailed study).

In CS rewriting, a *replacement map* (a mapping $\mu : \mathcal{F} \to \mathscr{P}(\mathbb{N})$ satisfying $\mu(f) \subseteq \{1, \ldots, k\}$, for each k-ary symbol f of a signature $\mathcal{F}$) is used to discriminate the argument positions on which the rewriting steps are allowed (μ-replacing positions); rewriting at Λ is always possible.

Example 11 *The following system from Borralleras [20] zips two lists of integers into a single one that contains quotients of the two components:*

$$\begin{array}{rcl@{\qquad}rcl}
\mathtt{sel}(\mathtt{0}, \mathtt{cons}(x, xs)) & \to & x & \mathtt{sel}(\mathtt{s}(n), \mathtt{cons}(x, xs)) & \to & \mathtt{sel}(n, xs) \\
\mathtt{minus}(x, \mathtt{0}) & \to & x & \mathtt{minus}(\mathtt{s}(x), \mathtt{s}(y)) & \to & \mathtt{minus}(x, y) \\
\mathtt{quot}(\mathtt{0}, \mathtt{s}(y)) & \to & \mathtt{0} & \mathtt{quot}(\mathtt{s}(x), \mathtt{s}(y)) & \to & \mathtt{s}(\mathtt{quot}(\mathtt{minus}(x, y), \mathtt{s}(y))) \\
\mathtt{zWquot}(\mathtt{nil}, x) & \to & \mathtt{nil} & \mathtt{from}(x) & \to & \mathtt{cons}(x, \mathtt{from}(\mathtt{s}(x))) \\
\mathtt{zWquot}(x, \mathtt{nil}) & \to & \mathtt{nil} & \mathtt{tail}(\mathtt{cons}(x, xs)) & \to & xs \\
\mathtt{head}(\mathtt{cons}(x, xs)) & \to & x & & & \\
\end{array}$$

$$\mathtt{zWquot}(\mathtt{cons}(x, xs), \mathtt{cons}(y, ys)) \to \mathtt{cons}(\mathtt{quot}(x, y), \mathtt{zWquot}(xs, ys))$$

with $\mu(\mathtt{cons}) = \{1\}$ *and* $\mu(f) = \{1, \ldots, ar(f)\}$ *for all other symbols* $f \in \mathcal{F}$.

As only the first argument of cons *can be rewritten, this prevents the infinite sequence* $\mathtt{from}(x) \to \mathtt{cons}(x, \mathtt{from}(\mathtt{s}(x))) \to \mathtt{cons}(x, \mathtt{cons}(\mathtt{s}(x), \mathtt{from}(\mathtt{s}(\mathtt{s}(x))))) \to \cdots$

Dependency pair criteria for context-sensitive rewriting were proposed in 2006 by Alarcón, Gutiérrez and Lucas [3, 4]. I worked in collaboration with Gutiérrez and Lucas on the extension of the notion of usable rules to CS rewriting [82]. As a result, we proposed a well-suited notion of usable rules, and we established that their classical notion [73, 83] was sound for the restricted class of *strongly conservative* context-sensitive systems.

Usable Rules for CS Rewriting

A straightforward extension of usable rules (with direct dependency) is indeed unsound for general systems. The problem comes in particular from *migrating* variables, i.e., variables that occur at non-μ-replacing positions in left-hand sides of rules but at μ-replacing positions in right-hand sides, and thus that possibly activate symbols *hidden* at non-μ-replacing positions in right-hand sides.

$$
R_1 \left\{
\begin{array}{rclrcl}
\mathsf{ef}(x)[y]_{\mathtt{t}} & \to & \mathsf{ef}(x[y]_{\mathtt{t}}) & \mathsf{Pe}(x)[y]_{\mathtt{f}} & \to & \mathsf{Pe}(x[y]_{\mathtt{t}}) \\
(f \vee g)[s]_{\mathtt{f}} & \to & (f[s]_{\mathtt{f}}) \vee (g[s]_{\mathtt{f}}) & \neg(f)[s]_{\mathtt{f}} & \to & \neg(f[s]_{\mathtt{f}}) \\
(f \wedge g)[s]_{\mathtt{f}} & \to & (f[s]_{\mathtt{f}}) \wedge (g[s]_{\mathtt{f}}) & x[i]_{\mathtt{t}} d & \to & x \\
(f \Rightarrow g)[s]_{\mathtt{f}} & \to & (f[s]_{\mathtt{f}}) \Rightarrow (g[s]_{\mathtt{f}}) & 1[x \cdot s]_{\mathtt{t}} & \to & x \\
\exists(f)[s]_{\mathtt{f}} & \to & \exists(f[1 \cdot (s \circ \uparrow)]_{\mathtt{f}}) & f[i]_{\mathtt{f}} d & \to & f \\
\forall(f)[s]_{\mathtt{f}} & \to & \forall(f[1 \cdot (s \circ \uparrow)]_{\mathtt{f}}) & (f[s]_{\mathtt{f}})[t]_{\mathtt{f}} & \to & f[s \circ t]_{\mathtt{f}} \\
(x[s]_{\mathtt{t}})[t]_{\mathtt{t}} & \to & x[s \circ t]_{\mathtt{t}} & id \circ s & \to & s \\
\uparrow \circ (x \cdot s) & \to & s & (s \circ t) \circ u & \to & s \circ (t \circ u) \\
(x \cdot s) \circ t & \to & (x[t]_{\mathtt{t}}) \cdot (s \circ t) & s \circ id & \to & s \\
1 \cdot \uparrow & \to & id & (1[s]_{\mathtt{t}}) \cdot (\uparrow \circ s) & \to & s
\end{array}
\right.
$$

$$
R_2 \left\{
\begin{array}{rcl}
a, \bigtriangledown & \to & a \\
a, a & \to & a \\
a \bullet \Diamond & \to & a \\
a \bullet a & \to & a \\
\neg(\neg(f)) & \to & f \\
f \wedge f & \to & f \\
f \vee f & \to & f \\
f \Rightarrow g & \to & \neg(f) \vee g \\
\exists(f) & \to & \neg(\forall(\neg(f))) \\
(a, \{_{\mathtt{f}} \neg(f)\}) \vdash b & \to & a \vdash (\{_{\mathtt{f}} f\}, b) \\
\{_{\mathtt{f}} \neg(f)\} \vdash b & \to & \bigtriangledown \vdash (b, \{_{\mathtt{f}} f\}) \\
a \vdash (\{_{\mathtt{f}} \neg(f)\}, b) & \to & (a, \{_{\mathtt{f}} f\}) \vdash b \\
a \vdash \{_{\mathtt{f}} \neg(f)\} & \to & (a, \{_{\mathtt{f}} f\}) \vdash \bigtriangledown \\
(a, \{_{\mathtt{f}} f \wedge g\}) \vdash b & \to & (a, \{_{\mathtt{f}} f\}, \{_{\mathtt{f}} g\}) \vdash b \\
\{_{\mathtt{f}} f \wedge g\} \vdash b & \to & (\{_{\mathtt{f}} f\}, \{_{\mathtt{f}} g\}) \vdash b \\
a \vdash (\{_{\mathtt{f}} f \vee g\}, b) & \to & a \vdash (\{_{\mathtt{f}} f\}, \{_{\mathtt{f}} g\}, b) \\
a \vdash \{_{\mathtt{f}} f \vee g\} & \to & a \vdash (\{_{\mathtt{f}} f\}, \{_{\mathtt{f}} g\}) \\
\{_{\mathtt{s}} a \vdash (\{_{\mathtt{f}} f \wedge g\}, b)\} & \to & \{_{\mathtt{s}} a \vdash (\{_{\mathtt{f}} f\}, b)\} \bullet \{_{\mathtt{s}} a \vdash (\{_{\mathtt{f}} g\}, b)\} \\
\{_{\mathtt{s}} a \vdash \{_{\mathtt{f}} f \wedge g\}\} & \to & \{_{\mathtt{s}} a \vdash \{_{\mathtt{f}} f\}\} \bullet \{_{\mathtt{s}} a \vdash \{_{\mathtt{f}} g\}\} \\
\{_{\mathtt{s}} (a, \{_{\mathtt{f}} f \vee g\}) \vdash b\} & \to & \{_{\mathtt{s}} (a, \{_{\mathtt{f}} f\}) \vdash b\} \bullet \{_{\mathtt{s}} (a, \{_{\mathtt{f}} g\}) \vdash b\} \\
\{_{\mathtt{s}} \{_{\mathtt{f}} f \vee g\} \vdash b\} & \to & \{_{\mathtt{s}} \{_{\mathtt{f}} f\} \vdash b\} \bullet \{_{\mathtt{s}} \{_{\mathtt{f}} g\} \vdash b\} \\
\{_{\mathtt{s}} (a, \{_{\mathtt{f}} f\}) \vdash (\{_{\mathtt{f}} f\}, b)\} & \to & \Diamond \\
\{_{\mathtt{s}} (a, \{_{\mathtt{f}} f\}) \vdash \{_{\mathtt{f}} f\}\} & \to & \Diamond \\
\{_{\mathtt{s}} \{_{\mathtt{f}} f\} \vdash (b, \{_{\mathtt{f}} f\})\} & \to & \Diamond \\
\{_{\mathtt{s}} \{_{\mathtt{f}} f\} \vdash \{_{\mathtt{f}} f\}\} & \to & \Diamond \\
\{_{\mathtt{s}} a \vdash b\} \bullet \{_{\mathtt{s}} (a, f) \vdash (g, b)\} & \to & \{_{\mathtt{s}} a \vdash b\} \\
\{_{\mathtt{s}} a \vdash b\} \bullet \{_{\mathtt{s}} (a, f) \vdash b\} & \to & \{_{\mathtt{s}} a \vdash b\} \\
\{_{\mathtt{s}} a \vdash b\} \bullet \{_{\mathtt{s}} a \vdash (b, f)\} & \to & \{_{\mathtt{s}} a \vdash b\} \\
\{_{\mathtt{s}} a \vdash \bigtriangledown\} \bullet \{_{\mathtt{s}} (a, f) \vdash b\} & \to & \{_{\mathtt{s}} a \vdash \bigtriangledown\} \\
\{_{\mathtt{s}} a \vdash (b, f)\} \bullet \{_{\mathtt{s}} \bigtriangledown \vdash b\} & \to & \{_{\mathtt{s}} \bigtriangledown \vdash b\} \\
\{_{\mathtt{s}} a \vdash b\} \bullet \{_{\mathtt{s}} \bigtriangledown \vdash b\} & \to & \{_{\mathtt{s}} \bigtriangledown \vdash b\} \\
\{_{\mathtt{s}} a \vdash b\} \bullet \{_{\mathtt{s}} a \vdash \bigtriangledown\} & \to & \{_{\mathtt{s}} a \vdash \bigtriangledown\} \\
\{_{\mathtt{s}} a \vdash b\} \bullet \{_{\mathtt{s}} \bigtriangledown \vdash \bigtriangledown\} & \to & \{_{\mathtt{s}} \bigtriangledown \vdash \bigtriangledown\}
\end{array}
\right.
$$

Figure II.3. Sequent calculus modulo, parts I & II.

Consider $R = \{f(c(x), x) \to f(x, x), b \to c(b)\}$ from [5] together with $\mu(f) = \{1, 2\}$ and $\mu(c) = \emptyset$. Regarding $C = \{\langle \widehat{f}(c(x), x), \widehat{f}(x, x)\rangle\}$, the set of classical usable rules is empty because $\widehat{f}(x, x)$ contains no symbol in $\mathcal{F}$. We could wrongly conclude to μ-termination, but we have the infinite minimal $(R, C, \widehat{\mu})$-chain:

$$\widehat{f}(c(b), b) \xrightarrow[C]{} \widehat{f}(b, b) \xrightarrow[R\mu]{\neq\Lambda} \widehat{f}(c(b), b) \to \cdots$$

We need an appropriate notion of dependency between symbols. The idea is to take into account symbols occurring at non-μ-replacing positions in the *left-hand sides* of the rules: for $R = (\mathcal{F}, \mu, R)$ a CS system, we say that $f \in \mathcal{F}$ *directly μ-depends* on $g \in \mathcal{F}$, written $f \trianglerighteq_{d,\mu} g$, if there is a rule $l \to r \in R$ with $f = \Lambda(l)$ and either (1) g occurs in r at a μ-replacing position, or (2) g occurs in l at a non-μ-replacing position.

Note that condition (2) is not too restrictive in programming practice because most programs are *constructor systems*, which means that no defined symbol occurs below the root in the left-hand side of the rules.

The notion of μ-dependency allowed us to define usable rules well-suited to context-sensitive rewriting, and to propose the effective related theorem (Definition 8 and Theorem 1 in [82]). These results were successfully implemented in the automated tool MU-TERM [2, 116], which gave the first automated proof of Example 11.

A Class of Systems for Classical Usable Rules

It is worth noticing that our usable rules contain the classical ones, and are generally much more numerous. To preserve efficiency, we identified a class of systems where classical usable rules can be safely used.

A *strongly conservative* rule is a rule with no migrating variable and where μ-replacing positions and non-μ-replacing positions do not share variables, in left-hand side, as well as in right-hand side. Strongly conservative systems consist solely of strongly conservative rules.

Gutiérrez, Lucas and I established that the straightforward use of classical usable rules [73, 83] is sound for the class of strongly-conservative systems (Theorem 2 in [82]). In fact, we show that if the dependency pairs in a part of the graph and the corresponding classical usable rules are strongly conservative, then the latter are the only rules to consider. This allows to use both our notion and the weaker but computationally cheaper classical one in different parts of a dependency graph. This is typically the case for Example 11, where two of the three cycles can be safely handled by the straightforward approach while the last one requires our improved notion.

Since then Alarcón *et al.* [1] have introduced another notion of CS dependency pairs which is closer to the usual case than the one we used. Their notion of usable rules is comparable, but does not improve over ours [81].

Finally, note that in the context of project A3PAT, basic context-sensitive dependency pairs were formalised in the COCCINELLE library (see Chapter III).

II.4.3 Membership Equational Programs

In addition to evaluation strategies like CS, sophisticated equational languages such as ELAN [19], MAUDE [27], OBJ [75], CAFEOBJ [67], HASKELL [89]..., enjoy advanced features such as polymorphism, higher-order, rewriting modulo, conditions and use of extra variables in conditions, partiality, and expressive type systems. However, many of those features are, at best, only partially supported by current termination tools (for instance MU-TERM, C*i*ME, APROVE, TTT [83], TERMPTATION [22], etc.) while they may be essential to ensure termination.

In collaboration with Durán, Lucas, Marché and Meseguer, I studied termination techniques for membership equational programs [54, 55].

We defined a sequence of theory transformations that can be used to bridge the gap between expressive equational programs, such as MAUDE programs, and termination tools, and proved these transformations to be sound.

Transformations for Automation of Operational Termination

In Membership Equational Logic (MEL) the two basic types of atomic predicates are equalities $t = t'$, and memberships $t : s$ stating that a term t has sort s. There is a basic level of typing by *kinds*; and a more sophisticated one by *sorts*, this is achieved by deduction using theory axioms.

Operationally, and assuming 'good' executability properties, equalities $t = t'$ can be treated as rewrite rules $t \to t'$. Rewriting with equations as rules can furthermore be made *context-sensitive* by providing a replacement map. In this way we arrive at the notion of a *context-sensitive membership rewrite theory* (CS-MRT), which is the operational form of a membership equational program. Roughly speaking, there are two kinds of rules: rewriting rules $l \to r \mid c$, and membership rules $t : k \mid c$, where c is a list of conditions of the form $s_i \to t_i$ or $s_i : k_i$. Note in particular that in a CS-MRT rewriting and computation of memberships $t : s$ are *recursively intertwined* because application of a conditional equation may require satisfying memberships in its conditions, and application of a conditional membership may likewise require satisfying equalities in its conditions.

This leads to specific termination behaviours. Consider, for example, the following membership equational program, expressed in MAUDE as follows:

```
fmod INF is
  protecting NAT .
  sort Inf .
  subsort Inf < Nat .
  var N : Nat .
  cmb s(N) : Inf if s(s(N)) : Inf .
endfm
```

Because of conditional membership `cmb s(N) : Inf if s(s(N)) : Inf .`, this program does not terminate. However, no rewriting is involved in its nontermination. This means that the standard termination notions are insufficient for dealing with termination of MEL programs. For this reason, we use a proof-theoretic termination notion, called *operational termination* [119], developed by Lucas, Marché and Meseguer, that faithfully captures the termination of conditional MEL programs, that is, of CS-MRT. Intuitively, a MEL program is operationally terminating if all its well-formed proof trees, for an axiomatised description of the relation, are *finite*. For example, the nontermination of the `INF` program above is witnessed by the infinite proof tree,

$$\cfrac{\cfrac{\cfrac{\ldots}{\texttt{s(s(s(N))):Inf}}}{\texttt{s(s(N)):Inf}}}{\texttt{s(N):Inf}}$$

The notion of operational termination is useful to clarify termination issues with conditions, as a conditional specification may have a terminating rewrite relation and still be nonterminating by 'looping' in evaluating the condition.

We tackle the problem of (operational) termination for membership equational programs by using a sequence of transformations that map the original program into increasingly simpler ones, each having the property that termination of the original program at each step ensures termination of the input program, until we reach a transformed program that can be entered into a termination prover.

We apply two transformations steps, a general description of which is given below, and that we prove to be both sound. See [55] for full technical details.

Transformation A eliminates sorts and memberships, resulting in an unsorted context-sensitive and conditional rewrite theory. We add a truth value $\top$, an unary operator $\mathrm{is}_k(_)$ for each sort k, and rules that *compute* sort/membership checks. One can then add adequate conditions to mimic those checks. Modulo technical details to disambiguate overloading and handle typing of variables, the idea is to get in particular rules like:

- $\mathrm{is}_k(f(x_1,\cdots,x_n)) \to \top \mid \{\mathrm{is}_{k_i}(x_i) \to \top\}_{1\leq i\leq n}$ for each $f:\ k_1\cdots k_n \to k\ \in \mathcal{F}$
- $s \to t \mid \{\mathrm{is}_{k_i}(x_i) \to \top\}_{1\leq i\leq n}, c'_1,\ldots c'_n$ for each conditional rule $s \to t \mid c_1,\ldots,c_n$ where the $x_i : k_i$ are the variables involved, and where $c'_j = \mathrm{is}_{k_j}(u_j) \to \top$ if c_j is $u_j : k_j$, and the rewrite condition otherwise.
- $\mathrm{is}_k(s) \to \top \mid \{\mathrm{is}_{k_i}(x_i) \to \top\}_{1\leq i\leq n}, c'_1,\ldots c'_n$ for each membership rule $s : k \mid c_1,\ldots,c_n$ where the $x_i : k_i$ are the variables involved, and the c'_j are as above.

We prove in particular in [55] that μ operational termination of the obtained conditional system implies (μ) operational termination of the original membership one.

We conjecture that this transformation is complete.

Transformation B eliminates conditions, possibly with extra variables. This transformation is based on a technique by Ohlebusch [138, 139] for *deterministic* conditional systems of *type 3*, that is systems consisting of rules $l \to r \mid c$ such that $\mathrm{Var}(r) \subseteq \mathrm{Var}(l) \cup \mathrm{Var}(c)$ (type 3) and such that for $c = s_1 \to t_1,\cdots,s_n \to t_n$, $\mathrm{Var}(s_i) \subseteq \mathrm{Var}(l) \cup \bigcup_{j=1}^{i-1} \mathrm{Var}(t_j)$ (deterministic)[15]. As defined by Ohlebusch, each conditional rule $l \to r|c$ with n conditions in c is transformed into $n+1$ unconditional rules using U inductively defined as:

$$\begin{array}{rcl} \mathrm{U}(l \to r) & = & \{l \to r\} \\ \mathrm{U}(l \to r \mid s \to t,\ c) & = & \{l \to u(s,\vec{x})\} \cup \mathrm{U}(u(t,\vec{x}) \to r \mid c) \end{array}$$

where u is a fresh new symbol added to the signature, and $\vec{x} = \mathrm{Var}(l) \cap (\mathrm{Var}(t) \cup \mathrm{Var}(r) \cup \mathrm{Var}(c))$.

We extend this transformation to our needs. In addition to U, we allow context-sensitive rewriting by extending the replacement map μ into μ' defined as $\mu'(U_i) = \{1\}$ for all U_i that are introduced to deal with the conditional part of each rule in R (that is, only the first argument of U can be evaluated), and $\mu'(f) = \mu(f)$ for all $f \in \mathcal{F}$.

We prove in particular in [55] that μ'-termination of the obtained system implies (μ) operational termination of the original conditional one.

Further note that this transformation is not complete.

These transformations are implemented in the tool MTT dedicated to the proof of termination for MAUDE programs[16]. MTT performs transformations A and B (several variants and technical optimisations are available) and then uses an external solver, like MU-TERM, C*i*ME or APROVE, for the termination proof of the resulting unconditional system.

Example 12 *Consider the* MAUDE *specification from [55], which computes the length of a* finite *list.*

```
fmod LengthOfFiniteLists is
   sorts Nat NatList NatIList .
   subsort NatList < NatIList .
   op 0 : -> Nat .
   op s : Nat -> Nat .
   op zeros : -> NatIList .
```

[15]This tranformation is similar to Marchiori's *unravelings* [125, 126].

[16]MTT is available at `http://www.lcc.uma.es/~duran/MTT`

```
   op nil : -> NatList .
   op cons : Nat NatIList -> NatIList [strat (1 0)] .
   op cons : Nat NatList -> NatList [strat (1 0)] .
   op length : NatList -> Nat .
   vars M N : Nat .
   var  IL : NatIList .
   var  L : NatList .
   eq zeros = cons(0,zeros) .
   eq length(nil) = 0 .
   eq length(cons(N, L)) = s(length(L)) .
endfm
```

`NatList` *and* `NatIList` *are intended to classify finite and infinite lists of natural numbers, respectively. Operator* `cons` *is overloaded, and is declared with evaluation strategy* `(1 0)`*. It can be used both for building finite and infinite lists of natural numbers. The interpretation of this strategy annotation is as follows: the evaluation of an expression* `cons(h,t)` *proceeds by first evaluating* `h` *and then trying a reduction step at the top position (represented by* `0`*). Evaluation on the second argument* `t` *is forbidden because index* `2` *is missing in the annotation. Note that* `NatList` *is a subsort of* `NatIList`*. This system is terminating but both the strategy for* `cons` *and the use of sorts (especially* `NatList` *and* `NatIList` *instead of a single one) are crucial to achieve this terminating behaviour. By removing either the strategy annotation or the sort information we would get a non-terminating program: On the one hand, if reductions were allowed on the second argument of* `cons`*, then the evaluation of* `zeros` *would never terminate. On the other hand, an attempt to evaluate* `length(xs)` *will not terminate if* `length` *'accepts' infinite lists* `xs` *like, e.g.,* `zeros`*; this is forbidden by specifying that* `length` *only accepts lists of sort* `NatList`*, i.e., finite lists.*

The corresponding context-sensitive membership rewriting theory is as follows:

```
fmod LengthOfFiniteListsMRT is
   kind [Nat].
   kind [NatIList] .
   op 0 : -> [Nat] .
   op s : [Nat] -> [Nat] .
   op zeros : -> [NatIList] .
   op nil : -> [NatList] .
   op cons : [Nat] [NatIList] -> [NatIList] [strat (1)] .
   op length : [NatIList] -> [Nat] .
   cmb L : NatIList if L : NatList .
   mb 0 : Nat .
   cmb s(N) : Nat if N : Nat .
   mb zeros : NatIList .
   mb nil : NatList .
   cmb cons(N,IL) : NatIList if N : Nat /\ IL : NatIList .
   cmb cons(N,L) : NatList if N : Nat /\ L : NatList .
   cmb length(L) : Nat if L : NatList .
   eq zeros = cons(0,zeros) .
   eq length(nil) = 0 .
   ceq length(cons(N,L)) = s(length(L)) if N : Nat /\ L : NatList .
endfm
```

An interesting optimisation consists in collapsing a conditional part without extra variables like $is_{s_1}(x_1) \;\to\; \top \wedge \cdots \wedge is_{s_k}(x_k) \;\to\; \top$*, of a conditional rule, into a single expression* $\mathtt{and}(is_{s_1}(x_1),\mathtt{and}(\cdots, is_{s_k}(x_k))\cdots) \to \top$ *by introducing a binary operator* `and` *defined by*

```
 op and : S S -> S .
 eq and(tt,T) = T .
```

Moreover, if the right-hand side of the conditional rule is $\top$*, one may replace it with the previous expression, resulting in an unconditional rule:* $l \to \mathtt{and}(is_{s_1}(x_1),\mathtt{and}(\cdots, is_{s_k}(x_k))\cdots)$.

After Transformation A with the aforementioned collapsing, and transformation B, program `LengthOfFiniteLists` *can be shown to be terminating by proving the* μ*-termination of the system:*

```
and(tt,T) -> T
isNatIList(IL) -> isNatList(IL)
isNat(0) -> tt
isNat(s(N)) -> isNat(N)
isNat(length(L)) -> isNatList(L)
isNatIList(zeros) -> tt
isNatIList(cons(N,IL)) -> and(isNat(N),isNatIList(IL))
isNatList(nil) -> tt
isNatList(cons(N,L)) -> and(isNat(N),isNatList(L))
isNatList(take(N,IL)) -> and(isNat(N),isNatIList(IL))
zeros -> cons(0,zeros)
length(nil) -> 0
length(cons(N,L)) -> uLength(and(isNat(N),isNatList(L)),L)
uLength(tt,L) -> s(length(L))
```

where $\mu(\mathtt{isNat}) = \mu(\mathtt{isNatList}) = \mu(\mathtt{isNatIList}) = \emptyset$*,* $\mu(\mathtt{and}) = \mu(\mathtt{cons}) = \mu(\mathtt{uTake1}) = \mu(\mathtt{uTake2}) = \mu(\mathtt{uLength}) = \{1\}$ *and* $\mu(f) = \{1, \ldots, ar(f)\}$ *for all other symbols* f*. The* μ*-termination of this system can be automatically proved with* APROVE *or* MU-TERM.

Up to this point, we proposed techniques and approaches aimed to prove termination of programs. Starting from vanilla first-order rewriting, we extended our concerns to rewriting modulo AC, and to systems of rules containing membership and conditions, that is, much closer to widespread programming languages, as illustrated by Example 12. Moreover, we developed techniques that are not limited to toy problems, but level to sizeable examples as one sees in practice, for instance termination for μ-CRL specifications of hundreds of rules (provided by Arts), or discovery of termination proofs for intricate calculi (Figure II.3), and for which human verification is out of reach.

All our contributions are implemented in automated provers, large pieces of software with many thousands lines of code. Can we trust their results? I shall describe in the next chapter the approach we developed in the A3PAT project dedicated to bridge the gap between skeptical proof assistants and automated provers.

Chapter III

An Architecture for Certification of Automated Proofs

III.1 Trustworthy Decision Procedures

Verification of programs and specifications may involve a significant amount of formal methods to guarantee required properties for complex or critical systems. In the context of proving, users rely on proof assistants with which they interact, step by step, until the proof is totally and mechanically verified. However, formal proof may be costly, and as proof assistants often lack automation, there is an increasing need to use fully automated tools providing powerful (and intricate) decision procedures. Although some proof assistants can check the soundness of a proof, the results of automated provers are often taken *as is*, even if the provers may be subject to bugs. Since application fields include possibly critical sectors such as security, code verification, cryptographic protocols, etc., reliance on verification tools is a crucial issue.

Automated provers and proof assistants do not mix easily. One of the strengths of proof assistants like COQ [152] is a *highly reliable* procedure that checks the soundness of proofs. For instance, COQ or ISABELLE/HOL [135] have a small and trustworthy *kernel*. In COQ, the kernel type-checks a *proof term* to ensure the soundness of a proof. Those assistants need to check mechanically the proof of *each property used*. Certified-programming environments based on these proof assistants find this an additional guarantee. However, this means that the proof assistant has to check a property proven by an external procedure before accepting it. Therefore, such a procedure must return a *proof trace* checkable by the assistant.

In 2005 the French National Agency for Scientific Research (ANR) gave me the opportunity to coordinate the A3PAT project[1] aimed to bridge the gap between proof assistants yielding formal guarantees of reliability, and highly automated tools one has to trust, and thus to fulfil the important need for externalisation of proofs using rewriting techniques. In its initial state, with a large spectrum, A3PAT gathered 10 research scientists, (associate and full) professors from 4 laboratories (partners): CÉDRIC (CNAM), LRI-PCRI (CNRS), LABRI (Bordeaux I), project MARELLE (INRIA Sophia-Antipolis), as well as 3 post-doctoral positions, two of whom (Forest and Paskevich) were under my supervision. I shall describe the project and how we want to provide both enhanced automation for proof assistants (by proof delegation), and formal certificates for automatically generated proofs (by mechanical checking of proof traces). As I shall explain, our proof structure, based on inference rules, is highly modular, and constitutes a good base for a multi-certifier platform.

There are several pieces of work to be mentioned with reference to communication between automated provers and COQ. Amongst them, the ZÉNON theorem-prover [17], based on tableaux, produces COQ proof terms as certificates. ELAN enjoys techniques to produce COQ certificates for rewriting [134]. Bezem describes an approach regarding resolution [15]. However, these systems do not tackle the problem of termination proofs.

[1] http://a3pat.ensiie.fr

Regarding certification of termination proofs, several libraries have been developed, amongst which COCCINELLE [29], but also CoLoR [23] and IsaFoR [155]. CoLoR is a library for COQ formalising theorems in a pure deep fashion. On the one hand, proof scripts for CoLoR are very short. On the other hand, some pure deep theorems may be difficult (costly) to compile, as their premises may include completeness properties, for instance graph-related theorems. CoLoR can manage orderings based on polynomials and matrices, DP criteria and relative termination. It also provides some results for termination modulo AC.

IsaFoR is a very recent library for ISABELLE/HOL. Its standalone certifier module is obtained by *extraction*, that is an automatically generated Haskell program (using the code-generation feature of ISABELLE/HOL). The extracted certifier is thus very efficient for checking termination proofs. However, its results have to be accepted as axioms (unchecked) by proof assistants. IsaFoR can manage orderings based on polynomials, and most DP refinements like usable rules [154], subtle graph approximations, etc. It features also some techniques to certify non-termination proof.

It should be noticed that the A3PAT approach neither subsumes nor is subsumed by the techniques of CoLoR or the newcomer IsaFoR. In fact, CoLoR even uses some of our developments.

III.2 Satellite-Based Architecture

The main purpose of proof assistants is to *check* proofs mechanically and not to *discover* them. Using satellite tools to which the proof assistant may delegate proofs is an interesting way to add automation to proof assistants. It does not add to the complexity of critical kernels, and it helps in delegating costly computations to highly specialised and efficient tools. The proof assistant is then left with what it is best at: checking that the given proof is correct.

The architecture of A3PAT is summarised in Figure III.1. It involves two main components: the C*i*ME 3 rewriting tool (embedding a certification engine that generates proof scripts) and the COCCINELLE library which formalises most of the properties needed.

The sequence of events is as follows. There are various scenarios for start: a development in a proof assistant, a verification of specifications for instance, needs a proof of termination for a rewriting system. The proof assistant may delegate this particular proof to an external (satellite) prover, like C*i*ME 3 or APROVE. Alternate case: a user of some termination automated prover may want a certificate for a termination proof, etc.

Once a proof has been discovered, it is encoded in a well-suited XML format and submitted to the certification engine in C*i*ME 3, which generates a script for COQ. COQ scripts may of course be a bit obscure. However, regarding termination proofs, the only parts that have to be human readable in scripts are: the definition of the system, and a theorem stating that the aforementioned system defines a well-founded relation.

Note that, as the trace (XML) format is compact, some more computations may be performed by C*i*ME 3 to generate specialised instantiations of lemmas and functions.

In order to certify the proof, COQ must compile the axiom-free script. This task usually requires formal definitions and theorems gathered in the COCCINELLE library.

III.2.1 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) [39]. λ-abstraction is denoted by `fun x:T => t`, and application is denoted by `t u`.

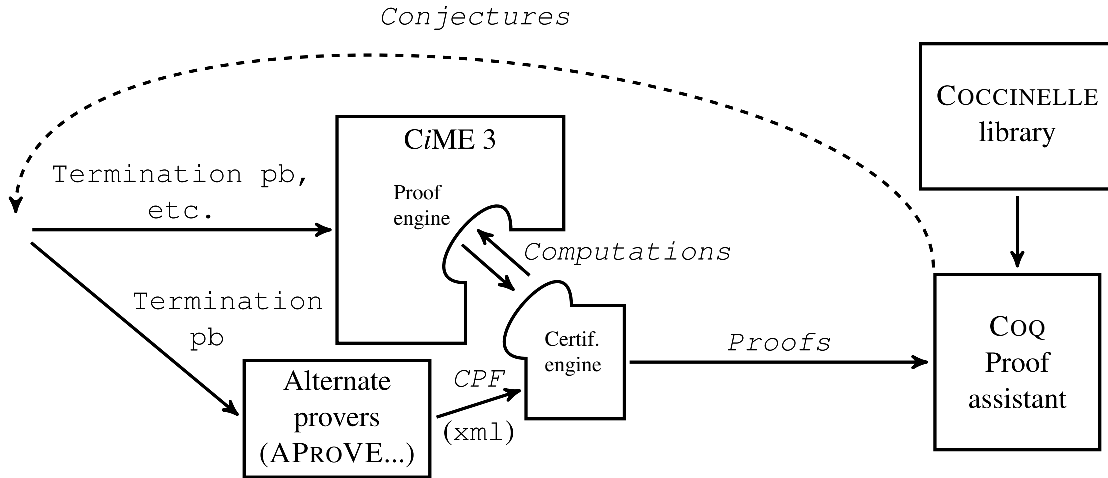


Figure III.1. Certification architecture.

2. A *proof checker* which checks the validity of proofs written as CIC-terms. In this framework, a term is a *proof* of its type, and checking a proof consists in typing a term. The correctness of the tool relies on this type checker, which is a small kernel of 5 000 lines of Objective Caml code.

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*[2], 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.

Termination in COQ

The termination property is defined in the COQ standard library as the well-foundedness of an *ordering*. Hence we model rewriting systems as *orderings* in the following. This notion is defined using the *accessibility* predicate: a term $t : A$ is said to be accessible for an ordering $<$ if all its predecessors are, inductively, 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 base case (when an element has no predecessor with reference to 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.

[2]Note that this notion of constructors is different from the one introduced in Section II.1.4.

If R is the rewrite relation modelling a system 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 I shall use as much as possible the COQ notation: t -[R]> u (and t -[R]*> u for $t \to^* u$) instead.

The desired final theorem stating that R is terminating has the following form:

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

Proofs of this theorem are supposed to be discovered by an automated prover, and generated in COQ syntax by the certifier tool, all with full automation. The only things the user has to check is firstly that the generated relation R corresponds to R, and secondly that the generated COQ files do compile.

Shallow or 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, term rewriting systems can be presented as 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 embeddings have advantages and drawbacks. On the plus side of shallow embedding are: an easy implementation of rewriting notions, and the absence of need for meta notions (as substitutions, etc.). For example, a definition of critical pairs and Newman's lemma is almost straightforward this way, allowing (simple) proofs of confluence with little effort. 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 (take for instance our RPO) but at the cost of a rather technical first step consisting in defining the generic notions and proving generic lemmas.

We chose an hybrid approach where some notions are deep (Σ-algebra, RPO) and others are (also) shallow (rewriting system, dependency graphs, polynomial interpretations). In fact, using both embeddings in a single proof is not a problem, moreover it is possible to take full benefit of both.

III.2.2 COCCINELLE

COCCINELLE is the formal library upon which we rely for certification using COQ. Developed by Contejean in the context of A3PAT, its main purpose is to model (in COQ) universal algebras, that is terms built on arbitrary signatures, as well as the usual notions defined in this framework. Amongst them, rewriting and equational theories generated by a set of rules/equations (seen as pairs of terms) are the key relations over terms, used to model computations and equality.

The links between COCCINELLE and C*i*ME 3 are of two distinct natures. Firstly, COCCINELLE models (some of) the structures and algorithms implemented in C*i*ME 3, but also states theorems on them. There is for example, a modelling of lists and filtering functions with reference to a property, but this model comes with some additional statements, in particular: the function actually computes what is intended: an element is filtered if and only if it belongs to the list given as an argument, and the wanted property holds for it. COCCINELLE is hence used to certify the modelling of some crucial algorithms used in C*i*ME 3, such as matching modulo associativity-commutativity (AC) [30], unification, RPO orderings, etc.

The second link may exist between COCCINELLE and any tool that produces traces in the relevant format (Section III.2.4). COCCINELLE provides some generic theorems that can be combined with the trace to prove some specific properties, such as the termination of a rewriting system in a given term

algebra. COCCINELLE together with the part of C*i*ME 3 that turns an XML trace into a COQ file can be seen as a trace compiler, which transforms a trace into a COQ certificate.

Amongst other examples, COCCINELLE has been successfully used by Castéran and Contejean to prove the accessibility of ordinals less than Γ_0 in the COQ library Cantor, thanks to its RPO [25], and by Stratulat for the integration of implicit induction in certified environments [150].

Terms and Relations over Terms

The modelling of terms is kept as simple as possible:

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

A term is either a variable (`Var`) or a function symbol applied to a list of arguments (`Term`).

The main relation defined over terms is a single step of rewriting. Using a constructor `Var`, instead of COQ's universal quantifier, allows to represent a (finite) rewriting system as a finite first-order data, actually a list of pairs of terms. For example, the following rewriting system:

$$\begin{cases} \text{plus}(x, \text{zero}) & \rightarrow & x \\ \text{plus}(x, \text{succ } y) & \rightarrow & \text{succ}(\text{plus}(x, y)) \end{cases}$$

can be represented by the (deep) object:

```
[ (Term plus [Var x; Term zero []], Var x);
  (Term plus [Var x; Term succ [Var y]],
     Term succ [Term plus [Var x; Var y]]) ]
```

or by the (shallow) COQ inductive relation:

```
Inductive Peano (relation term) :=
| Plus_zero : ∀t, Peano t (Term plus [t; Term zero []])
| Plus_succ :
   ∀ t t', Peano (Term succ [Term plus [t; t']])
                  (Term plus [t; Term succ [t']])
```

The first representation allows to reason by induction on the number of rules in the system, and define some functions that compute on it, for example a function that returns a list of dependency pairs. The second representation allows to use facilities and features COQ offers on inductive definitions, for example inversion.

Both representations are useful, and once the equivalence between them is proven (on a given rewrite system), both can be used, according to the wanted property.

The definition of rewriting is done in two steps: since the variables are implicitly universally quantified, a rule between terms t_1 and t_2 yields all the pairs $(t_1\sigma, t_2\sigma)$ obtained by instantiating the variables by any substitution σ.

```
Inductive axiom (R : relation term): relation term :=
 | instance : ∀t1 t2 sigma, R t1 t2 →
   axiom R (apply_subst sigma t1) (apply_subst sigma t2).
```

It is possible then to perform the replacement of $t_1\sigma$ by $t_2\sigma$ under a context, which is either empty (`at_top`), or deep (`in_context`).

```
Inductive one_step (R : relation term): relation term :=
 | at_top : ∀t1 t2, axiom R t1 t2 → one_step R t1 t2
 | in_context :
     ∀ f l1 l2, one_step_list (one_step R) l1 l2 →
           one_step R (Term f l1) (Term f l2).
```

The usual rewriting relation is the transitive closure of the `one_step` relation, and the equational theory is its reflexive, symmetric and transitive closure. Thanks to this formalisation, Contejean proved all the usual trivial facts on rewriting and equational theories gathered in the COCCINELLE specification file `term_algebra/equational_theory_spec.v`.

Available Developments

Available developments include:

- the RPO relation with status, proven to be an ordering, compatible with instantiation and addition of context, and to be well-founded when the underlying precedence over symbols is well-founded. The RPO relation can be decided thanks to a COQ function that computes a boolean result.
- dependency pair criteria, both marked and unmarked versions, with Dershowitz's improvement;
- dependency graphs, extended subterm (Section II.1.4);
- modularity (Section II.3), usable rules [83, 154];
- rewriting under some strategies (innermost [72], context-sensitive [3]) is modelled, and the corresponding versions of the DP criterion are proven;
- Gramlich's criterion [78] showing plain termination when innermost termination is proven.

III.2.3 C*i*ME 3

Our tool C*i*ME 3 can be used as a satellite prover for the COQ proof assistant. Evolving from the C*i*ME family[3], it consists of

- A rewriting toolbox, with a termination engine, and
- A proof compiler, generating COQ proof scripts.

The rewriting toolbox provides numerous techniques and procedures on term algebras: matching, unification, rewriting, and completion, either for free algebras or modulo equational theories [34] with commutative or associative and commutative operators, with unit elements, etc. It enjoys also a termination engine for first-order rewriting systems. Note that not all techniques at work in C*i*ME 2 have been implemented in C*i*ME 3 yet.

Certified termination criteria include:

- Manna-Ness, lexicographic composition of orderings,
- dependency pair criteria, marked or unmarked, with Dershowitz's improvement;
- dependency graphs, extended subterm criterion.

Certified termination orderings include:

- polynomial interpretations (Section II.2.1),
- various matrix interpretations [57, 42],
- full RPO with status [47] (including comparison with status 'lex' of symbols with different arities) and AFS refinements [9].

[3] `http://cime.lri.fr`

Some other techniques are not trace-producing yet, even though they are completely formalised in the COCCINELLE library (narrowing, innermost refinements [9], modules, usable rules etc.). Note that for efficiency reasons, the termination engine may use an external SAT-solver to find orderings, *à la* [6, 66].

When C*i*ME 3 finds a proof, it may either call directly the proof compiler to produce a COQ script for certification, or produce an XML trace to be used later by our proof compiler (or another one).

C*i*ME 3 accepts several input languages, depending on how one wants to use it: in batch mode or in interactive session.

Interactive sessions require the tool's dedicated programming language. The example below shows a basic interactive session with C*i*ME 3. The answers of the top-level are given in *slanted* font. We firstly define a signature, a term algebra for Peano arithmetic:

```
CiME> let F_peano =
          signature "O : constant; s : unary; plus : binary;";
F_peano : signature = signature "s : 1; plus : 2; O : 0"

CiME> let X = variables "x,y,z";
X : variable_set = variables "z,x,y"

CiME> let A_peano = algebra  F_peano ;
A_peano : F_peano algebra = algebra F_peano
```

We define then terms and term rewriting systems on this algebra:

```
CiME> let R_peano = trs A_peano "
    plus(x,O)    -> x ;
    plus(x,s(y)) -> s(plus(x,y)) ;";
R_peano : F_peano trs = trs A_peano ...
```

Now we may check the termination of the system:

```
CiME> termination R_peano;
[...information depending on verbosity...]
- : bool = true
```

and produce an XML trace:

```
CiME> xml_proof_trace  R_peano;
<?xml version="1.0" encoding="UTF-8"?>
<PROOF>
  <SIGNATURE>
  ...
</PROOF>
- : unit = ()
```

or directly a COQ certificate:

```
CiME> coq_certify_proof R_peano;
...
 Lemma wf :
  well_founded (algebra.EQT.one_step
    R_peano_deep_rew.R_peano_rules).
....
- : unit = ()
```

Note that the last lemma of the script is the well-foundedness of the given relation.

Termination proof search may be driven by heuristics. Heuristics are described with a small tactic language defined by the grammar below:

```
heuristic ::= Solve [ heuristic_list ]
            | Then [ heuristic_list ]
            | Use criterion
            | Repeat heuristic

heuristic_list ::= heuristic
                 | heuristic ; heuristic_list

criterion ::= DP | DPM | SCC | RMVx | MN  ...
```

The **`Use`** command applies the criterion given in argument, **`Solve`** tries to apply a list of heuristics on a goal until one succeeds, **`Then`** takes a list of heuristics to be applied in sequence and that must succeed on each step, finally **`Repeat`** repeatedly applies a heuristic until it fails.

The commands of the following example say to use the dependency pair criterion with marks (`DPM`) and then to repeat the sequence `SCC ; RMVx` where `SCC` tries to apply an instance of the GRAPH rule with a decomposition into strongly connected components which is indeed compatible, and `RMVx` (called on each of the resulting subsets) tries to find a pair $\langle t, u\rangle$ on which rule RMVERTEX applies, as described in Section II.1.4.

```
let h = heuristic
   "Then [Use DPM; Repeat Then [ Use SCC ; Use RMVx ]]";
set_heuristic "h";
```

It is also possible to drive the ordering constraints solving by selecting which orderings to try. This can be specified using the command `orderparams` with a list of ordering specification (bound or type of polynomials, dimensions of matrices, class of AFS, etc.).

All this can also be done in batch mode. We have to give the input format (amongst `cime`, `trs`, `srs`, `xml`, `cpf`) and the output format (amongst `cime`, `trs`, `srs`, `xml`, `cpf`, `coq`). The following example produces an CPF certificate `certif.xml` from a given TRS, specified in the `R.trs` file, using the information about orderings, heuristics, etc., provided in file `info.cim`:

```
cime -icime info.cim -itrs R.trs -ocpf certif.xml
```

Finally, the next example is an invocation of C*i*ME3 for the compilation of a CPF file `termination_trace.xml` (which may come from the C*i*ME 3 termination engine or from another tool) into a COQ source file `coq_certificate.v`

```
cime -icpf termination_trace.xml -ocoq coq_certificate.v
```

III.2.4 Trace Language

Defining an efficient trace language is a challenging task. If we restrict to C*i*ME alone, such a language may be used for objects as different as proof of termination, of confluence, of equality, etc. At the same time, it must keep the verbosity low: those proofs contain so many implicit steps that just emitting their sequence is illusory.

An important point is that the language must be designed to be as modular as possible, with no global declaration, in order to ease the composition of traces produced by different automated provers.

We started to define a first version of a trace language at the beginning of the A3PAT project. The overall structure of our original XML trace is similar to the inference tree mentioned in Section II.1.3. It is designed to be as concise as possible, while carrying all information that can be used by C*i*ME to *compute* a proof script. For example, all technical functions depending on the dependency graph will be computed and instantiated by C*i*ME. There is no trace left of a structure of graph in COQ, just sufficient lemmas, thanks to the techniques we proposed in [40] (see Section III.3).

The XML file lists the system, the applied criteria and gives the relevant orderings to solve the termination problem. An interesting tag is `<PROPERTY>`, which allows the combination of criteria. It has two mandatory attributes, `criterion` which contains the name of the (inference rule) criterion applied and `prop` which expresses the type of problem being solved. For instance `<PROPERTY criterion="dp" prop="sntrs">` means that the DP criterion has to be used to prove that the given system is terminating. The remaining trace contains the system, and the `<PROPERTY>` embedding the proof trace of the generated subproblems.

This format is not C*i*ME dependent, it is general enough to allow other provers' proofs to be certified by our approach. In particular, APROVE can output traces for COQ certification with A3PAT.

CPF

However, in order to facilitate interaction between provers and certifiers, the A3PAT and CeTA/IsaFoR [155] teams[4] recently decided to start the design of an extensible common proof format (CPF). While slightly more verbose, its overall structure is close to C*i*ME's XML format. C*i*ME supports CPF in addition to its own XML trace format, and other provers like APROVE and TTT2 [99] also support it, at least partially. A detailed description of CPF is available on `http://cl-informatik.uibk.ac.at/software/cpf/`.

CPF is now the official trace language for the termination competition [124] (certifying category).

III.3 Proof Techniques, the Case of Graphs

Certifying techniques as involved as one may find in the termination proof area amounts to defining efficient proof approaches, especially in the context of constructive proofs.

This is particularly the case when computationally complex structures like graphs are involved. Some properties may require heavy computations and particularly involved algorithmics, and may be very difficult to overcome for a proof assistant (even with the help of dedicated libraries). For instance Theorem 2, the graph criterion of [69], states that one has to find a suitable ordering "for *each* circuit [...] in the graph". That means that the property has to be shown for each cycle separately, and moreover that one has to prove that *all* cycles have been considered. Applying directly such a theorem is currently out of reach regarding the size of graphs that occur in the practice of termination proof.

Courtieu, Forest and I proposed in 2008 a first approach for certifying proofs using this theorem [40] which proves to be very efficient in practice.

The key point of our approach is that the graph will be defined *implicitly*. As it is the case for proof discovery, we do not apply Theorem 2 as is, but rather rely on a compatible decomposition (Section II.1.4). We never actually model a graph, we just use a set of vertices and a relation between them to build it implicitly as we prove the relevant property on its parts, in a hierarchical fashion. Regarding termination proofs: vertices will be dependency pairs, a pair p_1 will be in relation with a pair p_2 if $p_1 p_2$ may occur in a dependency chain.

Graph refinement theorems are proved for each instance, by shallow embedding. We generate a direct proof for each application of rules GRAPH and RMVERTEX. This proof is done by induction on the possible dependency chains in the initial graph. In particular this induction follows a graph that is now *implicit*. It proceeds, in fact, along the directed acyclic graph (DAG) of strongly connected components, which is the compatible decomposition we use.

All in all, the graph is only useful to find the suitable decomposition.

[4] With additional comments from the CoLoR team.

III.3.1 Formalisation of Hierarchical Decomposition

We work directly on dependency chains which we model by inductive relations. Using inductive types, reasoning on all possible dependency chains can be done by induction on the definition of the relation.

For example, suppose we have a strongly connected component `SCC` defined by the set of pairs:

$$\texttt{SCC} = \{\langle \texttt{plus(s x,y)},\texttt{plus(x,y)}\rangle, \langle \texttt{plus(x,s y)},\texttt{plus(x,y)}\rangle\}.$$

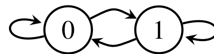
The corresponding dependency chain relation is generated as follows (notations are simplified):

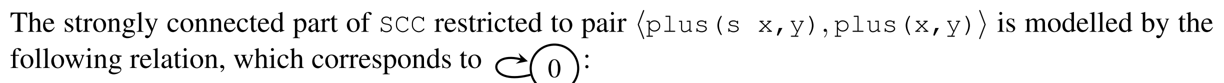
```
Inductive SCC : term → term → term:=
  SCC0: ∀V0 V1, x -[R]*> s(V0) → y -[R]*> V1
                   → plus(x,y) -[SCC]> plus(V0,V1)
| SCC1: ∀V0 V1, x -[R]*> V0 → y -[R]*> s(V1)
                   → plus(x,y) -[SCC]> plus(V0,V1)
```

Note that `SCC y x` is *exactly* equivalent to $\texttt{x} \twoheadrightarrow_{\texttt{SCC},R} \texttt{y}$. In particular: head reduction by `R` is disallowed *by construction*. Further note that this relation is not defined constructively: the set of terms `x` such that $\texttt{x -[R]*> S(V}_0\texttt{)}$ is not defined explicitly, and actually it cannot be computed in general. In our methodology the approximation lies, *during reasoning*, in the way we discard terms t that cannot reduce to u. The currently generated proofs implement EDG [9], this can be easily extended to EDG*, and actually be parameterised by an approximation.

Sub-DAG of Dependency Chains

Consider the following graph that corresponds to the relation given above:





The strongly connected part of `SCC` restricted to pair $\langle \texttt{plus(s x,y)},\texttt{plus(x,y)}\rangle$ is modelled by the following relation, which corresponds to (0):

```
Inductive SCC0 : term → term → term:=
  SCC00: ∀V0 V1, x -[R]*> S(V0) → y -[R]*> V1
                   → plus(x,y) -[SCC0]> plus(V0,V1)
```

An application of RMVERTEX that removes only pair $\langle \texttt{plus(s x,y)},\texttt{plus(x,y)}\rangle$ will generate a module (labelled as `weak`) dedicated to the remaining pairs, for which termination will in turn have to be proven, recursively. Termination of reductions in `SCC` will then be obtained by well-founded induction.

An application of GRAPH will present a graph `G` as a set of components $\texttt{C}_i$ that can be seen as a compatible DAG (hierarchy). When the relations restricted to the $\texttt{C}_i$ have been proven to be terminating, it remains to prove the following lemmas:

```
Lemma Acc_S0: ∀x y, C0 x y → Acc G x.
...
Lemma Acc_Sn: ∀x y, Cn x y → Acc G x.
```

The proof of `Acc_Si` may use any `Acc_Sj` for `j<i`.

The key point is that inductions on the paths in the graph can be simulated by a judicious generation of intermediate lemmas (basically by topological sort) that propagate the desired property along the arcs.

With this technique, our tool was the first one to certify, in a few seconds, termination proofs of systems (from the TPDB `http://www.termination-portal.org/wiki/TPDB`) involving graphs consisting of more than a thousand arcs[5] (See Figure III.2), or of more than 150 vertices[6].

To our knowledge, it is the only effective certification technique for big graphs in COQ.

---

[5]TPDB: `TRS/TRCSR/ExSec11_1_Luc02a_iGM.trs`

[6]TPDB: `TRS/TRCSR/PALINDROME_complete-noand_FR.trs`

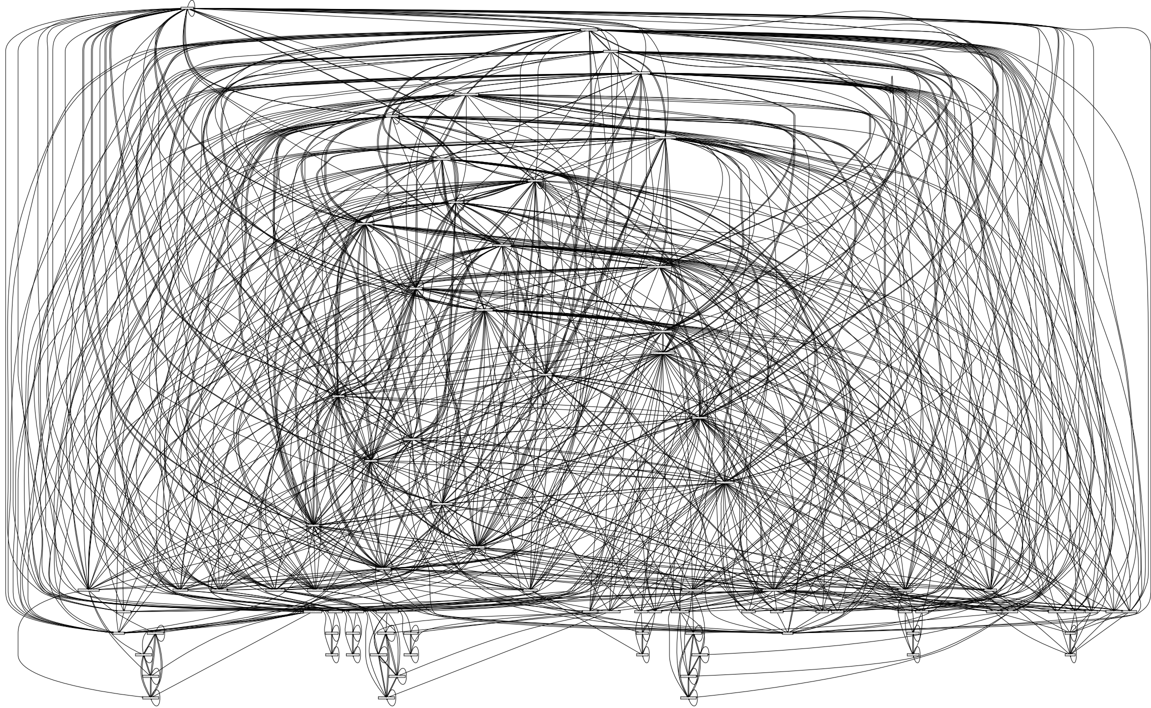


Figure III.2. A practical example from TPDB, consisting of 47 rules, 85 vertices, 1015 arcs.

Deeper Graph Approaches

We proposed recently a deepened version of this implicit graph approach [31], and implemented it in COCCINELLE. The presentation of the relevant connections is given as a *function* the good properties of which are easy to check, and which is computed by the satellite tool while compiling the proof trace. This function may be then taken as a premise in deep versions of theorems. It allows for more local checks of compatibility.

The formalisation in [31] is very similar to the proposal of Thiemann & Sternagel [155]. Concerning the splitting of the graph, both statements are roughly equivalent. A minor difference is that in our setting, the splitting is given as a total function from pairs to integers (having in mind a component rank in the graph), whereas in the other, it is an explicit partition given as a list of sets of pairs, sets that have to be checked as disjoint, which is not necessary in our case (we may even define the function as a relation, provided that all images fulfil the compatibility inequalities).

III.4 Formal Proofs and Criteria

Criteria can be used in formal proofs as instances of generic theorems proven once and for all (deep embedding) or as *ad-hoc* COQ definitions (shallow embedding).

Example of Deep Embedding: 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 over symbols) and a *status* (multiset/lexicographic) for each symbol. The library contains a functor building an RPO from a module of type `Term` and a precedence. It builds the usual properties of RPO (monotonicity, stability) and a proof that if the precedence is well-founded, then so is the RPO. Proofs of termination using RPO 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. A COQ function returning a boolean decides the RPO relation.

This is to date the only full RPO used by certifiers, embedded in particular in CoLoR.

It is worth noticing that the generic RPO uses a strict precedence and a comparison from left to right in the lexicographic case. This is not a restriction: a simple translation from terms into terms, mapping equivalent symbols onto a same symbol, and performing desired permutations (of the subterms under a given symbol with status lex) is both monotonic and stable. Hence the relation defined by comparing the translations of terms by the generic RPO still has the good properties. This can be done during AFS phases, defined as fixpoints, and applied at comparison time.

Example of Shallow Embedding: Polynomials

In our framework a polynomial interpretation is currently defined as a recursive function on terms.

From an interpretation such as the following:

$$[\![\#]\!] = 0; \quad [\![0]\!](x) = x + 1; \quad [\![1]\!](x) = x + 2; \quad [\![+]\!](x, y) = y + x + 1; \quad [\![\widehat{+}]\!](x, y) = 2y + x$$

one can produce a measure: `term` $\rightarrow$ `Z`:

```
Fixpoint measure t { struct t } := match t with
      | (Term plus [x5; x4]) => P_plus (measure x5) (measure x4)
      | (Term O [x4]) => P_O (measure x4)
      | (Term I [x4]) => P_I (measure x4)
      | (Term # []) => P_#
      | _ => A0
      end.
```

where `A0` is the lower bound. The shallow definition of relevant polynomials is straightforward:

```
Definition P_plus (x4:Z) (x5:Z) := 1* x4 + 1* x5 + 1.
Definition P_O (x4:Z) := 1* x4 + 1.
...
```

Polynomials and the measure function are separated for efficiency reasons, as a few properties can be proven on measure without *complete knowledge* of actual polynomials.

Remark that no marked symbol is defined in the signature. Since they can only appear at root position, it is sufficient to wrap the recursive `measure` by a special call:

```
Definition marked_measure t := match t with
| (Term plus [x5; x4]) => P_Marked_plus (measure x5) (measure x4)
| _ => measure t
end.
```

Thus, although the definition of terms is a deep embedding, the measure is defined as a shallow embedding[7]. It is a recursive function on terms directly, and does not refer to a data type of polynomials, substitutions or variables (`x4` above is a COQ variable, it is not a *rewriting* variable which would have been of the form `Var n`). This choice makes 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 `measure` above can be proven by an easy induction on `t`.

The approach involves a procedure to compare even *non-linear* polynomials that results from techniques in Section II.2.1, that is with nonnegative coefficients; it allows for efficient termination proofs using these kinds of interpretations.

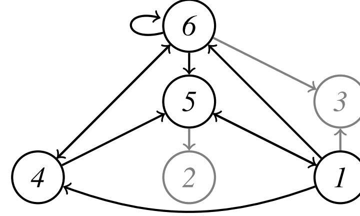
Example 13 *We look again at the Ackermann-Péter function of Example 6, let us recall system R:*

$$
\begin{array}{lcllcl}
\mathit{u11}(\mathit{Aout}(v_0)) & \to & \mathit{Aout}(v_0) & \mathit{u22}(\mathit{Aout}(v_0)) & \to & \mathit{Aout}(v_0) \\
\mathit{u21}(\mathit{Aout}(v_0), v_1) & \to & \mathit{u22}(\mathit{Ain}(v_1, v_0)) & \mathit{Ain}(O, v_0) & \to & \mathit{Aout}(S(v_0)) \\
\mathit{Ain}(S(v_0), O) & \to & \mathit{u11}(\mathit{Ain}(v_0, S(O))) & \mathit{Ain}(S(v_0), S(v_1)) & \to & \mathit{u21}(\mathit{Ain}(S(v_0), v_1), v_0)
\end{array}
$$

We may discover an alternate termination proof, using both polynomial orderings and RPO, that is mixing shallow and deep embedding in a mechanical certification.

The system yields 6 dependency pairs building the following EDG:

$$
\begin{array}{rl}
1: & \langle \widehat{\mathit{u21}}(\mathit{Aout}(V_0), V_1), \widehat{\mathit{Ain}}(V_1, V_0)\rangle \\
2: & \langle \widehat{\mathit{u21}}(\mathit{Aout}(V_0), V_1), \widehat{\mathit{u22}}(\mathit{Ain}(V_1, V_0))\rangle \\
3: & \langle \widehat{\mathit{Ain}}(S(V_0), O), \widehat{\mathit{u11}}(\mathit{Ain}(V_0, S(O)))\rangle \\
4: & \langle \widehat{\mathit{Ain}}(S(V_0), O), \widehat{\mathit{Ain}}(V_0, S(O))\rangle \\
5: & \langle \widehat{\mathit{Ain}}(S(V_0), S(V_1)), \widehat{\mathit{u21}}(\mathit{Ain}(S(V_0), V_1), V_0)\rangle \\
6: & \langle \widehat{\mathit{Ain}}(S(V_0), S(V_1)), \widehat{\mathit{Ain}}(S(V_0), V_1)\rangle
\end{array}
$$



Pairs 2 and 3 are not in strongly connected components. The following polynomial interpretation defines $(\preceq_1, <_1)$ *and, with* RMVERTEX, *takes care of pair 4.*

$$
\begin{array}{lll}
[\![\mathit{Aout}]\!](X_0) = 0 & [\![\mathit{u11}]\!](X_0) = 0 & [\![\mathit{u21}]\!](X_0, X_1) = 0 \\
[\![\mathit{Ain}]\!](X_0, X_1) = 1 & [\![\mathit{u22}]\!](X_0) = 0 & [\![O]\!] = 1 \\
[\![S]\!](X_0) = X_0 + 1 & [\![\widehat{\mathit{u21}}]\!]_1(X_0, X_1) = X_1 + X_0 & [\![\widehat{\mathit{Ain}}]\!]_1(X_0, X_1) = X_0
\end{array}
$$

The remaining strongly connected component consists of pairs 1, 5 and 6. Pairs 5 and 6 are pruned using $(\preceq_2, <_2)$ *defined with the conjunction of*

- *an AFS permuting arguments of u21 and* $\widehat{u21}$*, and projecting u22, u11 and Aout to their first argument,*
- *an RPO with the precedence O = Aout < S = u22 < Ain =* $\widehat{Ain}$ *= u21 =* $\widehat{u21}$ *< u11 and all statuses set to lex left-right.*

The remaining $\twoheadrightarrow_{\{1\},R}$ *leads to an empty problem as there is no circuit left in the graph.*

These steps are summarised on the proof tree (guards and trivial steps omitted):

[7] In particular polynomials are almost (and could be) completely handled by the trace generation part of CiME since our library COCCINELLE focuses on deep embedding.

$$
\textsc{DP}\;\dfrac{\textsc{Graph}\;\dfrac{\textsc{RmVertex}(\preceq_1,<_1)\;\dfrac{\textsc{Graph}\;\dfrac{\textsc{RmVertex*}(\preceq_2,<_2)\;\dfrac{\textsc{Graph}\;\dfrac{\emptyset}{\mathsf{SN}(\twoheadrightarrow_{\{1\},R})}\;\mathsf{Decomp}(\emptyset)\quad\cdots}{\mathsf{SN}(\twoheadrightarrow_{\{1,5,6\},R})}}{\mathsf{SN}(\twoheadrightarrow_{\{1,5,6\},R})}\;\mathsf{Decomp}(\{1,5,6\}\cdots)\quad\cdots}{\mathsf{SN}(\twoheadrightarrow_{\{1,4,5,6\},R})}\quad\cdots}{\mathsf{SN}(\twoheadrightarrow_{\{1,2,3,4,5,6\},R})}\;\mathsf{Decomp}(\{1,4,5,6\}\cdots)}{\mathsf{SN}(\rightarrow_R)}
$$

The proof trace, as a CPF (XML) file, follows the same structure of rule applications The system is given

```
<proof> ...
  <input>
    <trsInput> <trs> ... </trs> </trsInput>
  </input>
```

then the DP specifying $\twoheadrightarrow_{\mathsf{DP}(R),R}$*:*

```
<dpTrans>
  <dps> ... </dps>
```

which is proven to be terminating using GRAPH*. Termination proof of* $\twoheadrightarrow_{\{1,4,5,6\},R}$ *is described.*

```
<dpProof>
  <depGraphProc>
    <component>
      <dps> ...  <!-- DP #1,4,5,6 --> ... </dps>
```

Pair 4 is removed by a polynomial interpretations, application of RMVERTEX *(*`redPairProc`*).*

```
<dpProof>
  <redPairProc>
    <orderingConstraintProof>
      <redPair>
        <interpretation> <type> <polynomial> ... </polynomial> </type>
          ...
        </interpretation>
      </redPair>
    </orderingConstraintProof>
```

The remaining $\twoheadrightarrow_{\{1,5,6\},R}$ *is then proven to be terminating, using* GRAPH*. . .*

```
<dps> ... <!-- DP #1,5,6 --> ... </dps>
<dpProof>
  <depGraphProc>
    <component>
      <dps> ... <!-- DP #1,5,6 --> ... </dps>
```

. . . and RMVERTEX *with an RPO, leaving no circuit in the remaining* $\twoheadrightarrow_{\{1\},R}$*.*

```
<dpProof>
  <redPairProc>
    <orderingConstraintProof>
      <redPair>
        <pathOrder>
            ...
        </pathOrder>
      </redPair>
    </orderingConstraintProof>
    <dps> ...  <!-- DP #1 --> ... </dps>
    <dpProof>
      <depGraphProc>
        <component>
          <dps> ... <!-- DP #1 --> ... </dps>
          <realScc>false</realScc>
        </component>
      </depGraphProc>
```

What remains in the file is the closure of rules until the first GRAPH*, the trivial branches addressing the non-cycling pairs 2 and 3, and we are done.*

Finally, one obtains a `.v` *file that can be compiled by* COQ*, using* COCCINELLE*, in few seconds, thus certifying the proof. Let us have a look at its structure (most terms, and proofs are omitted).*

The two pairs outside connected component, and the component {1,4,5,6}: DP_R_scc$_2$

```
Inductive DP_R_non_scc0 : term → term → Prop :=
   (* <Ain(s(m),0),u11(Ain(m,s(0)))> *) .

Lemma acc_DP_R_non_scc0 :
  ∀ x sigma,
   In x (List.map (apply_subst sigma)
        [(Term u11 [(Term Ain [(Var 1); (Term s [(Term 0 [])])])])]) →
    Acc (sdp.Rcdp_min R_rules DP_R) x.
... Qed.

Inductive DP_R_non_scc1 : term → term → Prop :=
   (* <u21(Aout(n),m),u22(Ain(m,n))> *).

Lemma acc_DP_R_non_scc1 :
 ∀ x sigma,
  In x (List.map (apply_subst sigma)
       [(Term u22 [(Term Ain [(Var 1); (Var 2)])])]) →
   Acc (sdp.Rcdp_min R_rules DP_R) x.
 ... Qed.

Inductive DP_R_scc2 : term → term → Prop :=
   (* <Ain(s(m),s(n)),u21(Ain(s(m),n),m)> *)
   (* <Ain(s(m),s(n)),Ain(s(m),n)> *)
   (* <u21(Aout(n),m),Ain(m,n)> *)
   (* <Ain(s(m),0),Ain(m,s(0))> *) .
```

Note that the lemmas above are the actual presentation of lemmas mentioned in Section III.3. They state that any instance of a DP right-hand side in the relevant `scc` *(component or not) is accessible.*

The component is now split in {1,5,6} and {4}. . .

```
Module WF_DP_R_scc2.
 Inductive DP_R_scc2_weak : term → term → Prop :=
    (* <Ain(s(m),s(n)),u21(Ain(s(m),n),m)> *)
    (* <Ain(s(m),s(n)),Ain(s(m),n)> *)
    (* <u21(Aout(n),m),Ain(m,n)> *) .

 Inductive DP_R_scc2_strict : term → term → Prop :=
    (* <Ain(s(m),0),Ain(m,s(0))> *) .
```

. . . and {1,5,6} is itself split in {1} and {5,6}. The former is proven to be terminating. We can see one module per application of GRAPH *and* RMVERTEX*.*

```
Module WF_DP_R_scc2_weak.
 Inductive DP_R_scc2_weak_scc0 : term → term → Prop :=
    (* <Ain(s(m),s(n)),u21(Ain(s(m),n),m)> *)
    (* <Ain(s(m),s(n)),Ain(s(m),n)> *)
    (* <u21(Aout(n),m),Ain(m,n)> *) .

 Module WF_DP_R_scc2_weak_scc0.
  Inductive DP_R_scc2_weak_scc0_weak : term → term → Prop :=
     (* <u21(Aout(n),m),Ain(m,n)> *).
```

```
Inductive DP_R_scc2_weak_scc0_strict : term → term → Prop :=
  (* <Ain(s(m),s(n)),u21(Ain(s(m),n),m)> *)
  (* <Ain(s(m),s(n)),Ain(s(m),n)> *).

Module WF_DP_R_scc2_weak_scc0_weak.
 Inductive DP_R_scc2_weak_scc0_weak_non_scc0: term → term → Prop :=
   (* <u21(Aout(n),m),Ain(m,n)> *).

 Lemma acc_DP_R_scc2_weak_scc0_weak_non_scc0 :
  ∀ x sigma,
   In x (List.map (apply_subst sigma)
        [(Term Ain [(Var 1); (Var 2)])]) →
    Acc (sdp.Rcdp_min R_rules DP_R_scc2_weak_scc0_weak) x.
 ... Qed.

 Lemma wf :
  well_founded (sdp.Rcdp_min R_rules DP_R_scc2_weak_scc0_weak).
 ... Qed.
End WF_DP_R_scc2_weak_scc0_weak.
```

After relevant definitions of orderings, {1,5,6} is proven to be terminating, then all termination lemmas are proven, using previous ones as premises, as described in Section III.3. Notice in particular the organisation of lemmas `acc_DP...` *which deal with the graph structure.*

```
 Lemma wf :
 well_founded (sdp.Rcdp_min R_rules DP_R_scc2_weak_scc0).
 ... Qed.
End WF_DP_R_scc2_weak_scc0.

Lemma acc_DP_R_scc2_weak_scc0 :
 ∀ x sigma,
  In x (List.map (apply_subst sigma)
       [(Term u21 [(Term Ain [(Term s [(Var 1)]); (Var 2)]); (Var 1)]);
        (Term Ain [(Term s [(Var 1)]); (Var 2)]);
         (Term Ain [(Var 1); (Var 2)])]) →
   Acc (sdp.Rcdp_min R_rules DP_R_scc2_weak) x.
 ... Qed.

 Lemma wf :
  well_founded (sdp.Rcdp_min R_rules DP_R_scc2_weak).
 ... Qed.
End WF_DP_R_scc2_weak.

 Lemma wf : well_founded (sdp.Rcdp_min R_rules DP_R_scc2).
 ... Qed.
End WF_DP_R_scc2.

Lemma acc_DP_R_scc2 :
 ∀ x sigma,
  In x (List.map (apply_subst sigma)
       [(Term u21 [(Term Ain [(Term s [(Var 1)]); (Var 2)]); (Var 1)]);
       (Term Ain [(Term s [(Var 1)]); (Var 2)]);
         (Term Ain [(Var 1); (Var 2)]);
          (Term Ain [(Var 1); (Term s [(Term 0 [])])])]) →
   Acc (sdp.Rcdp_min R_rules DP_R) x.
... Qed.
```

```
 Lemma wf : well_founded (sdp.Rcdp_min R_rules DP_R).
 ... Qed.
End WF_DP_R.

Lemma wf : well_founded (one_step R_rules).
... Qed.
```

Experiments

The following table summarises our experiments with C*i*ME 3, on a 3GHz, 16GB computer running Debian Linux, using the following proof discovery strategy: lexicographical composition of orderings (rule removal) that is Manna-Ness using (linear and simple) polynomials over integers (coef. bounded by 3) and full RPO, composed with DP, dependency graphs, subterm and various orderings: full RPO with AFS, polynomials over integers and over matrices (3×3, coef. bounded by 1). This run does not use parallel constraint solving. Time limits are 10s for polynomials, 25s for RPO, 15s for matrices. We used MINISAT2 to solve orderings constraints[8]. Tests were run on 1391 problems of the TPDB in the TRS category. *With this strategy* that matches the currently implemented certification techniques, C*i*ME 3 produces 654 termination proofs, that is 654 CPF files. Note that the termination engine is restricted here to match the available techniques in the certification engine. The table below summarises discovery times, with an average time of 5.1s. Actually the 612 fastest discoveries (that is 93.6% of the proofs) are obtained with an average time of 2s.

Proven	$< 2s$	$< 10s$	$< 30s$
654	445 (68%)	612 (93.6%)	641 (98%)

The next step is to translate CPF files into `.v` inputs for COQ, and to compile them in order to have the proofs certified. Translation and compilation times are summarised below. The timeout is set to 500s. There are no failures, non-certified proofs come from timeout at compile time, mostly because of huge and intricate graphs that involve many subproofs.

Certified	$< 10s$	$< 20s$	$< 60s$	$< 120s$	$< 500s$	Timeout
632	334 (51%)	433 (66.2%)	541 (82.7%)	593 (90.7%)	632 (96.6%)	22 (3.4%)
Av. time	*5.8s*	*7.4s*	*12.6s*	*19.1s*	*47.2s*	

Note that this set of examples includes, in particular, a μ-CRL specification of communicating processes (377 rewriting rules) or transformations of context-sensitive programs (several problems each containing more than 50 rules and leading to dependency graphs of about 1000 arcs and 80 vertices, see Figure II.3). As already mentioned, these problems are handled well by our approach.

Some criteria take more time to certify than others; the overall time is thus a function of the proof search strategy. Nevertheless, the times for proof search, and for certification (with an average of 12.6s for 82.7% of the proofs), are reasonably small for C*i*ME 3 to be used as a practical tool in everyday development involving term rewriting systems and proof assistants, even if they cannot really compare with checking times by an extracted tool like CeTA.

Some Comments

The termination competition has given us the opportunity to uncover some problems within some automatic provers and the recent introduction of certifiers has allowed for an interesting evolution towards increasingly more reliable tools. This is particularly noticeable when dealing with complex and sizeable

[8] http://minisat.se/MiniSat.html

systems where one could not envisage a human input for verification. In this context, the creation of a common trace format encourages a prover to claim to be the one which produces the greatest number of accurate certified proofs. I believe that what motivates the certifiers is to be party to this claim by playing their part in providing the largest range of certified methods.

One is, nevertheless, entitled to question the merits and the benefits of a competition between certifiers. What, indeed, can we learn from it? At present, certifiers are given credit according to the proofs which they have validated, in other words as they formalise criteria that are distinct, essentially according to the provers' proving strategy. To research RPO and applications of the subterm criterion will benefit A3PAT whereas to research arctic matrices and usable rules will benefit CeTA, and both options will disadvantage CoLoR. The conclusion which can be drawn from the results is that it highlights more an option to research proofs, for a whole of problems from a database, rather than a measure for comparison of tools.

I am personally of the view that the future lies more with platforms that will allow verification by different certifiers and with competitions pointing to the safest provers. This is what we are working towards with team CeTA.

Collateral benefits

There is an interesting side effect of mechanisation that should be emphasised. A benefit of abstract formalisations is that keeping them as general as possible may lead to relaxing premises of theorems. This is particularly noticeable in the context of constructive proofs, as most proofs of the literature are classical. Developing a formal mechanical model of former results thus amounts to thinking the proofs anew. There are two particular examples from the A3PAT projects: the first full formalisation of the subterm criterion [83] for which we proposed an extension [31], and an extended notion of matrix-based interpretations by Courtieu, Gbedo and Pons [41, 42].

As always, all deliverable of the project (tools, libraries, manuals, etc.) are available from the A3PAT web page: `http://a3pat.ensiie.fr` under the CeCILL-C license.

Chapter IV

Conclusion and Perspectives

This past decade has been very active with reference to automated and formal proof. In various competitions like CASC, the competition affiliated to the *Conference on Automated Deduction* (CADE), SMT, affiliated to the conference on *Computer-Aided Verification* (CAV), or the already mentioned Termination competition, this can be monitored with two indicators: the number of problems solved, and the fast increasing number of tools engaged.

Automation of Termination Proofs

Regarding automated provers for termination of rewriting alone, the pioneer tools (mainly C*i*ME [36] and APROVE [71]) have been joined by many newcomers, implementing new techniques or dedicated to specific areas (strategies, complexity, etc.): TTT [83], TTT2 [99], MU-TERM [116], TERMPTATION [22], TORPA [161], JAMBOX[1], etc. We contributed to motivate this activity by defining incremental and modular termination techniques, termination approaches for systems modulo AC, context-sensitive systems, conditional systems with membership, techniques to discover orderings, etc. The constant care for automation made us implement our results, in particular in C*i*ME.

In the context of a dependency pair approach, one of the very active points regards new ways to remove rules, or to instantiate the rule RMVERTEX, that is to prune irrelevant pairs. To this end, new orderings have been introduced, yet there is still an important need for similar improvements regarding the expressive features I mentioned. For example, matrix-based interpretations are very hard to make compatible with AC without losing most of their termination power. It is even unclear how one can define interpretations that are not linear.

Automation and Mechanisation

Project A3PAT contributed significantly to adding automation to proof assistants, using both deep and shallow embeddings, and implementing a scheme of satellite tools. It focuses primarily but is not limited to the certification of termination proofs, and successful experiments target unification/disunification problems (using an external oracle for occur-check) , and proofs of confluence (for now with termination and local confluence).

Interesting perspectives come from certification. It is of course very important to certify as many techniques as possible, in all extensions of rewriting that we met in practice: modulo theories, with strategies, with conditions, etc. Many termination proofs cannot be certified yet. For instance proofs involving strategies or techniques based on strategies (like switching to innermost for special classes of systems) are not supported by current certifiers[2]. Regarding orderings, path orderings for AC-termination are not formalised, etc. Here again, I shall emphasise that formalisation could lead to weakening the premises and developing new criteria.

---

[1] `http://joerg.endrullis.de/jambox.html`

[2] Even if the relevant formalisations already exist. This is the case for context-sensitive dependency pairs or innermost techniques, implemented in COCCINELLE, but not yet operational in A3PAT.

As mentioned in Chapter III, there are now three approaches for certification of termination proofs, namely A3PAT, CeTA and CoLoR, which do not subsume each other. What to do when a proof is discovered using criteria that are not all supported by one of the approaches? I stated on page 52 that the trace language was designed to be as modular as possible, and as a matter of fact, CeTA and A3PAT are developing the trace language CPF in such a way that we are very close now to being able to build a platform that can use several certifiers simultaneously, on distinct applications of criteria, in the proof tree. Such a platform would be very interesting and would increase the number of problems certified to be terminating, as it would not restrict certified proofs to the techniques implemented in one certifier or another. Thus it would contribute to raise the level of confidence in automated provers.

Finally, the possibility of delegating a proof of termination opens an interesting way to automate termination proofs for *functions*. Whether one uses **`Function`** or **`Fixpoint`**, defining a function in COQ presently involves a mandatory decreasing argument that ensures termination. This argument may be structurally decreasing, one may also reason about its proof of decrease. Adding an automated way to provide such an argument would be a considerable help throughout developments with such a proof assistant, and that has to be further investigated.

To Other Paradigms[3]

Although I did not mention them in the body of this dissertation, some of my parallel research interests, while not being in the mainstream of this document, contribute to programs verification.

In 2001 I was appointed to a post-doctoral position in the European project IST VerifiCard, in the task that aimed at the proof of Java programs with the help of the COQ proof assistant. To this end, I was involved in the initial development of the KRAKATOA tool [121].

One of the challenges was that Java programs are *imperative* style programs, that is adapted to a Floyd-Hoare approach [64, 84], while the Curry-Howard based assistant COQ is well-suited to the verification of *functional* programs. To help bridge this gap, Filliâtre proposed an original technique [59, 60] that is the base of the tool WHY [61, 62]. WHY is a stand-alone tool aimed to produce verification conditions for programs written in its own language specially designed for program certification. This language is functional with limited imperative features, namely exceptions and references. The tool acts as a verification conditions generator, involving a calculus of weakest preconditions: the original program is correct if the produced formulas are valid.

To tackle the verification of Java programs, we developed an approach that is multi-provers, and on three levels. This approach is made concrete with the tool KRAKATOA, and is detailed in [121].

Firstly, annotated Java/Javacard programs are translated into entries for WHY. This is the task for which KRAKATOA [121] is dedicated. Programs submitted to KRAKATOA are Java programs that have been annotated in JML (*Java Modelling Language*[4]). In particular, the JML annotations typically state some class invariant, pre-conditions and the post-conditions that are supposed to hold at the end of a call if pre-conditions where valid at call time. The result is a program in the language of WHY, that is functional with exceptions and references.

We defined at that time two logical models for applets. A high-level model, based on values, and a memory model that allows the handling of addresses. The memory model is based on the work of Burstall [24] and Bornat [18]. Roughly speaking, each non-static field of a class is modelled by a table indexed with addresses. See [121] for details. We claim that the resulting code enjoys the same semantics as the original program.

Once the original annotated program is translated, WHY produces verification conditions out of this functional code. Finally, the proof of validity for these conditions are left to various provers (HARVEY, SIMPLIFY, etc.), or to a mechanised assistant for an interactive proof (COQ, ISABELLE/HOL, etc.).

---

[3]... and beyond! [108]

[4]`http://www.cs.iastate.deu/~leavens/JML/index.shtml`

Amongst the first tools dedicated to formal verification of programs written in widely used languages, KRAKATOA has been successfully applied to the study of Javacard applets [26] provided by industrial partners (Gemalto, etc.). The original approach served as a basis for other verification tools, and was adapted to the analysis of programs written in C (Caduceus [62]).

The fact that several tools (automated or interactive) may be used to prove different verification conditions for a single program makes it very difficult to obtain an axiom-free formal certification. As a matter of fact, it is somehow a general case of the combination of termination provers mentioned above, this time making cohabit various types of proofs, from various provers (SMT, etc.); it strongly advocates the concern of automation for proof assistants. Investigating approaches for multi-provers certification, possibly based on proof traces/certificates, is indeed a challenging perspective.

Imperative Style

The journey I proposed focused on termination for first-order term rewriting, that is close to the functional declarative style. However a wide variety of techniques can enrich this approach and adapt it to further programming paradigms. For instance, proving the termination of `while` loops in imperative languages raises very interesting issues. Depending on guards of the loop, and the nature of side effects in its body, some termination techniques might borrow arguments from unexpected domains, like linear algebra and fix-point theorems [156]. Mahboubi, Melquiond and I are currently working on the certification of similar techniques. In addition, we expect to achieve, thanks to their formalisation, an extension of the aforementioned techniques to less restrictive guards or bodies.

Distributed Computing

The past decade has also witnessed the coming of a widely distributed and complex information, for instance with the Internet. Technological breakthroughs regarding computers and networks have made it possible for the success of large scaled distributed applications. In this extremely connected world, distributed algorithms are an essential part. For example, a system running across several sites needs distributed algorithms to ensure basic services and properties.

In most situations, this is highly significant, as deficient distributed systems may induce very serious consequences: one possible instance is a huge database over several sites where remote operations may corrupt integrity, or in a system for which fault tolerance is crucial. However, distributed algorithms remain very difficult to develop and to certify, even for experts. They involve many problems unique to their distributed nature, and to which the developer must pay attention, both in specification and conception phases. This is particularly the case when processes have a *local* knowledge only, and they are supposed to converge towards a *global* property (that is of the whole network): leader election, fault tolerance, termination, etc.

Since distributed algorithms issues may appear in critical applications (for example regarding fault tolerant behaviours), producing safe distributed algorithms is a key issue. So far, the correctness of, for example, the rapidly emerging sensor networks has been partially explored either via restrictive simulations and experiments or via analytical tools built on top of restrictive assumptions (e.g. the nodes characteristics and status do not change during the whole system execution, etc.). These assumptions might be completely unrealistic in actual networks and thus disastrous in practice. I do believe that mechanical and automated tools have to be involved, so as to obtain certified local and global properties of such systems, and that mechanical certification of distributed processes is a key challenge for the future.

In this context, again, an interesting solution would be a framework dedicated to specification and proof-based development. Defining such an environment requires the discovery of adequate semantics for the specifications and definitions. A critical part is then the provision of well-suited proof techniques and automated generation of verification conditions for a skeptical proof assistant, and (semi-) automated tools to discharge them as much as possible.

I tackled this topic recently, with the coordination of project Pactole, which aims to provide basis for such a formal provable framework, using recent advances in automated proving and related areas in order to prove the correctness of localised distributed protocols in dynamic mobile sensor networks.

Rewriting techniques may appear at an early stage in a certification chain, depending on the model one may study: in Local Computation systems for instance, but also in alternate approaches based on λ-calculi (in this case, conditional rules are involved). Later on, when it comes to the definition of (semi-) decision procedures, efficient rewriting techniques may prove useful when extended to the context of distributed processes.

We focus on Local Computation systems and in particular on their graph rewriting model [112, 113, 13] and its extensions: dynamic network, probabilistic behaviours, etc. This model enjoys high abstraction (thanks to rewriting) and yielded important results.

In this context, termination of a distributed algorithm boils down to the termination of its translation as a label rewriting system over some graphs. Currently, it may be noticed that termination is proven by hand, by guessing an adequate measure decreasing with each rewriting step. From a rewriting expert point of view, this is simply an adaptation of the Manna-Ness criterion applied with a polynomial ordering. A first step is to automate the search of a Manna-Ness measure within an automated prover that already contains the search for polynomial orderings. A second step is to extend state-of-the-art termination criteria for *term* rewriting system (dependency pairs, modularity, etc.) and orderings (matrices, etc.) to label graph rewriting. A translation of graph rewriting into AC-rewriting could be helpful at this stage, in order to simulate the ‘unordered character’ of (non-oriented) graphs, where it makes no sense to state that a vertex is above or below another one. Because of the arbitrary structure of the graphs, it may not be possible to obtain extensions in the general case, a solution would be to investigate special topologies such as rings, grids, etc. Formal certificates are of course wanted, and dedicated libraries will have to be extended accordingly.

Invariants are another class of required properties: liveness, soundness, self-stabilisation, etc. There have been a few attempts regarding liveness and term rewriting systems. Arts proposes to verify liveness properties of communicating processes specified in μ-CRL [80]. His approach consists in translating the μ-CRL specification in a system for which termination implies liveness of the processes. Specifications may contain many rules, nevertheless modular techniques manage to prove some of them to be terminating with full automation. Arts and Giesl address verification of liveness for Erlang processes using rewriting systems [10]. Giesl and Zantema tackle liveness using strategies [74]. These approaches, as well as those of Koprowski and Zantema [98], are restricted to liveness, however it would be very interesting to have such translations of other properties of distributed computation into properties of rewriting systems, moreover in the graph rewriting frameworks in which we are interested.

Soundness involves some interpreted functions and predicates which are beyond the scope of tools like C*i*ME. However, some SMT-provers, like Alt-Ergo[5], are able to deal with such properties in the context of sequential program verification. An important work of formalisation is required to define, in such tools, certifiable procedures for distributed-specific theories.

An environment for proof and development of trustworthy distributed algorithms will have to link together results on model definition, generation of verification conditions, automated proof and its certification. Existing platforms (Visidia [132], Netquest [111], etc.) constitute interesting candidates for visualisation and simulation of the specified algorithms. The challenge is to equip them with languages powerful enough to specify properties in the scope of our studies, and with proof mechanisms for assisted and automated certification, thus defining a complete certification chain.

The techniques we contributed to developing during the past decade, regarding automation, regarding certification, and the experience gained on program verification, are surely undeniable benefits to reach this objective on this new path.

[5]Development started within the A3PAT project, `http://alt-ergo.lri.fr`

Bibliography

[1] B. Alarcón, F. Emmes, C. Fuhs, J. Giesl, R. Gutiérrez, S. Lucas, P. Schneider-Kamp, and R. Thiemann. Improving Context-Sensitive Dependency Pairs. In I. Cervesato, H. Veith, and A. Voronkov, editors, *15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'08*, volume 5330 of *Lecture Notes in Computer Science*, pages 636–651. Springer-Verlag, 2008.

[2] B. Alarcón, R. Gutiérrez, J. Iborra, and S. Lucas. Proving Termination of Context-Sensitive Rewriting with MU-TERM. *Electronic Notes in Theoretical Computer Science*, 188:105–115, 2007.

[3] B. Alarcón, R. Gutiérrez, and S. Lucas. Context-Sensitive Dependency Pairs. In S. Arun-Kumar and N. Garg, editors, *XXVI International Conference on Foundations of Software Technology and Theoretical Computer Science, FST&TCS'06*, volume 4337 of *Lecture Notes in Computer Science*, pages 297–308, Kolkata, India, 2006. Springer-Verlag.

[4] B. Alarcón, R. Gutiérrez, and S. Lucas. Improving the Context-Sensitive Dependency Graph. *Electronic Notes in Theoretical Computer Science*, 188:91–103, 2007.

[5] B. Alarcón and S. Lucas. Termination of Innermost Context-Sensitive Rewriting Using Dependency Pairs. In F. Wolter, editor, *Proc. of VI International Symposium on Frontiers of Combining Systems, FroCoS'07*, volume 4720 of *Lecture Notes in Computer Science*, pages 73–87, Liverpool, United Kingdom, 2007. Springer-Verlag.

[6] E. Annov, M. Codish, J. Giesl, P. Schneider-Kamp, and R. Thiemann. A SAT-based Implementation for RPO Termination. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (Short Paper)*, November 2006.

[7] P. Arrighi and G. Dowek. Linear-algebraic lambda-calculus: higher-order, encodings, and confluence. In A. Voronkov, editor, *19th International Conference on Rewriting Techniques and Applications (RTA 08)*, volume 5117 of *Lecture Notes in Computer Science*, pages 17–31, Hagenberg, Austria, July 2008. Springer-Verlag.

[8] T. Arts and J. Giesl. Automatically Proving Termination Where Simplification Orderings Fail. In M. Bidoit and M. Dauchet, editors, *Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, Lille, France, Apr. 1997. Springer-Verlag.

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

[10] T. Arts and J. Giesl. Verification of Erlang Processes by Dependency Pairs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1,2):39–72, 2001.

[11] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[12] L. Bachmair and D. A. Plaisted. Termination orderings for associative-commutative rewriting systems. *Journal of Symbolic Computation*, 1(4):329–349, Dec. 1985.

[13] M. Bauderon, Y. Métivier, M. Mosbah, and A. Sellami. From local computations to asynchronous message passing systems. Technical Report RR-1271-02, LaBRI, 2002.

[14] A. Ben Cherifa and P. Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9:137–159, 1987.

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

[16] G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet. Algorithms with polynomial interpretation termination proof. *Journal of Functional Programming*, 11(1):33–53, Mar. 2001.

[17] R. Bonichon, D. Delahaye, and D. Doligez. Zenon : An Extensible Automated Theorem Prover Producing Checkable Proofs. In N. Dershowitz and A. Voronkov, editors, *14th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 07)*, volume 4790 of *lncs*, pages 151–165, Yerevan, Armenia, Oct. 2007. Springer-Verlag.

[18] R. Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[19] P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285:155–185, 2002.

[20] C. Borralleras. *Ordering-Based Methods for Proving Termination Automatically*. PhD thesis, Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Barcelona, Spain, 2003.

[21] C. Borralleras and A. Rubio. Monotonic AC-compatible semantic path orderings. In R. Nieuwenhuis, editor, *14th International Conference on Rewriting Techniques and Applications (RTA 03)*, volume 2706 of *Lecture Notes in Computer Science*, pages 279–295, Valencia, Spain, June 2003. Springer-Verlag.

[22] C. Borralleras and A. Rubio. Termptation. In A. Rubio, editor, *Extended Abstracts of the 6th International Workshop on Termination, WST'03*, June 2003. Technical Report DSIC II/15/03, Univ. Politécnica de Valencia, Spain.

[23] F. Branqui, S. Coupet-Grimal, W. Delobel, S. Hinderer, and A. Koprowski. CoLoR, a Coq Library on Rewriting and termination. In A. Geser and H. Sondergaard, editors, *Extended Abstracts of the 8th International Workshop on Termination, WST'06*, Aug. 2006.

[24] R. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[25] P. Castréran and E. Contejean. Cantor. User contributions to the Coq system.

[26] N. Cataño, M. Gawlowski, M. Huisman, B. Jacobs, C. Marché, C. Paulin, E. Poll, N. Rauch, and X. Urbain. Logical Techniques for Applet Verification. Deliverable 5.2, IST VerifiCard project, 2003. `http://www.cs.kun.nl/VerifiCard/files/deliverables/deliverable_5_2.pdf`.

[27] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 285(2):187–243, Aug. 2002.

[28] P. Codognet and D. Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.

[29] É. Contejean. The Coccinelle library for rewriting.

[30] É. Contejean. A certified AC matching algorithm. In V. van Oostrom, editor, *15th International Conference on Rewriting Techniques and Applications (RTA 04)*, volume 3091 of *Lecture Notes in Computer Science*, pages 70–84, Aachen, Germany, June 2004. Springer-Verlag.

[31] É. Contejean, P. Courtieu, J. Forest, A. Paskevich, O. Pons, and X. Urbain. A3PAT, an Approach for Certified Automated Termination Proofs. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 10)*, pages 63–72. ACM, 2010.

[32] É. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Certification of automated termination proofs. In B. Konev and F. Wolter, editors, *6th International Symposium on Frontiers of Combining Systems (FroCos 07)*, volume 4720 of *Lecture Notes in Artificial Intelligence*, pages 148–162, Liverpool,UK, Sept. 2007. Springer-Verlag.

[33] É. Contejean, J. Forest, and X. Urbain. Deep-Embedded Unification. Technical Report 1547, Cédric, 2008.

[34] E. Contejean and C. Marché. CiME: Completion Modulo E. In H. Ganzinger, editor, *7th International Conference on Rewriting Techniques and Applications (RTA 96)*, volume 1103 of *Lecture Notes in Computer Science*, pages 416–419, New Brunswick, NJ, USA, July 1996. Springer-Verlag.

[35] É. Contejean, C. Marché, B. Monate, and X. Urbain. CiME version 2, 2000.

[36] É. Contejean, C. Marché, B. Monate, and X. Urbain. Proving termination of rewriting with CiME. In A. Rubio, editor, *Extended Abstracts of the 6th International Workshop on Termination, WST'03*, pages 71–73, June 2003. Technical Report DSIC II/15/03, Univ. Politécnica de Valencia, Spain.

[37] E. Contejean, C. Marché, and L. Rabehasaina. Rewrite systems for natural, integral, and rational arithmetic. In H. Comon, editor, *8th International Conference on Rewriting Techniques and Applications (RTA 97)*, volume 1232 of *Lecture Notes in Computer Science*, pages 98–112, Barcelona, Spain, June 1997. Springer-Verlag.

[38] É. Contejean, C. Marché, A. P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):325–363, 2005.

[39] T. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[40] P. Courtieu, J. Forest, and X. Urbain. Certifying a Termination Criterion Based on Graphs, Without Graphs. In C. Muñoz and O. Ait Mohamed, editors, *21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'08)*, volume 5170 of *Lecture Notes in Computer Science*, pages 183–198, Montréal, Canada, Aug. 2008. Springer-Verlag.

[41] P. Courtieu, G. Gbedo, and O. Pons. Matrix interpretations revisited. In A. Geser, editor, *Extended Abstracts of the 10th International Workshop on Termination, WST'09*, Leipzig, Germany, Aug. 2009.

[42] P. Courtieu, G. Gbedo, and O. Pons. Improved Matrix Interpretation. In J. van Leeuwen, A. Muscholl, D. Peleg, J. Pokorný, and B. Rumpe, editors, *36th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2010)*, volume 5901 of *Lecture Notes in Computer Science*, pages 283–295, Špindlerův Mlýn, Czech Republic, Jan. 2010. Springer-Verlag.

[43] M. Dauchet. Simulation of a Turing machine by a left-linear rewrite rule. In N. Dershowitz, editor, *International Conference on Rewriting Techniques and Applications (RTA 89)*, volume 355 of *Lecture Notes in Computer Science*, Chapel Hill, U.S.A., Apr. 1989. Springer-Verlag.

[44] C. Delor and L. Puel. Extension of the associative path ordering to a chain of associative-commutative symbols. In C. Kirchner, editor, *5th International Conference on Rewriting Techniques and Applications (RTA 93)*, volume 690 of *Lecture Notes in Computer Science*, pages 389–404, Montreal, Canada, June 1993. Springer-Verlag.

[45] É. Deplagne. Sequent Calculus Viewed Modulo. In Catherine Pilière, editor, *Fifth ESSLLI Student Session*, pages 66–76, University of Birmingham, 2000.

[46] N. Dershowitz. A note on simplification orderings. *Information Processing Letters*, 9(5):212–215, Nov. 1979.

[47] N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, Mar. 1982.

[48] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1):69–115, Feb. 1987.

[49] N. Dershowitz. 33 examples of termination. In H. Comon and J.-P. Jouannaud, editors, *Term Rewriting*, volume 909 of *Lecture Notes in Computer Science*, pages 16–26. French Spring School of Theoretical Computer Science, Springer-Verlag, 1995.

[50] N. Dershowitz. Hierarchical termination. In N. Dershowitz and N. Lindenstrauss, editors, *Fourth International Workshop on Conditional and Typed Rewriting Systems (Jerusalem, Israel, July 1994)*, volume 968, pages 89–105, Berlin, 1995. Springer-Verlag.

[51] N. Dershowitz. Termination Dependencies. In A. Rubio, editor, *Extended Abstracts of the 6th International Workshop on Termination, WST'03*, June 2003. Technical Report DSIC II/15/03, Univ. Politécnica de Valencia, Spain.

[52] N. Dershowitz and J.-P. Jouannaud. Notations for rewriting. *EATCS Bulletin*, 43:162–172, 1990.

[53] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, Aug. 1979.

[54] F. Durán, S. Lucas, J. Meseguer, C. Marché, and X. Urbain. Proving termination of membership equational programs. In *ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation (PEPM 04)*, Verona, Italy, Aug. 2004. ACM Press.

[55] F. Durán, S. Lucas, J. Meseguer, C. Marché, and X. Urbain. Proving operational termination of membership equational programs. *Higher-Order and Symbolic Computation*, 21(1–2):59–88, 2008.

[56] J. Endrullis, R. C. de Vrijer, and J. Waldmann. Local Termination. In R. Treinen, editor, *20th International Conference on Rewriting Techniques and Applications (RTA 09)*, volume 5595 of *Lecture Notes in Computer Science*, pages 270–284, Brasília, Brazil, July 2009. Springer-Verlag.

[57] J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *Journal of Automated Reasoning*, 40(2-3):195–220, 2008.

[58] M. Fernandez and J.-P. Jouannaud. Modular Termination of Term Rewriting Systems Revisited. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification*, volume 906 of *Lecture Notes in Computer Science*, pages 255–272, S. Margherita, Italy, 1994. Springer-Verlag.

[59] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. PhD thesis, Université Paris-Sud, July 1999.

[60] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.

[61] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, Mar. 2003. `http://www.lri.fr/~filliatr/ftp/publis/why-tool.ps.gz`.

[62] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification (CVA 07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer-Verlag.

[63] O. Fissore, I. Gnaedig, and H. Kirchner. CARIBOO: A multi-strategy termination proof tool based on induction. In A. Rubio, editor, *Extended Abstracts of the 6th International Workshop on Termination, WST'03*, pages 77–79, June 2003. Technical Report DSIC II/15/03, Univ. Politécnica de Valencia, Spain.

[64] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[65] R. Forgaard and D. Detlefs. Reve 2.4 : A program for generating and analyzing term rewriting systems. Massachusetts Institute, 1984.

[66] C. Fuhs, A. Middeldorp, P. Schneider-Kamp, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In *SAT 07*, volume 4501 of *Lecture Notes in Computer Science*, pages 340–354. Springer-Verlag, 2007.

[67] K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.

[68] J. Giesl. Generating polynomial orderings for termination proofs. In J. Hsiang, editor, *6th International Conference on Rewriting Techniques and Applications (RTA 95)*, volume 914 of *Lecture Notes in Computer Science*, pages 426–431, Kaiserslautern, Germany, Apr. 1995. Springer-Verlag.

[69] J. Giesl, T. Arts, and E. Ohlebusch. Modular Termination Proofs for Rewriting Using Dependency Pairs. *Journal of Symbolic Computation*, 34:21–58, 2002. doi:10.1006/jsco.2002.0541.

[70] J. Giesl and D. Kapur. Dependency Pairs for Equational Rewriting. In A. Middeldorp, editor, *12th International Conference on Rewriting Techniques and Applications (RTA 01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 93–108, Utrecht, The Netherlands, May 2001. Springer-Verlag.

[71] J. Giesl, P. Schneider-Kamp, and R. Thiemann. Aprove 1.2: Automatic termination proofs in the dependency pair framework. In U. Furbach and N. Shankar, editors, *Third International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, Seattle, USA, Aug. 2006. Springer-Verlag.

[72] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Improving dependency pairs. In *10th LPAR, LNAI 2850*, pages 165–179, 2003.

[73] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and Improving Dependency Pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

[74] J. Giesl and H. Zantema. Simulating liveness by reduction strategies. *Electronic Notes in Computer Science*, 86(4), 2003.

[75] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, 2000.

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

[77] B. Gramlich. Abstract relations between restricted termination and confluence properties of rewrite systems. *Fundamenta Informaticæ*, 24:3–23, 1995.

[78] B. Gramlich. On proving termination by innermost termination. In H. Ganzinger, editor, *7th International Conference on Rewriting Techniques and Applications (RTA 96)*, volume 1103 of *Lecture Notes in Computer Science*, pages 93–107, New Brunswick, NJ, USA, July 1996. Springer-Verlag.

[79] B. Gramlich. *Termination and Confluence Properties of Structured Rewrite Systems*. PhD thesis, Universität Kaiserslautern, 1996.

[80] J. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S. van Vlijmen, editors, *Algebra of Communicating Processes*, pages 26–62, 1995.

[81] R. Gutiérrez and S. Lucas. Proving Termination in the Context-Sensitive Dependency Pair Framework. In *8th International Workshop on Rewriting Logic and its Applications (WRLA 10)*, 2010. To appear.

[82] R. Gutierrez, S. Lucas, and X. Urbain. Usable rules for context-sensitive rewriting. In A. Voronkov, editor, *19th International Conference on Rewriting Techniques and Applications (RTA 08)*, volume 5117 of *Lecture Notes in Computer Science*, pages 126–141, Hagenberg, Austria, July 2008. Springer-Verlag.

[83] N. Hirokawa and A. Middeldorp. Tyrolean Termination Tool: Techniques and Features. *Information and Computation*, 205(4):474–511, 2007.

[84] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580 and 583, Oct. 1969.

[85] D. Hofbauer and C. Lautermann. Termination proofs and the length of derivations. In N. Dershowitz, editor, *International Conference on Rewriting Techniques and Applications (RTA 89)*, volume 355 of *Lecture Notes in Computer Science*, pages 167–177, Chapel Hill, U.S.A., Apr. 1989. Springer-Verlag.

[86] D. Hofbauer and J. Waldmann. Termination of $\{aa \to bc, bb \to ac, cc \to ab\}$. *Inf. Process. Lett.*, 98:156–158, 2006.

[87] D. Hofbauer and J. Waldmann. Termination of String Rewriting with Matrix Interpretations. In F. Pfenning, editor, *17th International Conference on Rewriting Techniques and Applications (RTA 06)*, volume 4098 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2006.

[88] H. Hong and D. Jakuš. Testing Positiveness of Polynomials. *Journal of Automated Reasoning*, 21(1):23–38, Aug. 1998.

[89] P. Hudak, S. J. Peyton-Jones, and P. Wadler. Report on the Functional Programming Language Haskell: a non–strict, purely functional language. *Sigplan Notices*, 27:1–164, 1992.

[90] G. Huet and D. S. Lankford. On the uniform halting problem for term rewriting systems. Research Report 283, INRIA, Mar. 1978.

[91] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Conference on Principles of Programming Languages*, pages 111–119, Munich, Germany, Jan. 1987. ACM Press.

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

[93] S. Kamin and J.-J. Lévy. Two generalizations of the recursive path ordering. Available as a report of the department of computer science, University of Illinois at Urbana-Champaign, 1980.

[94] D. Kapur and G. Sivakumar. A total, ground path ordering for proving termination of AC-rewrite systems. In H. Comon, editor, *8th International Conference on Rewriting Techniques and Applications (RTA 97)*, volume 1232 of *Lecture Notes in Computer Science*, Barcelona, Spain, June 1997. Springer-Verlag.

[95] D. Kapur and G. Sivakumar. Proving Associative-Commutative Termination Using RPO-compatible Orderings. In R. Caferra and G. Salzer, editors, *Automated Deduction in Classical and Non-Classical Logics*, number 1761 in Lecture Notes in Artificial Intelligence, pages 40–62. Springer-Verlag, Jan. 2000.

[96] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[97] A. Koprowski and J. Waldmann. Arctic termination ...below zero. In A. Voronkov, editor, *19th International Conference on Rewriting Techniques and Applications (RTA 08)*, volume 5117 of *Lecture Notes in Computer Science*, pages 202–216, Hagenberg, Austria, July 2008. Springer-Verlag.

[98] A. Koprowski and H. Zantema. Proving Liveness with Fairness Using Rewriting. In B. Gramlich, editor, *5th International Workshop on Frontiers of Combining Systems (FroCoS 05)*, volume 3717 of *Lecture Notes in Computer Science*, pages 232–247, Vienna, Austria, Sept. 2005. Springer-Verlag.

[99] M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In R. Treinen, editor, *20th International Conference on Rewriting Techniques and Applications (RTA 09)*, volume 5595 of *Lecture Notes in Computer Science*, pages 295–304, Brasília, Brazil, July 2009. Springer-Verlag.

[100] S. Kremer, A. Mercier, and R. Treinen. Proving Group Protocols Secure Against Eavesdroppers. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Fourth International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 116–131, Sydney, Australia, Aug. 2008. Springer-Verlag.

[101] M. R. K. Krishna Rao. Simple termination of hierarchical combinations of term rewriting systems. In *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 203–223. Springer-Verlag, 1994.

[102] M. R. K. Krishna Rao. Modular proofs for completeness of hierarchical term rewriting systems. *Theoretical Computer Science*, 151:487–512, 1995.

[103] M. Kurihara and A. Ohuchi. Decomposable Termination of Composable Term Rewriting Systems. *IEICE*, E78–D(4):314–320, Apr. 1995.

[104] K. Kusakari, C. Marché, and X. Urbain. Termination of associative-commutative rewriting using dependency pairs criteria. Technical Report 1304, L.R.I., 2002.

[105] K. Kusakari, M. Nakamura, and Y. Toyama. Argument filtering transformation. In G. Nadathur, editor, *Principles and Practice of Declarative Programming, International Conference PPDP'99*, volume 1702 of *Lecture Notes in Computer Science*, pages 47–61, Paris, 1999. Springer-Verlag.

[106] K. Kusakari and Y. Toyama. On proving AC-termination by AC-dependency pairs. *IEICE Transactions on Information and Systems*, E84-D(5):604–612, 2001.

[107] D. S. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Mathematics Department, Louisiana Tech. Univ., 1979.

[108] J. Lasseter, P. Docter, A. Stanton, and J. Ranft. Toy Story. Walt Disney Pictures, 1995.

[109] P. Lescanne. Computer experiments with the REVE term rewriting system generator. In *Proc. 10th ACM Symp. on Principles of Programming Languages, Austin, Texas*, 1983.

[110] P. Lescanne. Elementary interpretations in proofs of termination. *Formal Aspect of Computing*, 7:77–90, 1995.

[111] Liama Netquest Group. Netquest project page. *http://netquest.gforge.inria.fr*.

[112] I. Litovsky and Y. Métivier. Computing Trees with Graph Rewriting Systems with Priorities. *Tree Automata and Languages*, pages 115–139, 1992.

[113] I. Litovsky, Y. Métivier, and E. Sopena. Graph relabelling systems and distributed algorithms. In H. Ehrig, H. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 3, pages 1–56. World Scientific, 1999.

[114] S. Lucas. Context-Sensitive Computations in Functional and Functional Logic Programs. *Journal of Functional and Logic Programming*, 1998(1):1–61, 1998.

[115] S. Lucas. Context-Sensitive Rewriting Strategies. *Information and Computation*, 178(1):293–343, 2002.

[116] S. Lucas. MU-TERM: A Tool for Proving Termination of Context-Sensitive Rewriting. In V. van Oostrom, editor, *15th International Conference on Rewriting Techniques and Applications (RTA 04)*, volume 3091 of *Lecture Notes in Computer Science*, pages 200–209, Aachen, Germany, June 2004. Springer-Verlag. Available at `http://zenon.dsic.upv.es/muterm/`.

[117] S. Lucas. Polynomials over the reals in proofs of termination: from theory to practice. *R.A.I.R.O. : Informatique Théorique et Applications*, 39(3):547–586, 2005.

[118] S. Lucas. On the relative power of polynomials with real, rational, and integer coefficients in proofs of termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 17(1):49–73, 2006.

[119] S. Lucas, C. Marché, and J. Meseguer. Operational termination of conditional term rewriting systems. *Inf. Process. Lett.*, 95(4):446–453, Aug. 2005.

[120] Z. Manna and S. Ness. On the termination of Markov algorithms. In *Third Hawaii International Conference on Systems Sciences*, pages 789–792, Honolulu, USA, 1970. `http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html`.

[121] C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/-JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.

[122] C. Marché and X. Urbain. Termination of associative-commutative rewriting by dependency pairs. In T. Nipkow, editor, *9th International Conference on Rewriting Techniques and Applications (RTA 98)*, volume 1379 of *Lecture Notes in Computer Science*, pages 241–255, Tsukuba, Japan, Apr. 1998. Springer-Verlag.

[123] C. Marché and X. Urbain. Modular and incremental proofs of AC-termination. *Journal of Symbolic Computation*, 38:873–897, 2004.

[124] C. Marché and H. Zantema. The Termination Competition. In F. Baader, editor, *18th International Conference on Rewriting Techniques and Applications (RTA 07)*, volume 4533 of *Lecture Notes in Computer Science*, pages 303–313, Paris, France, June 2007. Springer-Verlag.

[125] M. Marchiori. Unravelings and ultra-properties. In M. Hanus and M. Rodríguez-Artalejo, editors, *Proc. of ALP'96*, volume 1039 of *Lecture Notes in Computer Science*, pages 107–121. Springer-Verlag, 1996.

[126] M. Marchiori. On Deterministic Conditional Rewriting. Technical Report 405, Massachusetts Institute of Technology, Laboratory for Computer Science, Dec. 1997.

[127] Y. V. Matiyasevich. Enumerable sets are Diophantine. *Soviet Mathematics (Dokladi)*, 11(2):354–357, 1970.

[128] Y. V. Matiyasevich. *Hilbert's Tenth Problem.* MIT Press, Oct. 1993.

[129] A. Middeldorp. Approximating dependency graphs using tree automata techniques. In R. Goré, A. Leitsch, and T. Nipkow, editors, *First International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 593–610, Siena, Italy, june 2001. Springer-Verlag.

[130] A. Middeldorp. Approximations for Strategies and Termination. *Electronic Notes in Computer Science*, 70(6), 2002.

[131] A. Middeldorp and Y. Toyama. Completeness of combinations of constructor systems. *Journal of Symbolic Computation*, 15:331–348, 1993.

[132] M. Mosbah and A. Sellami. Visidia: A tool for the VIsualization and SImulation of DIstributed Algorithms. *http://www.labri.fr/visidia/*.

[133] F. Neurauter and A. Middeldorp. Polynomial Interpretations over the Reals do not Subsume Polynomial Interpretations over the Integers. In C. Lynch, editor, *21st International Conference on Rewriting Techniques and Applications (RTA 10)*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 243–258, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

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

[135] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[136] E. Ohlebusch. On the modularity of termination of term rewriting systems. *Theoretical Computer Science*, 136:333–360, 1994.

[137] E. Ohlebusch. Modular Properties of Composable Term Rewriting Systems. *Journal of Symbolic Computation*, 20:1–41, 1995.

[138] E. Ohlebusch. Transforming conditional rewrite systems with extra variables into unconditional systems. In *6th International Conference on Logic for Programming and Automated Reasoning*, volume 1705 of *Lecture Notes in Artificial Intelligence*, pages 111–130, Berlin, 1999.

[139] E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer-Verlag, 2002.

[140] E. Ohlebusch, C. Claves, and C. Marché. TALP: A tool for the termination analysis of logic programs. In L. Bachmair, editor, *11th International Conference on Rewriting Techniques and Applications (RTA 00)*, volume 1833 of *Lecture Notes in Computer Science*, pages 270–273, Norwich, UK, July 2000. Springer-Verlag.

[141] G. E. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *J. ACM*, 28(2):233–264, Apr. 1981.

[142] D. A. Plaisted. A recursively defined ordering for proving termination of term rewriting systems. Unpublished report R-78-943, University of Illinois, Urbana, IL, 1978.

[143] J. Rouyer. A method to decide whether a multivariate integral polynomial admits roots in a product of closed intervals of R. Research Report 91-R-183, Centre de Recherche en Informatique de Nancy, 1991.

[144] J. Rouyer. Preuves de terminaison de systèmes de réécriture fondées sur les interprétations polynomiales. Une méthode basée sur le théorème de Sturm. *R.A.I.R.O.*, 25(2):157–169, 1991.

[145] A. Rubio. A fully syntactic AC-RPO. In P. Narendran and M. Rusinowitch, editors, *10th International Conference on Rewriting Techniques and Applications (RTA 99)*, volume 1631 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.

[146] M. Rusinowitch. On termination of the direct sum of term rewriting systems. *Inf. Process. Lett.*, 26:65–70, 1987.

[147] M. Rusinowitch. Path of subterms ordering and recursive decomposition ordering revisited. *Journal of Symbolic Computation*, 3(1), 1987.

[148] J. Steinbach. Proving polynomials positive. In R. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *Lecture Notes in Computer Science*, pages 191–202, New Delhi, India, Dec. 1992. Springer-Verlag.

[149] J. Steinbach. Simplification Orderings: History of results. *Fundamenta Informaticæ*, 24:47–88, 1995.

[150] S. Stratulat. Integrating Implicit Induction Proofs into Certified Proof Environments. In D. Méry and S. Merz, editors, *8th International Conference on Integrated Formal Methods*, volume 6396 of *Lecture Notes in Computer Science*, pages 320–335, Nancy, France, Oct. 2010. Springer-Verlag.

[151] The Coq Development Team. *The Coq Proof Assistant Documentation – Version V8.1*, Feb. 2007.

[152] The Coq Development Team. *The Coq Proof Assistant Documentation – Version V8.2*, June 2008.

[153] R. Thiemann. *The DP Framework for Proving Termination of Term Rewriting*. PhD thesis, RWTH Aachen, 2007.

[154] R. Thiemann, J. Giesl, and P. Schneider-Kamp. Improved Modular Termination Proofs Using Dependency Pairs. In *Second International Joint Conference on Automated Reasoning*, volume 3097 of *Lecture Notes in Artificial Intelligence*, pages 75–90, Cork, Ireland, 2004. Springer-Verlag.

[155] R. Thiemann and C. Sternagel. Certification of Termination Proofs using CeTa. In T. Nipkow and C. Urban, editors, *22st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09)*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468, Munich, Germany, Aug. 2009. Springer-Verlag.

[156] A. Tiwari. Termination of Linear Programs. In R. Alur and D. Peled, editors, *16th International Conference on Computer Aided Verification (CAV 04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 70–82, Boston, USA, July 2004. Springer-Verlag.

[157] Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Inf. Process. Lett.*, 25:141–143, Apr. 1987.

[158] X. Urbain. *Approche incrémentale des preuves automatiques de terminaison*. Thèse de doctorat, Université Paris-Sud, Orsay, France, Oct. 2001. `http://www.lri.fr/~urbain/textes/these.ps.gz`.

[159] X. Urbain. Automated Incremental Termination Proofs for Hierarchically defined Term Rewriting Systems. In R. Goré, A. Leitsch, and T. Nipkow, editors, *First International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 485–498, Siena, Italy, June 2001. Springer-Verlag.

[160] X. Urbain. Modular and incremental automated termination proofs. *Journal of Automated Reasoning*, 32(4):315–355, 2004.

[161] H. Zantema. Termination of String Rewriting Proved Automatically. *Journal of Automated Reasoning*, 34(2):105–139, 2005.